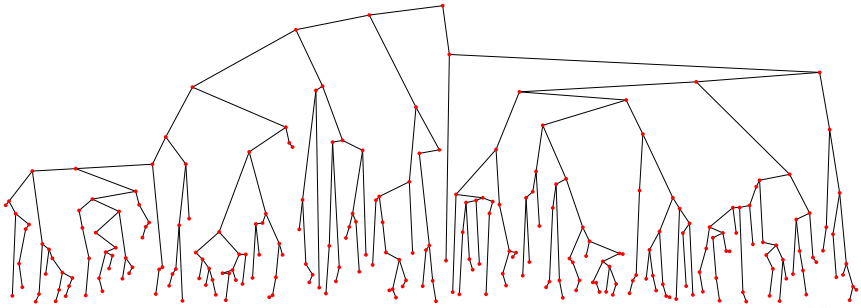


Open Data Structures (za programski jezik C++) v slovenščini

Izdaja 0.1Fβ

Pat Morin

Kazalo

1 Uvod **1**

1.1 Zahteva po učinkovitosti 2
1.2 Vmesniki 4
1.2.1 Vmesniki `Queue`, `Stack`, in `Deque` 4
1.2.2 Vmesnik `seznama`: linearne sekvence 6
1.2.3 Vmesnik `USet`: Neurejena množica 7
1.2.4 Vmesnik `SSet`: Urejena množica 8
1.3 Matematično ozdaje 9
1.3.1 Eksponenti in Logaritmi 9
1.3.2 Fakulteta 11
1.3.3 Asimptotična Notacija 11
1.3.4 Naključnost in verjetnost 15
1.4 Model računanja 18
1.5 Pravilnost, časovna in prostorska zahtevnost 19
1.6 Vzorci kode 21
1.7 Seznam Podatkovnih Struktur 22
1.8 Razprava in vaje 22

2 Izvedba seznama s poljem **29**

2.1 `ArrayStack`: Izvedba sklada s poljem 31
2.1.1 Osnove 31
2.1.2 Večanje in krčenje 33
2.1.3 Povzetek 35
2.2 `FastArrayStack`: Optimiziran ArrayStack 36
2.3 `ArrayQueue`: Vrsta na osnovi polja 37
2.3.1 Povzetek 40

2.4 ArrayDeque: Hitra obojestranska vrsta z uporabo polja . . . 41
2.4.1 Povzetek . . . 43
2.5 DualArrayDeque: Gradnja obojestranske vrste z dveh skladov . . . 43
2.5.1 Uravnoteženje . . . 47
2.5.2 Povzetek . . . 49
2.6 RootishArrayStack: Prostorsko učinkovit ArrayStack . . . 49
2.6.1 Analiza rasti in krčenja . . . 54
2.6.2 Poraba prostora . . . 54
2.6.3 Povzetek . . . 55
2.6.4 Računanje Kvadratnih Korenov . . . 56
2.7 Razprava in vaje . . . 59

3 Povezani seznam **63**
3.1 SLList: Enostransko povezani seznam . . . 64
3.1.1 Operacije Vrste . . . 66
3.1.2 Povzetek . . . 67
3.2 DLList: Obojestransko povezan seznam . . . 67
3.2.1 Dodajanje in odstranjevanje . . . 69
3.2.2 Povzetek . . . 71
3.3 SELList: Prostorsko učinkovit povezan seznam . . . 72
3.3.1 Prostorske zahteve . . . 73
3.3.2 Iskanje elementov . . . 73
3.3.3 Dodajanje elementov . . . 75
3.3.4 Odstranjevanje elementov . . . 78
3.3.5 Amortizirana analiza širjenja in združevanja . . . 79
3.3.6 Povzetek . . . 81
3.4 Razprave in vaje . . . 82

4 Preskočni seznami **89**
4.1 Osnovna struktura . . . 89
4.2 SkiplistSSet: Učinkovit SSet . . . 91
4.2.1 Povzetek . . . 95
4.3 SkiplistList: Učinkovit naključni dostop List . . . 95
4.3.1 Povzetek . . . 100
4.4 Analiza preskočnega seznama . . . 100

4.5 Razprava in vaje 104

5 Zgoščevalne tabele **109**
5.1 Zgoščevalna tabela z veriženjem 109
5.1.1 Zgoščevanje z množenjem 112
5.1.2 Povzetek 116
5.2 `LinearHashTable`: Linearno naslavljanje 116
5.2.1 Analiza odprtega naslavljanja 119
5.2.2 Povzetek 123
5.2.3 Tabelarno zgoščevanje 123
5.3 Zgoščene vrednosti 124
5.3.1 Zgoščene vrednosti osnovnih podatkovnih tipov . . 125
5.3.2 Zgoščene vrednosti sestavljenih podatkovnih tipov . 125
5.3.3 Zgoščevalne funkcije za polja in nize 127
5.4 Razprave in primeri 130

6 Dvojiška drevesa **135**
6.1 `BinaryTree`: Osnovno dvojiško drevo 137
6.1.1 Rekurzivni algoritmi 138
6.1.2 Obiskovanje dvojiškega drevesa 138
6.2 `BinarySearchTree`: Neuravnoteženo dvojiško iskalno drevo 141
6.2.1 Iskanje 142
6.2.2 Vstavljanje 143
6.2.3 Brisanje 146
6.2.4 Povzetek 148
6.3 `BinaryTree`: Razprava in vaje 148

7 Naključna iskalna binarna drevesa **153**
7.1 Naključna iskalna binarna drevesa 153
7.1.1 Dokaz 7.1 156
7.1.2 Povzetek 158
7.2 `Treap`: Naključno generirano binarno iskalno drevo 159
7.2.1 Povzetek 166

8 Drevesa "grešnega kozla" **169**
8.1 `ScapegoatTree`: Dvojiško iskalno drevo z delno rekonstrukcijo 170

8.1.1 Analiza pravilnosti in časovne kompleksnosti 173
8.1.2 Povzetek 176

9 Rdeče-Črna Drevesa **177**
9.1 2-4 Trees 178
9.1.1 Dodajanje lista 179
9.1.2 Odstranjevanje lista 179
9.2 `RedBlackTree`: Simulirano 2-4 drevo 182
9.2.1 Rdeče-Črna drevesa in 2-4 Drevesa 183
9.2.2 Levo-poravnana rdece-crna drevesa 186
9.2.3 Dodajanje 188
9.2.4 Odstranitev 191
9.3 Povzetek 196
9.4 Razprava in naloge 197

10 Kopice **203**
10.1 `BinaryHeap`: implicitno dvojiško drevo 203
10.1.1 Povzetek 207
10.2 `MeldableHeap`: Naključna zlivalna kopica 209
10.2.1 Analiza `merge(h1,h2)` 211
10.2.2 Povzetek 213
10.3 Diskusije in vaje 213

11 Algoritmi za urejanje **217**
11.1 Urenanje s primerjanjem 218
11.1.1 Urejanje z zlivanjem (merge-sort) 218
11.1.2 Hitro urejanje (quicksort) 222
11.1.3 Urejanje s kopico (heap-sort) 225
11.1.4 Spodnja meja algoritmov za urejanje, temelječih na primerjavah 227
11.2 Counting Sort and Radix Sort 230
11.2.1 Counting Sort 230
11.2.2 Radix-Sort 233
11.3 Diskusija in naloge 234

12 Grafi **237**
12.1 AdjacencyMatrix: Predstavitev grafov z uporabo matrik . 239
12.2 AdjacencyLists: Predstavitev grafov s seznamom sosednosti 242
12.3 Graph Traversal 246
12.3.1 Iskanje v širino 246
12.3.2 Iskanje v globino 248
12.4 Diskusija in vaje 251

13 Podatkovne strukture za cela števila **255**
13.1 BinaryTrie: digitalno iskalno drevo 256
13.2 XFastTrie: Iskanje v dvojnem logaritmičnem času 262
13.3 YFastTrie: Dvokratni-Logaritmični Čas SSet 265
13.4 Razprava in vaje 270

14 Iskanje v zunanjem pomnilniku **273**
14.1 Bločna shramba 275
14.2 B-drevesa 275
14.2.1 Iskanje 277
14.2.2 Dodajanje 280
14.2.3 Odstranjevanje 285
14.2.4 Amortizirana analiza *B*-Dreves 291
14.3 Razprave in vaje 294

Poglavje 1

Uvod

Vsak računalniški predmet na svetu vključuje snov o podatkovnih strukturah in algoritmih. Podatkovne strukture so *tako* pomembne; izboljšajo kvaliteto našega življenja in celo vsakodnevno rešujejo življenja. Veliko multimiljonskih in nekaj multimiljardnih družb je bilo ustanovljenih na osnovi podatkovnih struktur.

Kako je to možno? Če dobro pomislimo ugotovimo, da se s podatkovnimi strukturami srečujemo povsod.

- Odpiranje datoteke: podatkovne strukture datotečnega sistema se uporabljajo za iskanje delov datoteke na disku, kar ni preprosto. Diski vsebujejo stotine miljonov blokov, vsebina datoteke pa je lahko spravljena v kateremkoli od njih.

- Imenik na telefonu: podatkovna struktura se uporabi za iskanje telefonske številke v imeniku , glede na delno informacijo še preden končamo z vnosom iskalnega pojma. Naš imenik lahko vsebuje ogromno informacij - vsi, ki smo jih kadarkoli kontaktirali prek telefona ali elektronske pošte - telefon pa nima zelo hitrega procesorja ali veliko pomnilnika.

- Vpis v socialno omrežje: omrežni strežniki uporabljajo naše vpisne podatke za vpogled v naš račun. Največja socialna omrežja imajo stotine miljonov aktivnih uporabnikov.

- Spletno iskanje: iskalniki uporabljajo podatkovne strukture za iskanje spletnih strani, ki vsebujejo naše iskalne pojme. V internetu

je več kot 8.5 miljard spletnih strani, kjer vsaka vsebuje veliko potencialnih iskalnih pojmov, zato iskanje ni preprosto.

- Številke za klice v sili (112, 113): omrežje za storitve klicev v sili poišče našo telefonsko številko v podatkovni strukturi, da lahko gasilna, reševalna in policijska vozila pošlje na kraj nesreče brez zamud. To je pomembno, saj oseba, ki kliče mogoče ni zmožna zagotoviti pravilnega naslova in zamuda lahko pomeni razliko med življenjem in smrtjo.

1.1 Zahteva po učinkovitosti

V tem poglavju bomo pogledali operacije najbolj pogosto uporabljenih podatkovnih struktur. Vsak z vsaj malo programerskega znanja bo videl, da so te operacije lahke za implementacijo. Podatke lahko shranimo v polje ali povezan seznam, vsaka operacija pa je lahko implementirana s sprehodom čez polje ali povezan seznam in morebitnim dodajanjem ali brisanjem elementa.

Takšna implementacija je preprosta vendar ni učinkovita. Ali je to sploh pomembno? Računalniki postajajo vse hitrejši, zato je mogoče takšna implementacija dovolj dobra. Za odgovor naredimo nekaj izračunov.

Število operacij: predstavljajte si program z zmerno velikim naborom podatkov, recimo enim milijonom (10^6) elementov. V večini programov je logično sklepati, da bo program pregledal vsak element vsaj enkrat. To pomeni, da lahko pričakujemo vsaj milijon (10^6) iskanj. Če vsako od teh 10^6 iskanj pregleda vsakega od 10^6 elementov je to skupaj $10^6 \times 10^6 = 10^{12}$ (tisoč milijard) iskanj.

Procesorske hitrosti: v času pisanja celo zelo hiter namizni računalnik ne more opraviti več kot milijardo (10^9) operacij na sekundo. [1] To pomeni, da bo ta program porabil najmanj $10^{12}/10^9 = 1000$ sekund ali na grobo 16 minut in 40 sekund. Šestnajst minut je v računalniškem času

[1] Računalniške hitrosti se merijo v nekaj gigaherzih (milijarda ciklov na sekundo), kjer vsaka operacija zahteva nekaj ciklov.

ogromno, človeku pa bo to pomenilo veliko manj (sploh če si vzame odmor).

Večji nabori podatkov: predstavljajte si podjetje kot je Google, , ki upravlja z več kot 8.5 miljard spletnimi stranmi. Po naših izračunih bi kakršnakoli poizvedba med temi podatki trajala najmanj 8.5 sekund. Vendar vemo, da ni tako. Spletna iskanja se izvedejo veliko hitreje kot v 8.5 sekundah, hkrati pa opravljajo veliko zahtevnejše poizvedbe kot samo iskanje ali je določena stran na seznamu ali ne. V času našega pisanja Google prejme najmanj $4,500$ poizvedb na sekundo kar pomeni, da bi zahtevalo najmanj $4,500 \times 8.5 = 38,250$ zelo hitrih strežnikov samo za vzdrževanje.

Rešitev: ti primeri nam povedo, da preproste implementacije podatkovnih struktur ne delujejo ko sta tako število elementov, n, v podatkovni strukturi kot tudi število operacij, m, opravljenih na podatkovni strukturi, velika. V takih primerih je čas (merjen v korakih) na grobo $n \times m$.

Rešitev je premišljena organizacija podatkov v podatkovni strukturi tako, da vsaka operacija ne zahteva poizvedbe po vsakem elementu. Čeprav se sliši nemogoče bomo spoznali podatkovne strukture, kjer iskanje zahteva primerjavo samo dveh elementov v povprečju, neodvisno od števila elementov v podatkovni strukturi. V našem računalniku, ki opravi milijardo operacij na sekundo, zahteva iskanje v podatkovni strukturi, ki vsebuje milijardo elementov (ali več milijard), samo 0.000000002 sekund.

Pogledali bomo tudi implementacije podatkovnih struktur, ki hranijo elemente v vrstnem redu, kjer število poizvedenih elementov med operacijo raste zelo počasi v odvisnosti od števila elementov v podatkovni strukturi. Na primer, lahko vzdržujemo sortiran niz milijarde elementov, med poizvedbo do največ 60 elementov med katerokoli operacijo. V našem računalniku, ki opravi milijardo operacij na sekundo, zahteva izvajanje vsake izmed njih samo 0.00000006 sekund.

Preostanek tega poglavja vsebuje kratek pregled osnovnih pojmov, uporabljenih skozi celotno knjigo. **??** opisuje vmesnike, ki so implementirani z vsemi podatkovnimi strukturami opisanimi v tej knjigi in je smatran kot obvezno branje. Ostala poglavja so:

- pregled matematičnega dela, ki vključuje eksponente, logaritme, fa-

kultete, asimptotično (veliki O) notacijo, verjetnost in naključnost;

- računski model;
- pravilnost, časovna zahtevnost in prostorska zahtevnost;
- pregled ostalih poglavij;
- vzorčne kode in navodila za pisanje.

Bralec z ali brez podlage na tem področju lahko poglavja za zdaj enostavno preskoči in se vrne pozneje, če bo potrebno.

1.2 Vmesniki

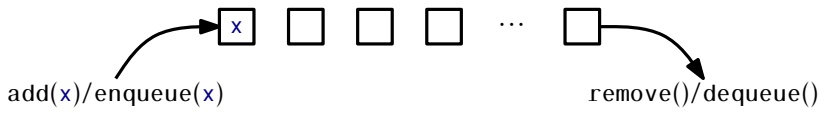
Pri razpravi o podatkovnih strukturah je pomembno poznati razliko med vmesnikom podatkovne strukture in njegovo implementacijo. Vmesnik opisuje kaj podatkovna struktura počne, medtem ko implementacija opisuje kako to počne.

Vmesnik, včasih imenovan tudi *abstrakten podatkovni tip*, definira množico operacij, ki so podprte s strani podatkovne strukture in semantiko oziroma pomenom teh operacij. Vmesnik nam ne pove nič o tem, kako podatkovna struktura implementira te operacije. Pove nam samo, katere operacije so podprte, vključno s specifikacijami o vrstah argumentov, ki jih vsaka operacija sprejme in vrednostmi, ki jih operacije vračajo.

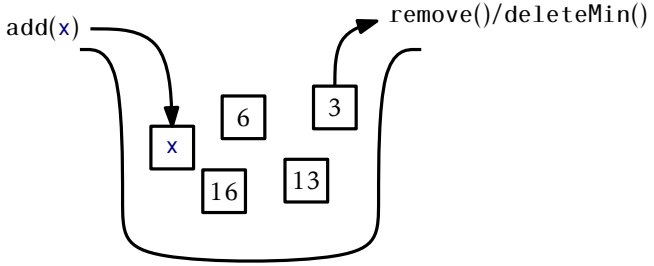
Implementacija podatkovne strukture, po drugi strani, vsebuje notranjo predstavitev podatkovne strukture, vključno z definicijami algoritmov, ki implementirajo operacije, podprte s strani podatkovne strukture. Zato imamo lahko veliko implementacij enega samega vmesnika. Na primer v 2 bomo videli implementacije vmesnika `seznama` z uporabo polj in v 3 bomo videli implementacije vmesnikov `seznama` z uporabo podatkovnih struktur, katere uporabljajo kazalce. Obe implementirajo isti vmesnik, `seznam`, vendar na drugačen način.

1.2.1 Vmesniki `Queue`, `Stack`, in `Deque`

Vmesnik `Queue` predstavlja zbirko elementov med katere lahko dodamo ali izbrišemo naslednji element. Bolj natančno, operaceije podprte z vme-



Slika 1.1: FIFO vrsta.



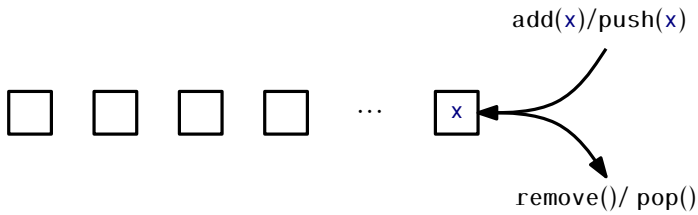
Slika 1.2: Vrsta s prednostjo.

snikom queue so

- add(x): dodaj vrednost x vrsti
- remove(): izbriši naslednjo (prej dodano) vrednost, y, iz vrste in vrni y

Opazimo lahko da metoda remove() ne sprejme nobenega argumenta. Implementacija vrste odloča kateri element bo izbrisan iz vrste. Poznamo veliko implementacij vrste, najbolj pogoste pa so FIFO, LIFO in vrste s prednostjo. *FIFO (first-in-first-out)* vrsta, ki je narisana v 1.1, odstrani elemente v enakem vrstnem redu kot so bili dodani, enako kot vrsta deluje, ko stojimo v vrsti za na blagajno v trgovini. To je najbolj pogosta implementaicija vrste, zato je kvalifikant FIFO pogosto izpuščen. V drugih besedilih se add(x) in remove() operacije na vrsti FIFO pogosto imenujejo enqueue(x) oziroma dequeue(x)

Vrste s prednostjo, prikazane na 1.2, vedno odstranijo najmanjši element iz vrste. To je podobno sistemu sprejema bolnikov v bolnicah. Ob prihodu zdravniki ocenijo poškodbo/bolezen bolnika in ga napotijo v čakalno sobo. Ko je zdravnik na voljo, prvo zdravi bolnika z najbolj smrtno nevarno poškodbo/boleznijo. V drugih besedilih je remove() operacija na vrsti s prednostjo ponavadi imenovana deleteMin().



Slika 1.3: sklad.

Zelo pogosta implementacija vrste je LIFO (last-in-first-out) prikazana na 1.3. Na *LIFO vrsti* je izbrisan nazadnje dodan element. To je najbolje prikazano s kupom krožnikov. Krožniki so postavljeni na vrh kupa, prav tako so odstranjeni iz vrha kupa. Ta struktura je tako pogosta, da je dobila svoje ime: `sklad`. Pogosto ko govorimo o `skladu`, so imena `add(x)` in `remove()` spremenjena v `push(x)` in `pop()`. S tem se izognemo zamenjavi implementacij vrst LIFO in FIFO.

`Deque` je generalizacija FIFO `vrste` in LIFO `vrste` (`sklad`). `Deque` predstavlja sekvenco elementov z začetkom in koncem. Elementi so lahko dodani na začetek ali pa na konec. Imena `deque` so samoumevna: `addFirst(x)`, `removeFirst()`, `addLast(x)` in `removeLast()`. `Sklad` je lahko implementiran samo z uporabo `addFirst(x)` in `removeFirst()`, medtem ko FIFO `vrsta` je lahko implementirana z uporabo `addLast(x)` in `removeFirst()`.

1.2.2 Vmesnik `seznama`: linearne sekvence

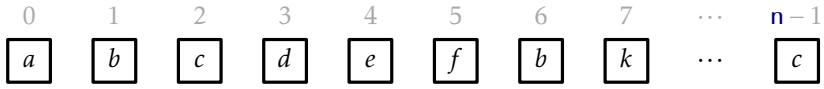
Ta knjiga govori zelo malo o FIFO `vrsti`, `skladu` ali `deque` vmesnikih, ker so vmesniki vključeni z vmesnikom `seznama`. Vmesnik `seznama` vključuje naslednje operacije:

1. `size()`: vrne `n`, dolžino seznama

2. `get(i)`: vrne vrednost x_i

3. `set(i,x)`: nastavi vrednost x_i na `x`

4. `add(i,x)`: doda `x` na mesto `i`, izrine $x_i, \ldots, x_{n-1}$;
   Nastavi $x_{j+1} = x_j$, za vse $j \in \{n-1, \ldots, i\}$, poveča `n`, in nastavi $x_i = x$



Slika 1.4: Seznam predstavlja sekvenco indeksov $0, 1, 2, \ldots, n-1$. V tem seznamu, bi klic get(2) vrnil vrednost c.

5. remove(i) :izbriše vrednost x_i, izrine $x_{i+1}, \ldots, x_{n-1}$;
   Nastavi $x_j = x_{j+1}$, za vse $j \in \{i, \ldots, n-2\}$ in zniža n

Opazimo lahko da te operacije enostavno lahko implementirajo deque vmesnik:

$$\begin{aligned} \texttt{addFirst}(\texttt{x}) &\Rightarrow \texttt{add}(0, \texttt{x}) \\ \texttt{removeFirst}() &\Rightarrow \texttt{remove}(0) \\ \texttt{addLast}(\texttt{x}) &\Rightarrow \texttt{add}(\texttt{size}(), \texttt{x}) \\ \texttt{removeLast}() &\Rightarrow \texttt{remove}(\texttt{size}() - 1) \end{aligned}$$

Čeprav ne bomo razpravljali o vmesnikih sklada, deque in FIFO vrste v podpoglavjih, sta izraza sklad in deque včasih uporabljena kot imeni podatkovnih struktur, ki implementirajo vmesnik seznama. V tem primeru želimo poudariti, da lahko te podatkovne strukture uporabimo za implementacijo vmesnika sklada in deque zelo efektivno. Na primer, ArrayDeque razred je implementacija vmesnika seznama, ki implementira vse deque operacije v konstantnem času na operacijo.

1.2.3 Vmesnik USet: Neurejena množica

USet vmesnik predstavlja neurejen set edinstvenih elementov, ki posnemajo matematični *set*. USet vsebuje n *različnih* elementov; noben element se ne pojavi več kot enkrat; elementi niso v nobenem določenem zaporedju. USet podpira naslednje operacije:

1. size(): vrne število, n, elementov v setu
2. add(x): doda element x v set, če ta že ni prisoten;
   Dodaj x setu, če ne obstaja tak element y v setu, da velja da je x enak y. Vrni true, če je bil x dodan v set, drugače false.

3. `remove(x)`: odstrani `x` iz seta;
   Najdi element `y` v setu, da velja da je `x` enak `y` in odstrani `y`. Vrni `y` ali `null`, če tak element ne obstaja.

4. `find(x)`: najde `x` v setu, če obstaja;
   Najdi element `y` v setu, da velja da je `y` enak `x`. Vrni `y` ali `null`, če tak element ne obstaja.

Te definicije se razlikujejo za razpoznavni element `x`, element, ki ga bomo odstranili ali našli, od elementa `y`, element, ki ga bomo verjetno odstranili ali našli. To je zato, ker sta `x` in `y` lahko različna objekta, ki sta lahko tretirana kot enaka. . Tako razlikovanje je uporabno, ker dovoljuje kreiranje *imenikov* ali *map*, ki preslika ključe v vrednosti.

Da naredimo imenik, eden tvori skupino objektov imenovanih `pari`, kateri vsebujejo *ključ* in a *vrednost*. Dva `para` sta si enakovredna, če so njuni ključi enaki. Če spravimo nek par `(k,v)` v `USet` in kasneje kličemo `find(x)` metodo z uporabo para `x = (k,null)` bi rezultat bil `y = (k,v)`. Z drugimi besedami povedano, možno je dobiti vrednost `v`, če podamo samo ključ `k`.

1.2.4 Vmesnik SSet: Urejena množica

Vmesnik `SSet` predstavlja urejen set elementov. `SSet` hrani elemente v nekem zaporedju, tako da sta lahko katera koli elementa `x` in `y` primerjana med sabo. V primeru bo to storjeno z metodo imenovano `compare(x,y)` v kateri

$$\texttt{compare(x,y)} \begin{cases} < 0 & \text{if } x < y \\ > 0 & \text{if } x > y \\ = 0 & \text{if } x = y \end{cases}$$

`SSet` podpira `size()`, `add(x)` in `remove(x)` metode z točno enako semantiko kot vmesnik `USet`. Razlika med `USet` in `SSet` je v metodi `find(x)`:

4. `find(x)`: locira `x` v urejenem setu;
   Najde najmanjši element `y` v setu, da velja $y \geq x$. Vrne `y` ali `null` če tak element ne obstaja.

Taka verzija metode `find(x)` je imenovana *iskanje naslednika*. Temeljno se razlikuje od `USet.find(x)`, saj vrne smiselen rezultat, tudi če v setu ni elementa, ki je enak `x`.

Razlika med `USet` in `SSet` `find(x)` operacijo je zelo pomembna in velikokrat prezrta. Dodatna funkcijonalnost priskrbljena s strani `SSet` ponavadi pride s ceno, da metoda porabi več časa za iskanje in večjo kompleknostjo kode. Na primer, večina implementacij `SSet` omenjenih v tej knjigi imajo `find(x)` operacije, ki potrebujejo logaritmičen čas glede na velikost podatkov. Na drugi strani ima implementacija `USet` kot `ChainedHashTable` v 5 `find(x)` operacijo, ki potrebuje konstanten pričakovani čas. Ko izbiramo katero od teh struktur bomo uporabili, bi vedno morali uporabiti `USet`, razen če je dodatna funkcionalnost, ki jo ponudi `SSet`, nujna.

1.3 Matematično ozdaje

V tem poglavju so opisane nekatere matematične notacije in orodja, ki so uporabljena v knjigi, vključno z logaritmi, veliko-O notacijo in verjetnostno teorijo. Opis ne bo natančen in ni mišljen kot uvod. Vsi bralci, ki mislijo da jim manjka osnovno znanje, si več lahko preberejo in naredijo nekaj nalog iz ustreznih poglavji zelo dobre in zastonj knjige o znanosti iz matematike in računalništva [**?**].

1.3.1 Eksponenti in Logaritmi

Izraz b^x označuje število b na potenco x. Če je x pozitivno celo število, potem je to samo število b pomnoženo samo s seboj $x-1$ krat:

$$b^x = \underbrace{b \times b \times \cdots \times b}_{x} \ .$$

Ko je x negativno celo število, je $b^x = 1/b^{-x}$. Ko je $x = 0$, $b^x = 1$. Ko b ni celo število, še vedno lahko definiramo potenciranje v smislu eksponentne funkcije e^x (glej spodaj), ki je definirana v smislu eksponentne serije, vendar jo je najboljše prepustiti računskemu besedilu.

V tej knjigi se izraz $\log_b k$ označuje *logaritem z osnovo-b* od k. To je edinstvena vrednost x za katero velja

$$b^x = k \ .$$

Večina logaritmov v tej knjigi ima osnovo 2 (*binarni logaritmi*).

Za te logaritme izpustimo osnovo, tako je $\log k$ skrajšan izraz za $\log_2 k$.

Neformalen ampak uporaben način je, da mislimo na $\log_b k$ kot število, koliko krat moramo deliti k z b, preden bo rezultat manjši ali enak 1. Na primer, ko izvedemo binarno iskanje, vsaka primerjava zmanjša število možnih odgovorov za faktor 2. To se ponavlja, dokler nam ne preostane samo en možen odgovor. Zato je število primerjav pri binarnem iskanju nad največ $n+1$ podatki enako največ $\lceil \log_2(n+1) \rceil$.

V knjigi se večkrat pojavi tudi *naravni logaritem*. Pri naravnem logaritmu uporabimo notacijo $\ln k$, ki označuje $\log_e k$, kjer je e — *Eulerjeva konstanta* — podan na naslednji način:

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \approx 2.71828 \ .$$

Naravni logaritem pride v poštev pogosto, ker je vrednost zelo pogostega integrala:

$$\int_1^k 1/x \, \mathrm{d}x = \ln k \ .$$

Dve najbolj pogosti operaciji, ki jih naredimo nad logaritmi sta, da jih umaknemo iz eksponenta:

$$b^{\log_b k} = k$$

in zamenjamo osnovo logaritma:

$$\log_b k = \frac{\log_a k}{\log_a b} \ .$$

Na primer, te dve operaciji lahko uporabimo za primerjavo naravnih in binarnin logaritmov.

$$\ln k = \frac{\log k}{\log e} = \frac{\log k}{(\ln e)/(\ln 2)} = (\ln 2)(\log k) \approx 0.693147 \log k \ .$$

1.3.2 Fakulteta

V enem ali dveh delih knjige je uporabljena *fakulteta*. Za nenegativna cela števila n je uporabljena notacija $n!$ (izgovorjena kot "n fakulteta") in pomeni naslednje:

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \ .$$

Fakulteta se pojavi, ker je $n!$ število različnih permutacij, naprimer zaporedja n različnih elementov.

Za poseben primer $n = 0$, je $0!$ definiran kot 1.

Vrednost $n!$ je lahko približno določena z uporabo *Stirlingovega približka*:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\alpha(n)} \ ,$$

kjer je

$$\frac{1}{12n+1} < \alpha(n) < \frac{1}{12n} \ .$$

Stirlingov približek prav tako približno določa $\ln(n!)$:

$$\ln(n!) = n\ln n - n + \frac{1}{2}\ln(2\pi n) + \alpha(n)$$

(V bistvu je Stirlingov približek najlažje dokazan z približevanjem $\ln(n!) = \ln 1 + \ln 2 + \cdots + \ln n$ z integralom $\int_1^n \ln n \, \mathrm{d}n = n\ln n - n + 1$.)

V relaciji s fakultetami so *binomski koeficienti*. Za nenegativna cela števila n in cela števila $K \in \{0, \ldots, n\}$, notacija $\binom{n}{k}$ označuje:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \ .$$

Binomski koeficient $\binom{n}{k}$ (izgovorjeno kot "n izbere k") šteje, koliko podmnožic elementa n ima velikost k, npr. število različnih možnosti pri izbiranju k različnih celih števil iz seta $\{1, \ldots, n\}$.

1.3.3 Asimptotična Notacija

Ko v knjigi analiziramo podatkovne strukture, želimo govoriti o časovnem poteku različnih operacij. Točen čas se bo seveda razlikoval od računalnika

do računalnika, pa tudi od izvedbe do izvedbe na določenem računalniku. Ko govorimo o časovni zahtevnosti operacije, se nanašamo na število inštrukcij opravljenih za določeno operacijo. Tudi za enostavno kodo je lahko to število težko za natančno določiti. Zato bomo namesto analiziranja natančnega časovnega poteka uporabljali tako imenovano *veliko-O notacijo*: Za funkcijo $f(n)$, $O(f(n))$ določi set funkciji

$$O(f(n)) = \left\{ \begin{array}{c} g(n) : \text{obstaja tak } c > 0, \text{ in } n_0 \text{ da velja} \\ g(n) \leq c \cdot f(n) \text{ za vse } n \geq n_0 \end{array} \right\} .$$

Grafično mišljeno ta set sestavljajo funkcije $g(n)$, kjer $c \cdot f(n)$ začne prevladovati nad $g(n)$ ko je n dovolj velik.

Po navadi uporabimo asimptotično notacijo za poenostavitev funkcij. Npr. na mesto $5n\log n+8n–200$ lahko zapišemo $O(n\log n)$. To je dokazano na naslednji način:

$$\begin{aligned} 5n\log n + 8n - 200 &\leq 5n\log n + 8n \\ &\leq 5n\log n + 8n\log n \qquad \text{za } n \geq 2 \text{ (zato da } \log n \geq 1) \\ &\leq 13n\log n \ . \end{aligned}$$

To dokazuja da je funkcija $f(n) = 5n\log n+8n–200$ v množici $O(n\log n)$ z uporabo konstant $c = 13$ in $n_0 = 2$.

Pri uporabi asimptotične notacije poznamo veliko bližnjic. Prva:

$$O(n^{c_1}) \subset O(n^{c_2}) \ ,$$

za vsak $c_1 < c_2$. Druga: Za katerokoli konstanto $a, b, c > 0$,

$$O(a) \subset O(\log n) \subset O(n^b) \subset O(c^n) \ .$$

Te relacije so lahko pomnožene s katerokoli pozitivno vrednostjo, brez da bi se spremenile. Npr. če pomnožimo z n, dobimo:

$$O(n) \subset O(n\log n) \subset O(n^{1+b}) \subset O(nc^n) \ .$$

Z nadaljevanjem dolge in ugledne tradicije bomo zapisali $f_1(n) = O(f(n))$, medtem ko želimo izraziti $f_1(n) \in O(f(n))$. Uporabili bomo tudi izjave kot so "časovna zahtevnost te operacije je $O(f(n))$", vendar pa bi izjava morala biti napisana "časovna zahtevnost te operacije je *element* $O(f(n))$." Te

krajšnjice se uporabljajo zgolj za to, da se izognemu nerodnemu jeziku in da lažje uporabimo asimptotično notacijo v besedilu enačb. Nenavaden primer tega se pojavi, ko napišemo izjavo:

$$T(n) = 2\log n + O(1) \ .$$

Bolj pravilno napisano kot

$$T(n) \leq 2\log n + [\text{član } O(1)] \ .$$

Izraz $O(1)$ predstavi nov problem. Ker v tem izrazu ni nobene spremenljivke, ni čisto jasno katera spremeljivka se samovoljno povečuje. Brez konteksta ne moremo vedeti. V zgornjem primeru, kjer je edina spremeljnivka n, lahko predpostavimo, da bi se izraz moral prebrati kot $T(n) = 2\log n + O(f(n))$, kjer $f(n) = 1$.

Velika-O notacija ni nova ali edinstvena v računalniški znanosti. Že leta 1894 jo je uporabljal številčni teoretik Paul Bachmann, saj je bila neizmerno uporabna za opis časovne zahtevnosti računalniških algoritmov.

Če upoštevamo naslednji del kode:

Simple

```
void  snippet() {
  for (int i = 0; i < n; i++)
    a[i] = i;
}
```

Ena izvedba te metode vključuje

- 1 dodelitev (`int i = 0`),
- `n` + 1 primerjav (`i < n`),
- `n` povečav (`i++`),
- `n` izračun odmikov v polju (`a[i]`),
- `n` posrednih dodelitev (`a[i] = i`).

Zato lahko napišemo časovno zahtevnost kot

$$T(\mathtt{n}) = a + b(\mathtt{n}+1) + c\mathtt{n} + d\mathtt{n} + e\mathtt{n} \ ,$$

kjer so a, b, c, d, in e konstante, ki so odvisne od naprave, ki izvaja kodo in predstavlja čas, v katerem se zaporedno izvedejo dodelitve, primerjave, povečevalne operacije, izračuni odmikov v poljih in posredne dodelitve. Če pa izraz predstavlja časovno zahtevnost dveh vrstic kode, potem se taka analiza ne more ujemati z zapleteno kodo ali algoritmi. Časovno zahtevnost lahko poenostavimo z uporabo velike-O notacije, tako dobimo

$$T(\mathrm{n}) = O(\mathrm{n}) \ .$$

Tak zapis je veliko bolj kompakten in nam hkrati da veliko informacij. To, da je časovna zahtevnost v zgornjem primeru odvisna od konstante a, b, c, d, in e, pomeni, da v splošnem ne bo mogoče primerjati dveh časov izvedbe, da bi razločili kateri je hitrejši, brez da bi vedeli vrednosti konstant. Tudi če uspemo določiti te konstante (npr. z časovnimi testi), bi naša ugotovitev veljala samo za napravo na kateri smo izvajali teste.

Velika-O notacija daje smisel analiziranju zapletenih funkcij pri višjih stopnjah. Če imata dva algoritma enako veliko-O časovno izvedbo, potem ne moremo točno vedeti, kateri je hitrejši in ni očitnega zmagovalca. En algoritem je lahko hitrejši na eni napravi, drugi pa na drugi napravi. Če imata dva algoritma dokazljivo različno veliki-O časovni izvedbi, potem smo lahko prepričani, da bo algoritem z manjšo časovno zahtevnostjo hitrejši *pri dovolj velikih vrednostih* n.

Kako lahko primerjamo veliko-O notacijo dveh različnih funkcij prikazuje 1.5, ki primerja stopnjo rasti $f_1(\mathrm{n}) = 15\mathrm{n}$ proti $f_2(n) = 2\mathrm{n}\log\mathrm{n}$. Npr., da je $f_1(n)$ časovna zahtevnost zapletenega linearnega časovnega algoritma in je $f_2(n)$ časovna zahtevnost bistveno preprostejšega algoritma, ki temelji na vzorcu deli in vladaj. Iz tega je razvidno, da čeprav je $f_1(\mathrm{n})$ večji od $f_2(n)$ pri manjših vrednostih n, velja nasprotno za velike vrednosti n. Po določenem času bo $f_1(\mathrm{n})$ zmagal zaradi stalne povečave širine marže. Analize, ki uporabljajo veliko-O notacijo, kažejo da se bo to zgodilo, ker je $O(\mathrm{n}) \subset O(\mathrm{n}\log\mathrm{n})$.

V nekaterih primerih bomo uporabili asimptotično notacijo na funkcijah z več kot eno spremenljivko. Predpisan ni noben standard, ampak za naš namen je naslednja definicija zadovoljiva:

$$O(f(n_1,\ldots,n_k)) = \left\{ \begin{array}{l} g(n_1,\ldots,n_k) : \text{obstaja } c > 0, \text{ in } z \text{ da velja} \\ \quad g(n_1,\ldots,n_k) \leq c \cdot f(n_1,\ldots,n_k) \\ \qquad \text{za vse } n_1,\ldots,n_k \text{ da velja } g(n_1,\ldots,n_k) \geq z \end{array} \right\} .$$

Ta definicija zajema položaj, ki nas zanima, ko g prevzame višje vrednosti zaradi argumenta $n_1,\ldots,n_k$. Ta definicija se sklada z univarijatno definicijo $O(f(n))$, ko je $f(n)$ naraščujoča funkcija n. Bralci naj bodo pozorni, da je lahko v drugih besedilih uporabljena asimptotična notacija drugače.

1.3.4 Naključnost in verjetnost

Nekatere podatkovne strukture predstavljene v knjigi so *naključne*; odločajo se naključno in neodvisno od podatkov, ki so spravljeni v njih in od opracij, ki se izvajajo nad njimi. Zaradi tega, se lahko časi izvajanja razlukujejo med seboj, kljub temu, da uporabimo enako zaporedje operacij nad strukturo. Ko analiziramo podatkovne strukture, nas zanima povprečje oziroma *pričakovan* čas poteka.

Formalno je čas poteka operacije na naključni podatkovni strukturi je naključna spremenljivka, želimo pa preučevati njeno *pričakovano vrednost*.

Za diskretno naključno spremenljivko X, ki zavzame vrednosti neke univerzalne množice U, je pričakovana vrednost X označena z $\mathrm{E}[X]$ podana z enačbo

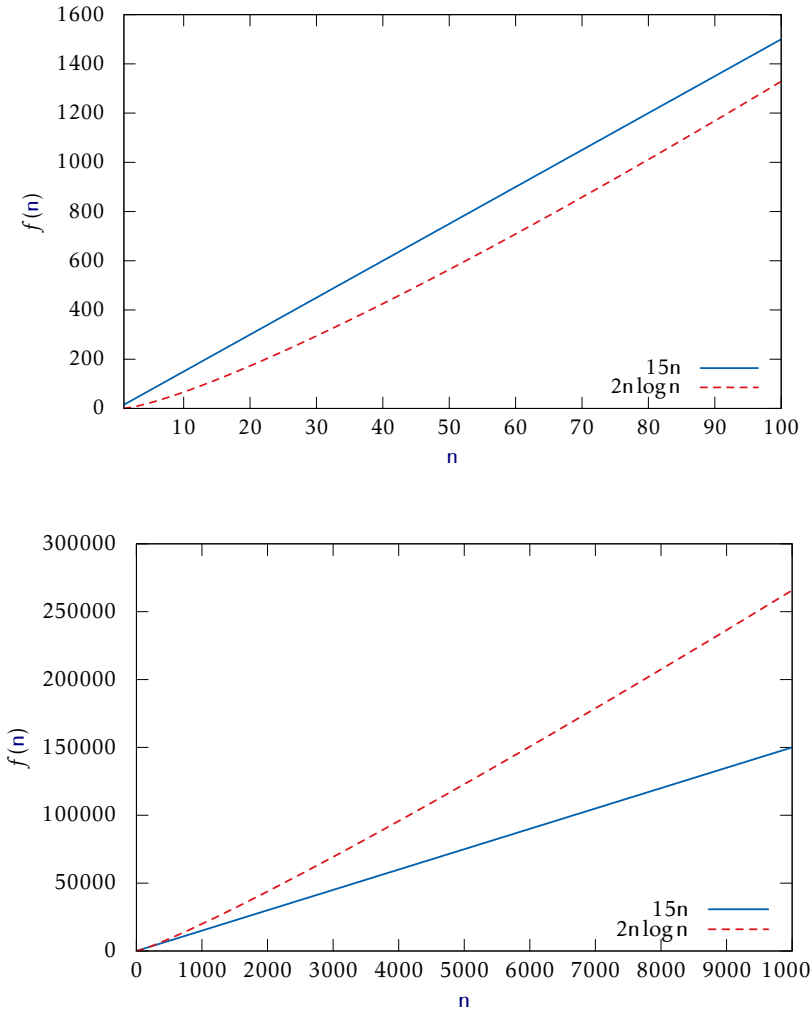
$$\mathrm{E}[X] = \sum_{x \in U} x \cdot \Pr\{X = x\} .$$

Tukaj $\Pr\{\mathcal{E}\}$ označuje verjetnost, da se pojavi dogodek $\mathcal{E}$. V vseh primerih v knjigi so te verjetnosti v spoštovanju z naključnimi odločitvami narejenimi s strani podatkovnih struktur. Ne moremo sklepati, da so naključni podatki, ki so shranjeni v strukturi, niti sekvence operacij izvedene na podatkovni strukturi.

Ena pomembnejših lastnosti pričakovane verjetnosti je *linearnost pričakovanja*.

Za katerekoli dve naključne spremenljivke X in Y,

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y] .$$



Slika 1.5: Plots of $15n$ versus $2n \log n$.

Bolj splošno, za katerokoli naključno spremenljivko $X_1, \ldots, X_k$,

$$\mathrm{E}\left[\sum_{i=1}^{k} X_k\right] = \sum_{i=1}^{k} \mathrm{E}[X_i] \ .$$

Linearnost pričakovanja nam dovoljuje, da razbijemo zapletene naključne spremenljivke(kot leva stran od zgornjih enačb) v vsote enostavnejših naključnih spremenljivk(desna stran).

Uporaben trik, ki ga bomo pogosto uporabljali, je definiranje indikatorja naključnih *spremenljivk*. Te binarne spremenljivke so uporabne, ko želimo nekaj šteti in so najbolje ponazorjene s primerom - vržemo pravičen kovanec k krat in želimo vedeti pričakovano število, koliko krat bo kovanec kazal glavo.

Intuitivno vemo, da je odgovor $k/2$. Če pa želimo to dokazati z definicijo pričakovane vrednosti, dobimo

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=0}^{k} i \cdot \Pr\{X = i\} \\
&= \sum_{i=0}^{k} i \cdot \binom{k}{i} / 2^k \\
&= k \cdot \sum_{i=0}^{k-1} \binom{k-1}{i} / 2^k \\
&= k/2 \ .
\end{aligned}$$

To zahteva, da vemo dovolj, da izračunamo, da $\Pr\{X = i\} = \binom{k}{i}/2^k$ in, da vemo binomske identitete $i\binom{k}{i} = k\binom{k-1}{i}$ in $\sum_{i=0}^{k}\binom{k}{i} = 2^k$.

Z uporabo indikatorskih spremenljivk in linearnostjo pričakovanja so stvari veliko lažje. Za vsak $i \in \{1, \ldots, k\}$ opredelimo indikatorsko naključno spremenljivko.

$$I_i = \begin{cases} 1 & \text{če je } i\text{ti met kovanca glava} \\ 0 & \text{drugače.} \end{cases}$$

Potem

$$\mathrm{E}[I_i] = (1/2)1 + (1/2)0 = 1/2 \ .$$

Sedaj $X = \sum_{i=1}^{k} I_i$ so

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{k} I_i\right] \\ &= \sum_{i=1}^{k} \mathrm{E}[I_i] \\ &= \sum_{i=1}^{k} 1/2 \\ &= k/2 \ . \end{aligned}$$

¡¡¡¡¡¡¡ .mine To je malo bolj zapleteno, vendar za to ne potrebujemo nobenih magičnih identitet ali računanja kakršnih koli ne trivijalnih verjetnosti. Še boljše, strinja se z intuicijo, da pričakujemo polovico kovancev, da pristanejo na glavi točno zato, ker vsak posamezni kovanec pristane na glavi z verjetnostjo 1/2.

1.4 Model računanja

V tej knjigi bomo analizirali teoretično časovno zahtevnost operacij na podatovnih strukturah, ki smo se jih učili. Da bi to natančneje preučili, potrebujemo formalni model računanja. Uporabljali bomo w-*bit besedni-RAM* model. RAM pomeni stroj z naključnim dostopom (Random Access Machine). V tem modelu imamo dostop do naključnega podatkovnega pomnilnika sestavljenega iz *celic*, pri katerih vsaka shranjuje w-bitno *besedo*. To pomeni, da lahko vsaka pomnilniška celica predstavlja, npr. vsa števila od $\{0, \ldots, 2^w - 1\}$.

V besedni-RAM modelu porabijo osnovne operacije konstanten čas. To so aritmetične operacije ($+$, $-$, $*$, $/$, $\%$), primerjave ($<$, $>$, $=$, $\leq$, $\geq$), in bitwise (vektor bitov) boolean (bitwise - IN, ALI, ekskluzivni ALI .)

V vsako celico lahko pišemo ali beremo v konstantnem času. Računalniški pomnilnik upravlja sistem, preko katerega lahko dodelimo ali ne, pomnilniški blok poljubne velikosti. Dodelitev pomnilniškega bloka velikosti k porabi $O(k)$ časa in vrne referenco (a pointer) do nazadnje dodeljenega pomnilniškega bloka. Ta referenca je dovolj majhna, da je lahko

predstavljena z eno samo besedo (zavzame prostor) v RAM-u.

Velikost besede w je zelo pomemben parameter v tem modelu. Edina predpostavka, ki jo bomo dodelili w -ju je spodnja meja $w \geq \log n$, kjer je n število elementov ki so shranjeni v naši podatkovni strukturi.

Pomnilniški prostor je merjen z besedami, tako, da ko govorimo koliko prostora zavzame podatkovna struktura, se sklicujemo na število besed, ki jih porabi struktura. Vse naše podatkovne strukture shranjujejo generično vrednost tipa T, predvidevamo pa, da element tipa T zasede eno besedo v pomnilniškem prostoru.

w-bit besedni-RAM model je približek modernim namiznim računalnikom ko je $w = 32$ ali $w = 64$. Podatkovne strukture, ki so uporabljene v tej knjigi ne uporabljajo nobenih specialnih metod, ki ne bi bile implementirane v C++ in večino drugih arhitektur.

1.5 Pravilnost, časovna in prostorska zahtevnost

Med učenjen uspešnosti podatkovnih struktur so najpomembjenše 3 stvari:

Pravilnost: podatkovna struktura mora pravilno implementirati svoj vmesnik

Časovna zahtevnost: operacijski časi v podatkovni strukturi morajo biti čim manjši

Prostorska zahtevnost: podatkovna struktura mora porabiti čim manj prostora

V tem uvodnem besedilu bomo uporabili pravilnost kot nam je podana; ne bomo predpostavljali, da podatkovne strukture podajajo napačne poizvedbe, ali da ne podajajo pravilnih posodobitev. Videli bomo, da podatkovne strukture stremijo k čim manjši porabi podatkovnega prostora.To ne bo vedno vplivalo na izvedbeni čas operacij, ampak lahko malce upočasnijo podatkovne strukture v praksi.

Med analiziranjen časovne zahtevnosti v kontekstu s podatkovnimi strukturami se nagibamo k 3 različnim možnostim:

Časovna zahtevnost v najslabšem primeru: : je najtrdnejša časovna zahtevnost, saj če imajo operacije v podatkovni strukturi časovno zahtevnost v najslabšem primeru enako $f(n)$, tpomeni, da nobena od teh operacij ne bo porabila več kot $f(n)$ časa.

Amortizirana časovna zahtevnost: če predpostavimo, da ima amortizirana časovna zahtevnost operacij v podatkovni strukturi časovno zahtevnost enako $f(n)$,), pomeni, da imajo operacije največjo zahtevnost enako $f(n)$. Natančneje pomeni, da če ima podatkovna struktura amortizirano časovno zahtevnost $f(n)$, potem zaporedje m operacij, porabi največ $mf(n)$ časa. Nekatere operacije lahko porabijo tudi več kot $f(n)$ časa, ampak je povprečje celotnega zaporedja operacij največ $f(n)$.

Pričakovana časovna zahtevnost: če predpostavimo, da je pričakovana časovna zahtevnost operacij na podatkovni strukturi enaka $f(n)$, pomeni, da je naključni čas delovanja enak naključni spremenljivki (glej 1.3.4) in pričakovana vrednost naključne spremenljivke je lahko največ $f(n)$. Naključna izbira v tem modelu podpira izbiro, ki jo izbere podatkovna struktura.

Da bi razumeli razliko med temi časovnimi zahtevnostmi, nam najbolj pomaga če si pogledamo primerjavo iz financ, pri nakupu nepremičnine:

Najslabši primer proti amortizirani ceni: Predpostavimo, da je cena nepremičnine \$120 000. Če želimo kupiti nepremičnino vzamemo 120 mesev (10 let) kredit, ki ga odplačujemo po \$1 200 na mesec. V tem primeru je najslabša možnost mesečnega plačila kredita enaka \$1 200 na mesec.

Če pa imamo dovolj denarja, se lahko odločimo za nakup nepremičnine z enkratnim plačilom \$120 000. V tem primeru, v obdobju 10 let, je amortizirana cena pri nakupu nepremičnine enaka:

$$\$120\,000/120 \text{ mesecev} = \$1\,000 \text{ na mesec} \ .$$

To je pa veliko manj, kot bi plačevali, če bi pri nakupu nepremičnine vzeli kredit.

Najslabši primer proti pričakovani ceni: Sedaj upoštevajmo zavarovanje proti požaru pri naši nepremičnini, ki je vredna $120 000 . Pri proučevanju tisočih primerov so zavarovalnice določile, da je požarna škoda pri taki nepremičnino kot je naša, enaka $10 na mesec. To je majhna številka, če predpostavimo, da veliko nepremičnin nikoli nima požara, nekatere imajo majhno škodo v primeru požara, najmanjše število pa je tistih, ki pri požaru zgorijo do tal. Upoštevajoč te podatke, zavarovalnice zaračunajo $15 mesečno za zavarovanje v primeru požara.

Sedaj je pa čas odločitve, ali naj v najslabšem primeru plačujemo $15 mesečno za zavarovanje v primeru požara, ali pa naj se sami zavarujemo in predpostavimo, da bi v primeru požara znašal $10 mesečno? Res je, $10 mesečno je manj kot je pričakovano, ampak moramo pa tudi sprejeti dejstvo, da bo strošek v primeru požara bistveno večji, saj če nepremičnina v primeru požara zgori do tal, bo ta strošek enak $120 000.

Te finančne primerjave nam prikažejo, zakaj se raje odločimo za amortizirano ali pričakovano časovno zahtevnost, kot časovno zahtevnost v najslabšem primeru. Večkrat je mogoče, da dobimo manjšo aqli amortizirano časovno zahtevnost, kot časovno zahtevnost v najslabšem primeru. Na koncu je pa še velikokrat mogoče, da dobimo preprostejšo podatkovno strukuro, če se odločimo za amortizirano ali pa pričakovano časovno zahtevnost.

1.6 Vzorci kode

Vzorci kode v tej knjigi so napisani v C++ .ampak, da bi bila ta knjiga bližje tudi bralcem, ki niso seznanjeni z C++ključnimi besedami so bili izrazi poenostavljeni. Na primer, bralci ne bodo naleteli na ključne besede kot so `public`, `protected`, `private`, or `static`. Bralec tudi ne bo naletel na diskusijo o hierarhiji razredov, razredih in vmesnikih ter podedovanju. Če bo to relevantno za bralca bo jasno razvidno iz teksta.

Ti dogovori bi morali narediti primere razumljive vsem z znanjem algoritemskih jezikov kot so B, C, C++, C#, Objective-C, D, Java, JavaScript, in tako dalje. Bralci, ki želijo vpogled v vse podrobnosti implementacij so dobrodošli, da si pogledajo C++ izvorno kodo, ki spremlja knjigo.

Ta knjiga je mešanica matematične analize izvajanja programom v

C++ . This means that To pomeni ,da nekatere enačbe vsebujejo spremenljivke, ki jih najdemo v izvorni kodi. Te spremenljivke so povsod uporabljene v istem pomenu, to velja za izvorno kodo kot tudi za enačbe. Na primer, pogosto uporabljena spremenljivka `n` je brez izjeme povsod uporabljena kot število, ki predstavlja število trenutno shranjenih vrednosti v podani podatkovni strukturi.

1.7 Seznam Podatkovnih Struktur

V tabelah 1.1 in 1.2 so povzete učinkovitosti podatkovnih struktur zajetih v tej knjigi, ki implementirajo vsakega od vmesnikov `List`, `USet`, and `SSet`, opisanih v **??**. 1.6 pokaže odvisnosti med različnimi poglavji zajetimi v knjigi. Črtkana puščica kaže le šibko odvisnost znotraj katere je le majhen del poglavja odvisen od prejšnjega poglavja ali samo glavnih rezultatov prejšnjega poglavja.

1.8 Razprava in vaje

Vmesniki `List`, `USet` in `SSet`, ki so opisani v poglavju **??** se kažejo kot vpliv Java Collections Framework [**?**]

V osnovi gre za poenostavljene vrezije `List`, `Set`, `Map`, `SortedSet` in `SortedMap` vmesnikov, ki jih najdemo v Java Collections Framework.

Za detajlno obravnavo in razumevanje matematične vsebine tega poglavja, ki vsebuje asimptotično notacijo, logaritme, fakulteto, Stirlingovo aproksimacijo, osnove verjetnosti in ostalo, vzemi v roke učbenik Lyman, Leighton in Meyer [**?**]. Za osnove matematične analize, ki obravnava definicije algoritmov in eksponentnih funkcij, se obrni na (prosto dostopno) besedilo, ki ga je spisal Thompson [**?**].

Več informacij o osnovah verjetnosti, predvsem področja, ki je tesno povezana z računalništvom, sezi po učbeniku Rossa [**?**]. Druga priporočljiva referenca, ki pokriva asimptotično notacijo in verjetnost, je učbenik Grahama, Knutha in Patashnika [**?**].

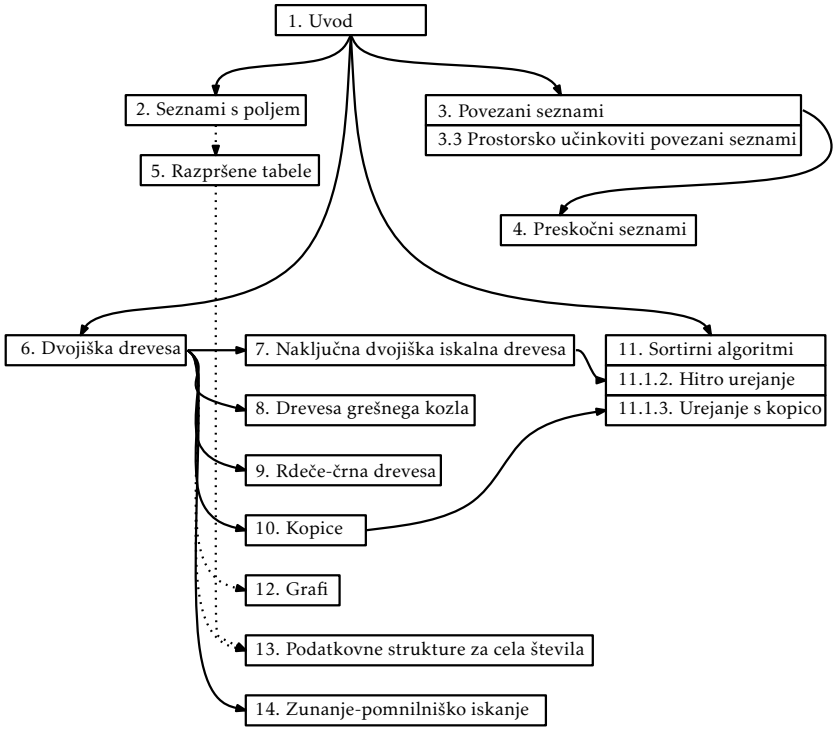
Naloga 1.1. Naloga je sestavljena tako, da bralca seznani s pravilnim izbiranjem najbolj ustrezne podatkovne strukture za dani primer. Če je del

List implementacije			
	get(i)/set(i,x)	add(i,x)/remove(i)	
ArrayStack	$O(1)$	$O(1+n-i)^{A}$	§ 2.1
ArrayDeque	$O(1)$	$O(1+\min\{i,n-i\})^{A}$	§ 2.4
DualArrayDeque	$O(1)$	$O(1+\min\{i,n-i\})^{A}$	§ 2.5
RootishArrayStack	$O(1)$	$O(1+n-i)^{A}$	§ 2.6
DLList	$O(1+\min\{i,n-i\})$	$O(1+\min\{i,n-i\})$	§ 3.2
SEList	$O(1+\min\{i,n-i\}/b)$	$O(b+\min\{i,n-i\}/b)^{A}$	§ 3.3
SkiplistList	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 4.3

USet implementacije			
	find(x)	add(x)/remove(x)	
ChainedHashTable	$O(1)^{E}$	$O(1)^{A,E}$	§ 5.1
LinearHashTable	$O(1)^{E}$	$O(1)^{A,E}$	§ **??**

[A] Označuje *amortizacijski* čas izvajanja.
[E] Označuje *pričakovani* čas izvajanja.

Tabela 1.1: Povzetek implementacij List in USet.



Slika 1.6: Odvisnosti med poglavji v tej knjigi.

SSet implementacije			
	`find(x)`	`add(x)`/`remove(x)`	
`SkiplistSSet`	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 4.2
`Treap`	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 7.2
`ScapegoatTree`	$O(\log n)$	$O(\log n)^{A}$	§ ??
`RedBlackTree`	$O(\log n)$	$O(\log n)$	§ 9.2
`BinaryTrie`[I]	$O(w)$	$O(w)$	§ 13.1
`XFastTrie`[I]	$O(\log w)^{A,E}$	$O(w)^{A,E}$	§ 13.2
`YFastTrie`[I]	$O(\log w)^{A,E}$	$O(\log w)^{A,E}$	§ 13.3

(Priority) Queue implementations			
	`findMin()`	`add(x)`/`remove()`	
`BinaryHeap`	$O(1)$	$O(\log n)^{A}$	§ 10.1
`MeldableHeap`	$O(1)$	$O(\log n)^{E}$	§ ??

[I] Ta struktura lahko shrani le `w`-bitne celoštevilske podatke.

Tabela 1.2: Povzetek implementacij `SSet` in priority `Queue`.

naloge že implementiran, potem je mišljeno, da se naloga reši s smiselno uporabo danega vmesnika (`Stack`, `Queque`, `Deque`, `Uset` ali `SSet`), ki ga priskrbi C++ Standard Template Library.

Problem reši tako, da nad vsako vrstico prebrane tekstovne datoteke izvršiš operacijo in pri tem uporabiš najbolj primerno podatkovno strukturo. Implementacija programa mora biti dovolj hitra, da obdela datoteko z milijon vnosi v nekaj sekundah.

1. Preberi vhod vrstico po vrstico in izpiši vrstice v obratnem vrstnem redu tako, da bo zadnji vnos izpisan prvi, predzadnji drugi in tako naprej.

2. Preberi prvih 50 vrstic vhoda in jih nato izpiši v obratnem vrstnem redu. Nato preberi naslednjih 50 vrstic in jih ponovno vrni v obratnem vrstnem redu. Slednje ponavljaj, dokler ne zmanjka vrstic vhoda. Ko program pride do točke, da je na vhodu manj kot 50 vrstic, naj vse preostale izpiše v obratnem vrstnem redu.

   Z drugimi besedami povedano, izhod se bo začel z ispisom 50. vr-

stice, nato 49. , za to 48. in vse tako do prve vrstice. Prvi vrstici bo sledila 100. vrstica vhoda, njej 99. in vse tako do 51. vrstice ter tako naprej.

Tekom izvajanja naj program v pomnilniku ne hrani več kot 50 vrstic naenkrat.

3. Beri vhod vrstico po vrstico. Program bere po 42 vrstic in če je katera od teh prazna (npr. niz dolžine nič), potem izpiše 42. vrstico pred to, ki je prazna. Na primer, če je 242. prazna, potem naj program izpiše 200. vrstico. Program naj bo implementiran tako, da v danem trenutku ne shranjuje več kot 43 vrstic vhoda naenkrat.

4. Beri vhod vrstico po vrstico in na izhod izpiši le tiste, ki so se na vhodu pojavile prvič. Bodi posebno pozoren na to, da datoteka, četudi ima veliko podvojenih vrstic, ne porabi več pomnilnika, kot je zahtevano za zapis unikatnih vrstic.

5. Beri vhod vrstico po vrstico in izpiši vse vrstice, ki so se vsaj enkrat že pojavile na vhodu (cilj je, da se izločijo unikatne vrstice vhoda). Bodi posebno pozoren na to, da datoteka, četudi ima veliko podvojenih vrstic, ne porabi več pomnilnika, kot je zahtevano za zapis unikatnih vrstic.

6. Preberi celoten vnos vrstico za vrstico in izpiši vse vrstice, razvrščene po velikosti, začenši z najkrajšo. Če sta dve vrstici enake dolžine, naj ju sortira "sorted order." Podvojene vrstice naj bodo izpisane samo enkrat.

7. Naredi enako kot pri prejšnji nalogi, le da so tokrat podvojene vrstice izpisane tolikokrat kolikor krat so bile vnesene.

8. Preberi celoten vnos vrstico za vrstico in izpiši najprej sode vrstice, začenši s prvo, vrstico 0, katerim naj sledijo lihe vrstice.

9. Preberi celoten vnos vrstico za vrstico, jih naključno premešaj in izpiši. Torej, ne sme se spremeniti vsebina vrstice, le njihov vrstni red naj se zamenja.

Naloga 1.2. *Dyck word* je sekvenca +1 in -1 z lastnostjo, da vsota katerekoli prepone zaporedja ni negativna. Na primer, $+1, -1, +1, -1$ je Dyck word, med tem ko $+1, -1, -1, +1$ ni Dyck word ker je predpona $+1-1-1 < 0$. Opiši katerokoli relacijo med `Sklad` `push(x)` in `pop()` operacijo.

Naloga 1.3. *Matched string* je zaporedje {, }, (,), [, in] znakov, ki se ustrezno ujemajo. Na primer, "{{()[]}}" je matched string, medtem ko "{{()]}" ni, saj se drugi { ujema z]. Pokaži kako uporabiti sklad, da za niz dolžine `n`, ugotoviš v $O(n)$ časa ali je matched string ali ne.

Naloga 1.4. Predpostavimo, da imamo `Sklad`, `s`, ki podpira samo operaciji `push(x)` in `pop()`. Pokaži kako lahko samo z uporabo FIFO `vrste`, `q`, obrnemo vrstni red vseh elementov v `s`.

Naloga 1.5. Z uporabo `USet`, implementiraj `Bag`. `Bag` je podoben `USet`—podpira metode `add(x)`, `remove(x)` in `find(x)` —ampak dovoljuje hrambo dvojnih elementov. `Find(x)` operacija v `Bag` vrne nekatere (če sploh kateri) element, ki je enak `x`. Poleg tega `Bag` podpira operacijo `findAll(x)`, ki vrne seznam vseh elementov, ki so enaki `x`.

Naloga 1.6. Iz samega začetka implementiraj in testiraj implementacijo vmesnikov `List`, `USet` in `SSet`, za katere ni nujno, da so učinkovite. Lahko so uporabljene za testiranje pravilnosti in zmogljivosti bolj učinkovitih implementacij. (Najlažji način za dosego tega je, da se shrani vse elemente v polje)

Naloga 1.7. Izboljšaj zmogljivost implementacije prejšnjega vprašanja z uporabo kateregakoli trika, ki ti pade na pamet. Eksperimentiraj in razmisli o tem, kako bi lahko izboljšal zmogljivost implementacij `add(i,x)` in `remove(i)` v svoji implementaciji vmesnika `List`. Razmisli, kako bi se dalo izboljšati zmogljivost operacije `find(x)` tvoje implementacije `USet` in `SSet`. Ta naloga je zasnovana tako, da ti predstavi kako težko je doseči učinkovitost v implementaciji teh vmesnikov.

Poglavje 2

Izvedba seznama s poljem

V tem poglavju si bomo pogledali izvedbe vmesnikov `Seznama` in `Vrste`, kjer je osnoven podatek hranjen v polju, imenovanem *podporno polje*. V spodnji tabeli imamo prikazane časovne zahtevnosti operacij za podatkovne strukture predstavljene v tem poglavju:

	`get(i)`/`set(i,x)`	`add(i,x)`/`remove(i)`
`ArrayStack`	$O(1)$	$O(n-i)$
`ArrayDeque`	$O(1)$	$O(\min\{i, n-i\})$
`DualArrayDeque`	$O(1)$	$O(\min\{i, n-i\})$
`RootishArrayStack`	$O(1)$	$O(n-i)$

Podatkovne strukture, kjer podatke shranjujemo v enojno polje imajo veliko prednosti, a tudi omejitev:

- V polju imamo vedno konstantni čas za dostop do kateregakoli podatka. To nam omogoča, da se operaciji `get(i)` in `set(i,x)` izvedeta v konstantnem času.

- Polja niso dinamična. Če želimo vstaviti ali izbrisati element v sredini polja moramo premakniti veliko elementov, da naredimo prostor za novo vstavljen element oz. da zapolnimo praznino po tem, ko smo element izbrisali. Zato je časovna zahtevnost operacij `add(i,x)` in `remove(i)` odvisna od spremenljivk `n` in `i`.

- Polja ne moremo širiti ali krčiti. Ko imamo večje število elementov, kot je veliko naše podporno polje, moramo ustvariti novo, dovolj

veliko polje, v katerega kopiramo podatke iz prejšnjega polja. Ta operacija pa je zelo draga.

Tretja točka je zelo pomembna, saj časovne zahtevnosti iz zgornje tabele ne vključujejo spreminjanja velikosti polja. V nadaljevanju bomo videli, da širjenje in krčenje polja ne dodata veliko k *povprečni* časovni zahtevnosti, če jih ustrezno upravljamo. Natančneje, če začnemo s prazno podatkovno strukturo in izvedemo zaporedje operacij m `add(i,x)` ali `remove(i)` ,potem bo časovna zahtevnost širjenja in krčenja polja za m operacij $O(m)$. Čeprav so nekatere operacije dražje je povprečna časovna zahtevnost nad vsemi m operacijami samo $O(1)$ za operacijo.

V tem poglavju in v celotni knjigi je priročno uporabljati polja, ki imajo števec za velikost. Navadna polja v C++ nimajo te funkcije, zato definiramo razred, `array`, ki hrani dolžino polja. Implementacija tega razreda je enostavna. Implementiran je kot običajno C++ polje, `a`, in število, `length`:

array

```
T *a;
int length;
```

Velikost polja `array` je določena od kreaciji:

array

```
array(int len) {
  length = len;
  a = new T[length];
}
```

Elementi v polju so lahko indeksirani:

array

```
T& operator[](int i) {
  assert(i >= 0 && i < length);
  return a[i];
}
```

Na koncu, ko imamo eno polje dodeljeno drugemu, potrebujemo samo še premikanje kazalca, ki pa se izvede v konstantnem času:

array

```
array<T>& operator=(array<T> &b) {
  if (a != NULL) delete[] a;
```

```
  a = b.a;
  b.a = NULL;
  length = b.length;
  return *this;
}
```

2.1 ArrayStack: Izvedba sklada s poljem

Z operacijo ArrayStack implementiramo vmesnik za seznam z uporabo polja a, imenovanega the *podporno polje*. Element v seznamu na indeksu i je hranjen v a[i]. V večini primerov je velikost polja a večja, kot je potrebno, zato uporabimo število n kot števec števila elementov spravljenih v polju a. Tako imamo elemente spravljene v a[0],...,a[n − 1] in v vseh primerih velja, a.length ≥ n.

ArrayStack

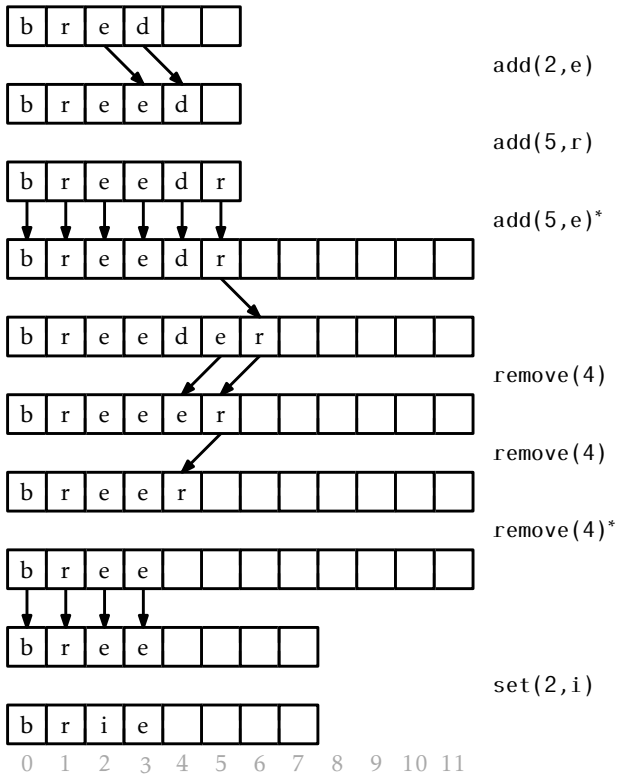
```
array<T> a;
int n;
int size() {
  return n;
}
```

2.1.1 Osnove

Dostop in spreminjanje elementov v ArrayStack z uporabo operacij get(i) in set(i,x) je zelo lahko. Po izvedbi potrebnih mejnih preverjanj polja vrnemo množico oz. a[i].

ArrayStack

```
T get(int i) {
  return a[i];
}
T set(int i, T x) {
  T y = a[i];
  a[i] = x;
  return y;
}
```



Slika 2.1: Zaporedje operacij `add(i,x)` in `remove(i)` v `ArrayStack`. Puščice označujejo elemente, ki jih je potrebno kopirati. Operacije, po katerih moramo klicati metodo `resize()` so označene z zvezdico.

Operaciji vstavljanja in brisanja elementov iz `ArrayStack` sta predstavljeni v 2.1. Za implementacijo `add(i,x)` operacije najprej preverimo če je polje `a` polno. Če je, kličemo metodo `resize()` za povečanje velikosti polja `a`. Kako je metoda `resize()` implementirana, si bomo pogledali kasneje, saj nas trenutno zanima samo to, da potem, ko kličemo metodo `resize()` še vedno ohranjamo pogoj `a.length` > `n`. Sedaj lahko premaknemo elemente $a[i],\ldots,a[n-1]$ za ena v desno, da naredimo prostor za `x`, množico $a[i]$ spravimo v `x` in povečamo `n`, saj smo vstavili nov element.

ArrayStack
```
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  for (int j = n; j > i; j--)
    a[j] = a[j - 1];
  a[i] = x;
  n++;
}
```

Če zapostavimo časovno zahtevnost ob morebitnem klicanju metode `resize()`, potem je časovna zahtevnost operacije `add(i,x)` sorazmerna številu elementov, ki jih moramo premakniti, da naredimo prostor za novo vstavljen element `x`. Zato je časovna zahtevnost operacije (zanemarimo časovno zahtevnost spreminjanja polja `a`) $O(n-i)$.

Implementacija operacije `remove(i)` je zelo podobna. Premaknemo elemente $a[i+1],\ldots,a[n-1]$ za ena v levo (prepišemo $a[i]$) in zmanjšamo vrednost `n`. Potem preverimo, če števec `n` postaja občutno manjši kot `a.length` s preverjanjem $a.length \geq 3n$. Če je občutno manjši kličemo metodo `resize()` za zmanjšanje velikosti polja `a`.

ArrayStack
```
T remove(int i) {
    T x = a[i];
  for (int j = i; j < n - 1; j++)
    a[j] = a[j + 1];
  n--;
  if (a.length >= 3 * n) resize();
  return x;
}
```

Če zanemarimo časovno zahtevnost metode `resize()` je časovna zahtevnost operacije `remove(i)` sorazmerna s številom elementov, ki jih moramo premakniti. To pomeni, da je časovna zahtevnost $O(n-i)$.

2.1.2 Večanje in krčenje

Metoda `resize()` je dokaj enostavna; alocira novo polje `b` velikosti $2n$ in skopira `n` elementov iz polja `a` v prvih `n` mest polja `b` in nato postavi `a` v `b`. Tako po klicu `resize()`, $a.length = 2n$.

ArrayStack

```
void resize() {
  array<T> b(max(2 * n, 1));
  for (int i = 0; i < n; i++)
    b[i] = a[i];
  a = b;
}
```

Analiza cene operacije `resize()` je lahka. Metoda naredi polje `b` velikosti $2n$ in kopira `n` elementov iz `a` v `b`. To traja $O(n)$ časa.

Pri analizi časa delovanja iz prejšnjega poglavja ni bila všteta cena klica `resize()` funkcije. V tem poglavju bomo analizirali to ceno z uporabo tehnike znane pod imenom *amortizirana analiza*. Ta način ne poskuša ugotoviti cene za spreminjanje velikosti med vsako `add(i,x)` in `remove(i)` operacijo. Namesto tega, se posveti ceni vseh klicev `resize()` med zaporedjem m klicev funkcije `add(i,x)` ali `remove(i)`.

Predvsem pokažemo:

Lema 2.1. *Če je ustvarjen prazen* `ArrayList` *in katerokoli zaporedje, ko je* $m \geq 1$ *kliče* `add(i,x)` *ali* `remove(i)` *potem je skupen porabljen čas za vse klice* `resize()` *enak* $O(m)$.

Dokaz. Pokazali bomo, da vsakič ko je klican `resize()`, je število klicev `add` ali `remove` od zadnjega klica `resize()` funkcije, vsaj $n/2-1$. Torej, če n_i označuje vrednost `n` med $i$tim klicem metode `resize()` in r označuje število klicev funkcije `resize()`, potem je skupno število klicev `add(i,x)` ali `remove(i)` vsaj

$$\sum_{i=1}^{r} (n_i/2 - 1) \leq m \ ,$$

kar je enako kot

$$\sum_{i=1}^{r} n_i \leq 2m + 2r \ .$$

Na drugi strani, je skupno število časa uporabljenega med vsem `resize()` klici enako

$$\sum_{i=1}^{r} O(n_i) \leq O(m+r) = O(m) \ ,$$

ker r ni več kot m. Vse kar nam ostane je pokazati, da je število klicev `add(i,x)` ali `remove(i)` med $(i-1)$tim in $i$tim klicem za `resize()` enako vsaj $n_i/2$.

Upoštevati moramo dva primera. V prvem primeru, je bila metoda `resize()` klicana s strani funkcije `add(i,x)`, ker je bilo polje `a` polno, t.j., `a.length` $= n = n_i$. Gledano na prejšnji klic funkcije `resize()`: je bila velikost `a`-ja po klicu enaka `a.length`, vendar je bilo število elementov shranjenih v `a`-ju največ `a.length`$/2 = n_i/2$. Zdaj pa je število elementov shranjenih v `a` enako $n_i =$ `a.length`, torej se je moralo, od prejšnjega klica `resize()` izvesti vsaj $n_i/2$ klicev `add(i,x)`. Drugi primer se zgodi, ko je `resize()` klicana s strani funkcije `remove(i)`, ker je `a.length` $\geq 3n = 3n_i$. Enako kot prej je po prejšnjemu klicu `resize()` bilo število elementov shranjenih v `a` najmanj `a.length`$/2-1$.[1] Zdaj pa je v `a` shranjenih $n_i \leq$ `a.length`$/3$ elementov. Zato je število `remove(i)` operacij od zadnjega `resize()` klica vsaj

$$\begin{aligned} R &\geq \texttt{a.length}/2 - 1 - \texttt{a.length}/3 \\ &= \texttt{a.length}/6 - 1 \\ &= (\texttt{a.length}/3)/2 - 1 \\ &\geq n_i/2 - 1 \ . \end{aligned}$$

V vsakem primeru je število klicev `add(i,x)` ali `remove(i)`, ki se zgodijo med $(i-1)$tim klicem za `resize()` in $i$tim klicem za `resize()` je natanko toliko $n_i/2-1$, kot je tudi potrebno za dokončanje dokaza. □

2.1.3 Povzetek

Naslednji izrek povzema učinkovitost izvedbe podatkovne strukture `ArrayStack`:

Izrek 2.1. *`ArrayStack` implementira `List` vmesnik. Z ignoriranjem cene klicev funkcije* `resize()` *`ArrayStack` podpira naslednje operacije:*

- `get(i)` *in* `set(i,x)` *v času* $O(1)$ *a eno operacijo; in*
- `add(i,x)` *in* `remove(i)` *v času* $O(1+n-i)$ *na operacijo.*

[1] -1 v tej formuli pomeni poseben primer ko je `n` $= 0$ in `a.length` $= 1$.

Poleg tega, če začnemo z prazno strukturo `ArrayStack` *in potem izvajamo katerokoli zaporedje od* m `add(i,x)` *in* `remove(i)` *operacij privede v skupno* $O(m)$ *časa uporabljenega med vsem klici funkcije* `resize()`.

`ArrayStack` je učinkovit način za implementiranje `Sklada`. Funkcijo `push(x)` lahko implementiramo kot `add(n,x)` in funkcijo `pop()` kot `remove(n-1)`, V tem primeru bodo te operacije potrebovale $O(1)$ amortiziranega časa.

2.2 FastArrayStack: Optimiziran ArrayStack

`ArrayStack` opravi večino dela z zamenjevanjem (s `add(i,x)` in `remove(i)`) in kopiranjem (z `resize()`) podatkov. V izvedbah prikazanih zgoraj, je bilo to narejeno s pomočjo `for` zanke. Izkaže se, da ima veliko programskih okolij posebne funkcije, ki so zelo učinkovite pri kopiranju in premikanju blokov podatkov. V programskem jeziku C, obstajajo funkcije `memcpy(d,s,n)` in `memmove(d,s,n)`. V C++ jeziku je `std::copy(a0,a1,b)` algoritem. V Javi je metoda `System.arraycopy(s,i,d,j,n)`.

FastArrayStack

```
void resize() {
  array<T> b(max(1, 2*n));
  std::copy(a+0, a+n, b+0);
  a = b;
}
void add(int i, T x) {
  if (n + 1 > a.length) resize();
  std::copy_backward(a+i, a+n, a+n);
  a[i] = x;
  n++;
}
```

Te funkcije so ponavadi zelo optimizirane in lahko uporabljajo tudi posebne strojne ukaze, ki lahko kopirajo veliko hitreje, kot z uporabo zanke `for`. Vseeno s pomočjo teh funkcij ne moremo asimptotično zmanjšati izvajalnih časov, a je ta optimizacija še vedno koristna. V C++ izvedbah Jave, uporaba nativnega povzroči pohitritve za faktor med 2 in 3, odvi-

sno od vrste izvajanih operacij. Izvajane pohitritve se lahko razlikujejo od sistema do sistema.

2.3 ArrayQueue: Vrsta na osnovi polja

V tem poglavju bomo predstavili podatkovno strukturo ArrayQueue, ki implementira FIFO vrsto; elemente iz vrste odstranjujemo (z uporabo operacije remove()) v istem vrstnem redu, kot so bili dodani (z uporabo operacije add(x)).

Opazimo, da ArrayStack ni dobra izbira za izvedbo FIFO vrste in sicer zato, ker moramo izbrati en konec seznama, na katerega dodajamo elemente, nato pa elemente odstranjujemo z drugega konca. Ena izmed operacij mora delovati na glavi seznama, kar vključuje klicanje add(i,x) ali remove(i), kjer je vrednost $i = 0$. To nudi čas izvajanja sorazmeren n.

Da bi dosegli učinkovito implementacijo vrste na osnovi seznama, najprej opazimo, da bi bil problem enostaven, če bi imeli neskočno veliko polje a. Lahko bi hranili indeks j, ki hrani naslednji element za odstranitev ter celo število n, ki šteje število elementov v vrsti. Elementi vrste bi bili vedno shranjeni v

$$a[j], a[j+1], \ldots, a[j+n-1] \ .$$

Sprva bi bila j in n nastavljena na 0. Na novo dodan element bi uvrstili v $a[j+n]$ in povečali n. Za odstranitev elementa bi ga odstranili iz $a[j]$, povečali j in zmanjšali n.

Težava te rešitve je potreba po neskončno velikem polju. ArrayQueue to simulira z uporabo končnega polja in *kongruence*. To je vrsta aritmetike, ki jo uporabljamo pri izračunu časa. Na primer 10:00 plus pet ur je 3:00. Formalno pravimo, da je

$$10 + 5 = 15 \equiv 3 \pmod{12} \ .$$

Zadnji del enačbe beremo kot "15 je skladno s 3 po modulu 12." Operator mod lahko obravnavamo tudi kot binarni operator, da je

$$15 \bmod 12 = 3 \ .$$

V splošnem je za celo število a in pozitivno celo število m, a mod m enolično celo število $r \in \{0,\ldots,m-1\}$ tako, da velja $a = r + km$ za poljubno celo število k. Poenostavljeno vrednost r predstavlja ostanek pri deljenju a z m. V večini programskih jezikov, vključno s C++, je operator mod predstavljen z znakom %.[2]

Modularna aritmetika je uporabna za simulacijo neskončno velikega polja, ker `i` mod `a.length` vedno vrne vrednost na intervalu $0,\ldots,$ `a.length` -1. Z uporabo kongruence lahko elemente vrste shranimo na naslednja mesta v polju

$$\texttt{a[j\%a.length]}, \texttt{a[(j+1)\%a.length]}, \ldots, \texttt{a[(j+n-1)\%a.length]} \ .$$

To obravnava polje `a` kot *krožno polje* kjer polje indekse večje kot `a.length`-1 "ovije naokrog" na začetek polja.

Paziti moramo le še, da število elementov v `ArrayQueue` ne preseže velikosti `a`.

ArrayQueue

```
array<T> a;
int j;
int n;
```

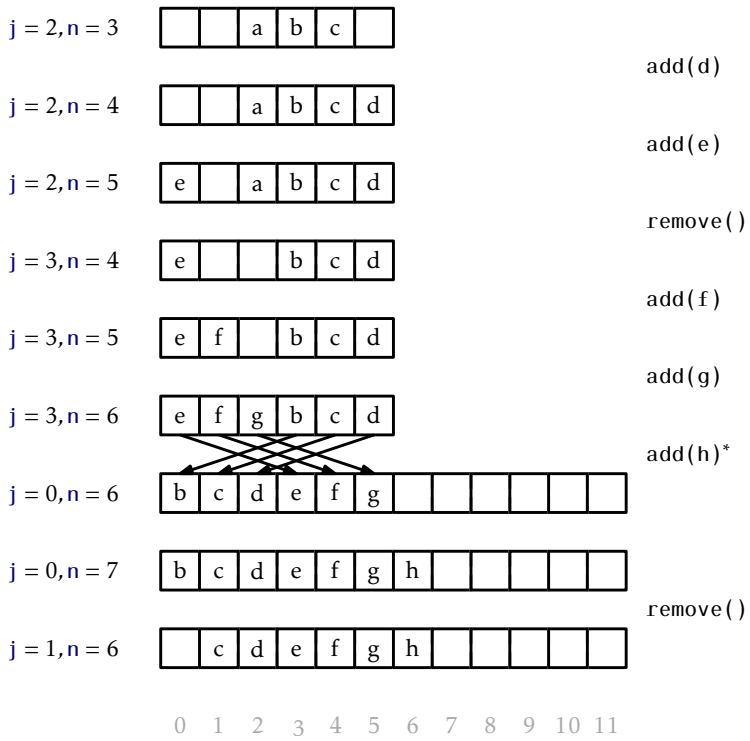
Zaporedje operacij `add(x)` in `remove()` nad `ArrayQueue` je prikazano na 2.2. Za izvedbo `add(x)` moramo najprej preveriti, če je `a` poln, in s klicem `resize()` velikost `a` povečati. Nato `x` shranimo v `a[(j+n)%a.length]` in povečamo `n`.

ArrayQueue

```
bool add(T x) {
   if (n + 1 > a.length) resize();
   a[(j+n) % a.length] = x;
   n++;
   return true;
 }
```

Za izvedbo `remove()` moramo najprej za kasnejšo rabo shraniti `a[j]`. Nato zmanjšamo `n` in povečamo `j` (po modulu `a.length`) tako, da nasta-

[2] Temu včasih rečemo operator *brain-dead*, ker nepravilno implementira matematični operator mod, ko je prvi argument negativno število.



Slika 2.2: Zaporedje operacij `add(x)` in `remove(i)` nad `ArrayQueue`. Puščice označujejo kopiranje elementov. Operacije, ki se zaključijo s klicem `resize()` so označene z zvezdico.

vimo `j` = (`j`+1) mod `a.length`. Na koncu vrnemo shranjeno vrednost `a[j]`. Po potrebi lahko zmanjšamo velikost `a` s klicem `resize()`.

ArrayQueue
```
T remove() {
  T x = a[j];
  j = (j + 1) % a.length;
  n--;
  if (a.length >= 3*n) resize();
  return x;
}
```

Operacija `resize()` je zelo podobna operaciji `resize()` pri `ArrayStack`. Dodeli novo polje `b` velikosti $2n$ in prepiše

$$a[j], a[(j+1)\%a.length], \ldots, a[(j+n-1)\%a.length]$$

na

$$b[0], b[1], \ldots, b[n-1]$$

in nastavi $j = 0$.

ArrayQueue
```
void resize() {
  array<T> b(max(1, 2*n));
  for (int k = 0; k < n; k++)
    b[k] = a[(j+k)%a.length];
  a = b;
    j = 0;
}
```

2.3.1 Povzetek

Naslednji izrek povzema učinkovitost podatkovne strukture `ArrayQueue`:

Izrek 2.2. *`ArrayQueue` implementira vmesnik (FIFO) `Vrste`. Če izvzamemo ceno klica* `resize()`*, omogoča `ArrayQueue` izvajanje operacij* `add(x)` *in* `remove()` *v času $O(1)$ na operacijo. Poleg tega, začenši s prazno vrsto `ArrayQueue`, vsako zaporedje m operacij* `add(i,x)` *in* `remove(i)` *porabi skupno $O(m)$ časa za vse klice* `resize()`*.*

2.4 ArrayDeque: Hitra obojestranska vrsta z uporabo polja

Struktura ArrayQueue iz prejšnjega poglavja je podatkovna struktura za predstavitev zaporedja, ki omogoča učinkovito dodajanje na en konec in odstranjevanje z drugega konca. Podatkovna struktura ArrayDeque pa omogoče tako učinkovito dodajanje kot tudi odstranjevanje z obeh koncev. Ta struktura implementira vmesnik List z uporabo enake tehnike krožnega polja, ki je uporabljena pri ArrayQueue.

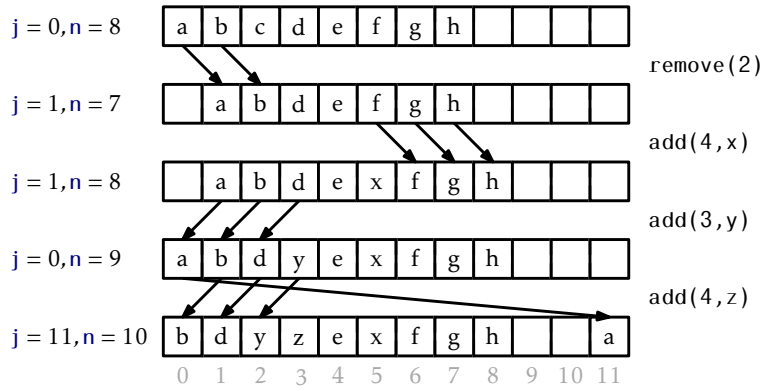
ArrayDeque
```
array<T> a;
int j;
int n;
```

Operaciji get(i) in set(i,x) nad ArrayDeque sta enostavni. Vrneta oziroma nastavita element polja a[(j + i) mod a.length].

ArrayDeque
```
T get(int i) {
  return a[(j + i) % a.length];
}
T set(int i, T x) {
  T y = a[(j + i) % a.length];
  a[(j + i) % a.length] = x;
  return y;
}
```

Implementacija operacije add(i,x) je bolj zanimiva. Kot ponavadi, najprej preverimo, če je a poln in ga po potrebi povečamo s klicem resize(). Želimo, da je ta operacija hitra tako, ko je i majhen (blizu 0), kot tudi, ko je i velik (blizu n). Zato preverimo, če drži $i < n/2$. Če drži, zamaknemo elemente $a[0],\ldots,a[i-1]$ za eno mesto v levo. Sicer ($i \geq n/2$), elemente $a[i],\ldots,a[n-1]$ zamaknemo za eno mesto v desno. 2.3 prikazuje operaciji add(i,x) in remove(x) nad ArrayDeque.

ArrayDeque
```
void add(int i, T x) {
  if (n + 1 > a.length)  resize();
  if (i < n/2) { // shift a[0],..,a[i-1] left one position
```



Slika 2.3: Zaporedje operacij add(i,x) in remove(i) nad ArrayDeque. Puščice označujejo prestavljanje elementov.

```
    j = (j == 0) ? a.length - 1 : j - 1;
    for (int k = 0; k <= i-1; k++)
      a[(j+k)%a.length] = a[(j+k+1)%a.length];
  } else { // shift a[i],..,a[n-1] right one position
    for (int k = n; k > i; k--)
      a[(j+k)%a.length] = a[(j+k-1)%a.length];
  }
  a[(j+i)%a.length] = x;
  n++;
}
```

S prestavljanjem elementov na tak način zagotovimo, da add(i,x) nikoli ne potrebuje prestaviti več not $\min\{i, n-i\}$ elementov. Čas izvajanja operacije add(i,x), (če ignoriramo ceno operacije resize()), je potemtakem $O(1+\min\{i, n-i\})$.

Operacija remove(i) je izvedena podobno. Odvisno od $i < n/2$, remove(i) bodisi zamakne elemente $a[0],\ldots,a[i-1]$ za eno mesto v desno, bodisi elemente $a[i+1],\ldots,a[n-1]$ zamakne za eno mesto v levo. To spet pomeni, da remove(i) za zamik elementov nikoli ne potrebuje več kot $O(1+\min\{i, n-i\})$ časa.

ArrayDeque

```
T remove(int i) {
```

```
    T x = a[(j+i)%a.length];
    if (i < n/2) { // shift a[0],..,[i-1] right one position
      for (int k = i; k > 0; k--)
      a[(j+k)%a.length] = a[(j+k-1)%a.length];
    j = (j + 1) % a.length;
    } else { // shift a[i+1],..,a[n-1] left one position
    for (int k = i; k < n-1; k++)
      a[(j+k)%a.length] = a[(j+k+1)%a.length];
    }
    n--;
    if (3*n < a.length) resize();
    return x;
}
```

2.4.1 Povzetek

Naslednji izrek povzema učinkovitost podatkovne strukture ArrayDeque:

Izrek 2.3. *ArrayDeque implementira vmesnik* `List`*. Če izvzamemo ceno klica* `resize()`*, omogoča ArrayDeque izvajanje operacij*

- `get(i)` *in* `set(i,x)` *v času* $O(1)$ *na operacijo; in*
- `add(i,x)` *in* `remove(i)` *v času* $O(1+\min\{i, n-i\})$ *na operacijo.*

Poleg tega, začenši s prazno obojestransko vrsto ArrayDeque, vsako zaporedje m *operacij* `add(i,x)` *in* `remove(i)` *porabi skupno* $O(m)$ *časa za vse klice* `resize()`*.*

2.5 DualArrayDeque: Gradnja obojestranske vrste z dveh skladov

V sledečem poglavju bomo predstavili podatkovno strukturo DualArrayDeque, ki za dosego enakih meja učinkovitosti kot ArrayDeque, uporablja dve skladovni polji (ArrayStack). Čeprav ni asimptotična učinkovitost DualArrayDeque nič boljša kot pri ArrayDeque, je struktura vseeno zanimiva, ker nudi dober primer napredne strukture z združitvijo dveh enostavnih.

`DualArrayDeque` predstavlja seznam z uporabo dveh `ArrayStackov`. Spomnimo se, da `ArrayStack` deluje hitro, ko operacije nad njim spreminjajo elementa z njegovega konca. `DualArrayDeque` sestoji iz dveh `ArrayStackov`, enega `spredaj` (`front`) in enega `zadaj` (`back`), s konci nasproti, da to operacije hitre na obeh straneh.

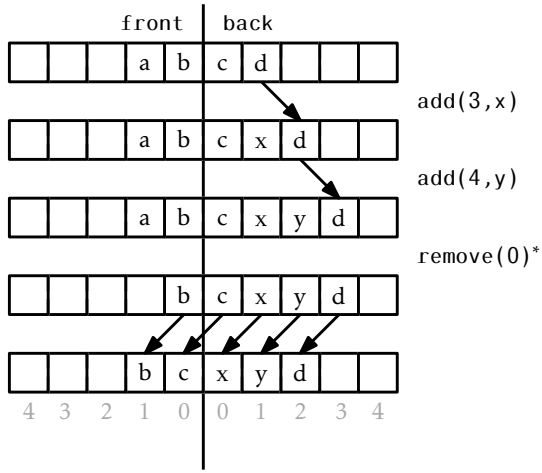
DualArrayDeque
```
ArrayStack<T> front;
ArrayStack<T> back;
```

`DualArrayDeque` ne hrani eksplicitno števila elementov, `n`, ki jih vsebuje. Števila ni potrebno hraniti, saj vsebuje `n` = `front.size()+back.size()` elementov. Vseeno pa bomo pri analizi `DualArrayDeque` uporabljali `n` za označevanje števila vsebovanih elementov.

DualArrayDeque
```
int size() {
  return front.size() + back.size();
}
```

`Sprednji ArrayStack` hrani seznam elementov z indeksi $0,\ldots,$`front.size()`-1, vendar jih hrani v obratnem vrstnem vredu. `Zadnji ArrayStack` pa hrani seznam elementov z indeksi `front.size()`$,\ldots,$`size()`-1 v običajnem vrstnem redu. Na tak način se `get(i)` in `set(i,x)` prevedeta v primerne klice `get(i)` ali `set(i)` na bodisi `sprednjem` ali `zadnjem` koncu, kar potrebuje $O(1)$ časa na operacijo.

DualArrayDeque
```
T get(int i) {
  if (i < front.size()) {
    return front.get(front.size() - i - 1);
  } else {
    return back.get(i - front.size());
  }
}
T set(int i, T x) {
  if (i < front.size()) {
    return front.set(front.size() - i - 1, x);
  } else {
```



Slika 2.4: Zaporedje operacij `add(i,x)` in `remove(i)` nad `DualArrayDeque`. Puščice označujejo prestavljanje elementov. Operacije, po katerih se seznam uravnoteži s klicem `balance()`, so označene z zvezdico.

```
    return back.set(i - front.size(), x);
  }
}
```

Če je indeks `i` < `front.size()`, potem opazimo da ustreza elementu `spredaj` na položaju `front.size()` − `i` − 1, ker so elementi `spredaj` shranjeni v obratnem vrstnem redu.

Dodajanje in odstranjevanje elementov iz `DualArrayDeque` je prikazano na sliki 2.4. Operacija `add(i,x)` doda element `spredaj` ali `zadaj`, odvisno od stanja:

DualArrayDeque
```
void add(int i, T x) {
  if (i < front.size()) {
    front.add(front.size() - i, x);
  } else {
    back.add(i - front.size(), x);
  }
  balance();
```

```
}
```

Metoda add(i,x) uravnoteži sprednji in zadnji ArrayStack s klicom metode balance(). Izvedba balance() je prikazana spodaj, za enkrat pa je dovolj, če vemo, da razen če je size() < 2, balance() poskrbi za to, da se front.size() in back.size() ne razlikujeta več kot za faktor 3. Natančneje, $3 \cdot \texttt{front.size}() \geq \texttt{back.size}()$ in $3 \cdot \texttt{back.size}() \geq \texttt{front.size}()$.

Nato, analiziramo ceno metode add(i,x), pri tem ne upoštevamo ceno klicev metode balance(). Če $\texttt{i} < \texttt{front.size}()$, potem se add(i,x) izvede s klicem na front.add(front.size() − i − 1,x). Ker je front ArrayStack je cena tega

$$O(\texttt{front.size}() - (\texttt{front.size}() - \texttt{i} - 1) + 1) = O(\texttt{i} + 1) \ . \tag{2.1}$$

Po drugi strani pa, če drži $\texttt{i} \geq \texttt{front.size}()$, potem je add(i,x) implementirana kot back.add(i − front.size(),x). Cena tega pa je

$$O(\texttt{back.size}() - (\texttt{i} - \texttt{front.size}()) + 1) = O(\texttt{n} - \texttt{i} + 1) \ . \tag{2.2}$$

Opazimo, da se prvi primer (2.1) pojavi, ko velja $\texttt{i} < \texttt{n}/4$. Drugi primer (2.2) se pojavi, ko velja $\texttt{i} \geq 3\texttt{n}/4$. Kadar velja $\texttt{n}/4 \leq \texttt{i} < 3\texttt{n}/4$, ne moremo biti prepričani ali delovanje vpliva na front ali back, ampak v vsakem primeru se postopek izvaja $O(\texttt{n}) = O(\texttt{i}) = O(\texttt{n} - \texttt{i})$ časa, saj je $\texttt{i} \geq \texttt{n}/4$ in $\texttt{n} - \texttt{i} > \texttt{n}/4$. Če povzamemo situacijo imamo

$$\text{Čas izvajanja add}(\texttt{i},\texttt{x}) \leq \begin{cases} O(1+\texttt{i}) & \text{if } \texttt{i} < \texttt{n}/4 \\ O(\texttt{n}) & \text{if } \texttt{n}/4 \leq \texttt{i} < 3\texttt{n}/4 \\ O(1+\texttt{n}-\texttt{i}) & \text{if } \texttt{i} \geq 3\texttt{n}/4 \end{cases}$$

Tako je čas izvajanja add(i,x), če zanemarimo ceno klicev metode balance() sledeč $O(1 + \min\{\texttt{i}, \texttt{n} - \texttt{i}\})$.

Metoda remove(i) in njene analize spominjajo na add(i,x) metodo.

DualArrayDeque

```
T remove(int i) {
    T x;
    if (i < front.size()) {
            x = front.remove(front.size()-i-1);
    } else {
```

```
            x = back.remove(i-front.size());
    }
    balance();
    return x;
}
```

2.5.1 Uravnoteženje

Osredotočimo se na metodo `balance()` izvedeno z metodo `add(i,x)` in `remove(i)`. Ta postopek zagotavlja, da niti `front` in niti `back` ne postaneta prevelika (ali premajhna). Zagotavlja, da razen, če obstajata manj kot dva elementa, tako `front` in `back` vsebujeta vsaj $n/4$ elementov. Če temu ni tako, potem se premika elemente med njima tako, da `front` in `back` vsebujeta natanko $\lfloor n/2 \rfloor$ elementov in $\lceil n/2 \rceil$ elementov.

DualArrayDeque

```
void balance() {
  if (3*front.size() < back.size()
      || 3*back.size() < front.size()) {
    int n = front.size() + back.size();
    int nf = n/2;
    array<T> af(max(2*nf, 1));
    for (int i = 0; i < nf; i++) {
      af[nf-i-1] = get(i);
    }
    int nb = n - nf;
    array<T> ab(max(2*nb, 1));
    for (int i = 0; i < nb; i++) {
      ab[i] = get(nf+i);
    }
    front.a = af;
    front.n = nf;
    back.a = ab;
    back.n = nb;
  }
}
```

Če metoda `balance()` izvede uravnoteženje, potem premakne $O(n)$ elementov in za to potrebuje $O(n)$ časa. To je slabo zato, ker je metoda

balance() klicana z vsakim add(i,x) in remove(i) klicem. V vsakem primeruu, sledeč dokaz dokazuje, da metoda balance() v povprečju porabi samo konstantno količino časa na operacijo.

Lema 2.2. *Če ustvarimo prazen DualArrayDeque, potem zaporedje $m \geq 1$ izvede klice metode* add(i,x) *in* remove(i)*, potem je skupen porabljen čas za klice metode* balance() $O(m)$.

Dokaz. Dokazali bomo, da če metoda balance() premeša elemente, potem je število add(i,x) in remove(i) operacij vsaj n/2−1, od kar so bili elementi nazadnje premešani z metodo balance(). Z dokazom v 2.1 lahko dokažemo, da je skupen porabljen čas metode balance() $O(m)$.

Izvedli bomo našo analizo z uporabo tehnike, poznane kot *potencialna metoda*. Določimo *potencialni* Φ za DualArrayDeque kot razliko v dolžini med front in back:

$$\Phi = |\texttt{front.size()} - \texttt{back.size()}| \ .$$

Zanimiva stvar glede potenciala je, da klic metode add(i,x) ali remove(i), ki ne opravi nobenega uravnoteženja, lahko poveča potencial skoraj največ za 1.

Potrebno je upoštevati, da je takoj po klicu metode balance(), ki premeša elemente, potencial Φ_0 največ 1, saj

$$\Phi_0 = |\lfloor n/2 \rfloor - \lceil n/2 \rceil| \leq 1 \ .$$

Razmislite o trenutku takoj pred klicem funkcije balance(), ki premeša elemente in domnevajte, da balance() premeša elemente zaradi $3\texttt{front.size()} < \texttt{back.size()}$. To opazimo v sledečem primeru,

$$\begin{aligned} \texttt{n} &= \texttt{front.size()} + \texttt{back.size()} \\ &< \texttt{back.size()}/3 + \texttt{back.size()} \\ &= \frac{4}{3}\texttt{back.size()} \end{aligned}$$

Poleg tega je s časom potencial na tem mestu

$$\begin{aligned}
\Phi_1 &= \texttt{back.size()} - \texttt{front.size()} \\
&> \texttt{back.size()} - \texttt{back.size()}/3 \\
&= \frac{2}{3}\texttt{back.size()} \\
&> \frac{2}{3} \times \frac{3}{4}\texttt{n} \\
&= \texttt{n}/2
\end{aligned}$$

Zato je število klicev metode `add(i,x)` ali `remove(i)`, od kar je metoda `balance()` nazadnje premešala elemente, najmanj $\Phi_1 - \Phi_0 > \texttt{n}/2 - 1$. To zaključuje dokaz. □

2.5.2 Povzetek

Naslednji izrek povzame lastnosti `DualArrayDeque`:

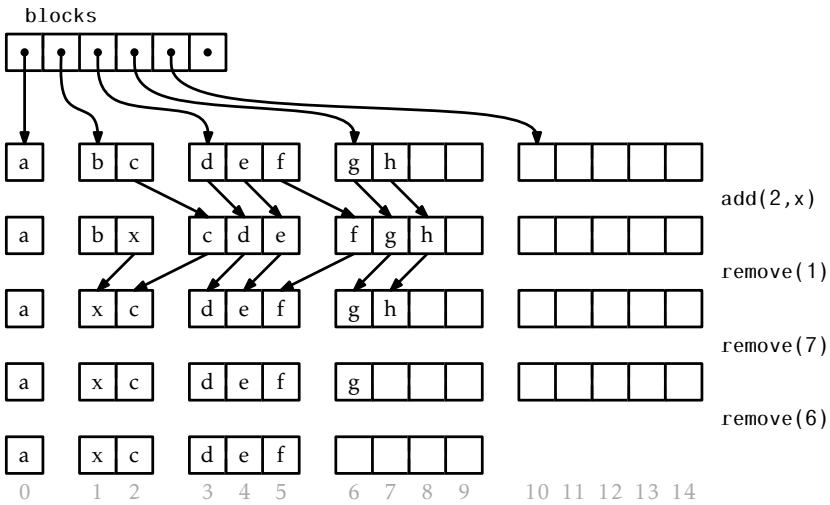
Izrek 2.4. *`DualArrayDeque` implementira vmesnik `List`. Z ignoriranjem cene klicev metod `resize()` in `balance()` `DualArrayDeque` podpira operacije*

- `get(i)` *in* `set(i,x)` *v času* $O(1)$ *na operacijo; in*
- `add(i,x)` *in* `remove(i)` *v času* $O(1 + \min\{\texttt{i}, \texttt{n} - \texttt{i}\})$ *na operacijo.*

Poleg tega, če začnemo z praznim `DualArrayDeque`, potem zaporedje m `add(i,x)` *in* `remove(i)` *metod, konča z skupnim rezultatom* $O(m)$ *časa porabljenega med vsemi klici metod `resize()` in `balance()`.*

2.6 `RootishArrayStack`: Prostorsko učinkovit `ArrayStack`

Ena izmed slabosti vseh prejšnjih podatkovnih struktur v tem poglavju je ta, da ker se shranjujejo podatki v eni ali dveh tabelah, ki se izogibajo spreminjanju velikosti, se pogosto zgodi, da so tabele precej prazne. Na primer, takoj po operaciji `resize()` nad `ArrayStack`-om, je tabela `a` le na pol polna. Še huje, veliko je primerov, kjer samo 1/3 tabele `a` vsebuje podatke.



Slika 2.5: Sekvenca `add(i,x)` in `remove(i)` operacij na `RootishArrayStack`. Puščice označujejo kopirane elemente.

Ta razdelek je namenjen podatkovni strukturi `RootishArrayStack`, ki se posveča problemu zapravljenega prostora. `RootishArrayStack` vsebuje n elementov z uporabo $O(\sqrt{n})$ tabel. V teh tabelah je največ $O(\sqrt{n})$ lokacij neuporabljenih v poljubnem času. Vse preostale lokacije v tabeli so uporabljene za shrambo podatkov. Potemtakem te podatkovne strukture zapravijo največ $O(\sqrt{n})$ prostora pri shranjevanju n elementov.

`RootishArrayStack` shrani svoje elemente v seznam r tabel poimenovanih *blocks*, ki so oštevilčene $0, 1, \ldots, r-1$. Glej 2.5. Blok b vsebuje $b+1$ elemente, zato vsi r bloki vsebujejo največ

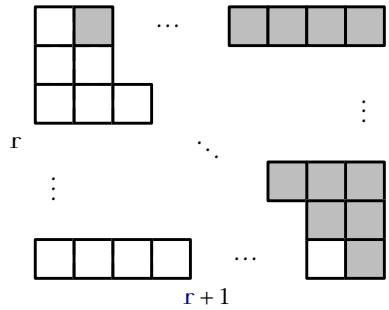
$$1 + 2 + 3 + \cdots + r = r(r+1)/2$$

elementov. Zgornja formula se izpelje kot je prikazano na 2.6.

RootishArrayStack

```
ArrayStack<T*> blocks;
int n;
```

Kot lahko pričakujemo, so elementi v seznamu razvrščeni po vrsti v bloku. Element v seznamu z indeksom 0 je shranjen v blok 0, elementa



Slika 2.6: Število belih kvadratov je $1+2+3+\cdots+r$. Število osenčenih kvadratov je isto. Beli in osenčeni kvadrati skupaj tvorijo pravokotnik, ki vsebuje $r(r+1)$ kvadratov.

z indeksoma 1 in 2 sta shranjena v blok 1, elementi z indeksi 3, 4 in 5 so shranjeni v blok 2, itn. Glavni problem ki ga je potrebno nasloviti, je pri odločanju, ko nam je podan indeks i, kateri blok vsebuje tako i, kot tudi ustrezni indeks do i v samem bloku.

Določanje indeksa i v njegovem bloku se izkaže kot lahko. Če je indeks i v bloku b, potem je število elementov v blokih $0,\ldots,b-1$ $b(b+1)/2$. Potemtakem je i shranjen na lokaciji

$$j = i - b(b+1)/2$$

v bloku b. Malo bolj zahteven je problem določanja vrednosti bloku b. Število elementov, ki ima indekse manj ali enake i je $i+1$. Na drugi strani pa je število elementov v blokih $0,\ldots,b$, ki je enako $(b+1)(b+2)/2$. Potemtakem je b najmanjše število, ki še ustreza

$$(b+1)(b+2)/2 \geq i+1 \ .$$

To enačbo lahko preoblikujemo tako

$$b^2 + 3b - 2i \geq 0 \ .$$

Ustrezno kvadratna enačba $b^2+3b-2i=0$ ima dve rešitvi: $b = (-3+\sqrt{9+8i})/2$ in $b = (-3-\sqrt{9+8i})/2$.

Druga rešitev nima smisla za našo uporabo, ker da vedno negativno rešitev. Zato uporabimo $b = (-3+\sqrt{9+8i})/2$. V splošnem ta rešitev ni

število, vendar če se vrnemo k naši neenakosti, hočemo najmanjšo število b, tako, da velja $b \geq (-3+\sqrt{9+8i})/2$. To je preprosto

$$b = \left\lceil (-3+\sqrt{9+8i})/2 \right\rceil \ .$$

RootishArrayStack
```
int i2b(int i) {
    double db = (-3.0 + sqrt(9 + 8*i)) / 2.0;
    int b = (int)ceil(db);
    return b;
}
```

Ko je to jasno, sta tudi metodi `get(i)` in `set(i,x)` jasni. Najprej izračunamo ustrezen blok `b` in ustrezen indeks `j` v bloku. Potem izvedemo primerno operacijo:

RootishArrayStack
```
T get(int i) {
    int b = i2b(i);
    int j = i - b*(b+1)/2;
    return blocks.get(b)[j];
}
T set(int i, T x) {
    int b = i2b(i);
    int j = i - b*(b+1)/2;
    T y = blocks.get(b)[j];
    blocks.get(b)[j] = x;
    return y;
}
```

V primeru, da uporabimo katerokoli podatkovno strukturo v tem poglavju za zastopanje `blocks` seznam, potem se `get(i)` in `set(i,x)` izvajata v konstantnem času.

Metoda `add(i,x)` nam je že poznana. Najprej preverimo, če je naša podatkovna strukura polna tako, da je število blokov `r` tako, da drži $r(r+1)/2 = n$. Če je, pokličemo `grow()`, ki nam doda še en blok. Ko to naredimo, zamaknemo elemente z indeksi $i,\ldots,n-1$ v desno za eno pozicijo, da naredimo prostor za nov element z indeksom `i`:

RootishArrayStack
```
void add(int i, T x) {
    int r = blocks.size();
    if (r*(r+1)/2 < n + 1) grow();
    n++;
    for (int j = n-1; j > i; j--)
            set(j, get(j-1));
    set(i, x);
}
```

Metoda grow() naredi pričakovano. Doda nov blok:

RootishArrayStack
```
void grow() {
    blocks.add(blocks.size(), new T[blocks.size()+1]);
}
```

Če ignoriramo ceno operacije grow(), potem je cena add(i,x) dominirana z vrednostjo zamikanja in je potemtakem enaka $O(1+n-i)$, kar je enako, kot pri ArrayStack.

Operacija remove(i) je podobna metodi add(i,x). Le ta zamakne elemente z indeksi $i+1,\ldots,n$ levo za eno pozicijo. Za tem, če je več kot en blok še prazen, pokliče metodo shrink(), da odstrani vse, razen enega še ne uporabljenega bloka:

RootishArrayStack
```
T remove(int i) {
    T x = get(i);
    for (int j = i; j < n-1; j++)
            set(j, get(j+1));
    n--;
    int r = blocks.size();
    if ((r-2)*(r-1)/2 >= n) shrink();
    return x;
}
```

RootishArrayStack
```
void shrink() {
    int r = blocks.size();
```

```
    while (r > 0 && (r-2)*(r-1)/2 >= n) {
            delete [] blocks.remove(blocks.size()-1);
            r--;
    }
}
```

Če spet ignoriramo ceno operacije shrink(), je cena remove(i) dominirana z vrednostjo zamikanja in je potemtakem enaka $O(n-i)$.

2.6.1 Analiza rasti in krčenja

Zgornja analiza add(i,x) in remove(i) ne vzema v zakup cene metodi grow() in shrink(). Upoštevajte, da metodi grow() in shrink() ne kopirata nobenih podatkov, kot to dela operacija ArrayStack.resize(), temveč le alocirajo ali izpraznijo tabelo velikosti r. V določenih okoljih se to zgodi v konstantnem času, dočim zna v drugih to zahtevati proporcionalen čas glede na r.

Takoj po klicu grow() ali shrink() se situacija počisti. Zanji blok je popolnoma prazen, vsi ostali pa so povsem zapolnjeni. Dodaten klic grow() ali shrink() se ne bo zgodil dokler vsaj $r-1$ elementov ni bilo dodanih ali odstranjenih. Četudi vzamejo grow() in shrink() $O(r)$ časa, je lahko vrednost cene grow() in shrink() amortizirana na $O(1)$ za vsako posamezno operacijo.

2.6.2 Poraba prostora

Sedaj bomo analizirali količino dodatnega prostora, ki ga uporablja RootishArrayStack. Bolj natančno, hočemo prešteti ves prostor, ki ga uporablja RootishArrayStack in le ta ni element tabele, ki je trenutno uporabljen za držanje elementa seznama. Takemu prostoru rečemo *wasted space*.

Operacija remove(i) zagotavlja, da RootishArrayStack nikoli nima več kot dva zapolnjena bloka. Število blokov, r, uporabljenih s strani RootishArrayStack, ki imajo shranjenih n elementov potemtakem zadovoljijo

$$(r-2)(r-1) \leq n \ .$$

Če uporabimo kvadratno enačbo nam da

$$r \leq (3 + \sqrt{1 + 4n})/2 = O(\sqrt{n}) \ .$$

Zadnje dva bloka sta velikosti r in $r - 1$, zato je največ zapravljenega prostora $2r - 1 = O(\sqrt{n})$. Če shranimo bloka v (npr.) `ArrayList`, ima potem `List`, ki shranjuje r bloke, $O(r) = O(\sqrt{n})$ zapravljenega prostora. Ostali prostor, ki ga potrebujemo za shrambo n in ostalih informacij je potemtakem $O(1)$. Skupaj je zapravljenega prostora v `RootishArrayStack` $O(\sqrt{n})$.

Nato trdimo, da je tak način uporabe prostora optimalen za katerokoli podatkovno strukturo, ki je na začetku prazna in podpira seštevanje enega elementa v določenem času. Bolj natančno smo zmožni prikazati, da v točno določenem času med seštevanjem n elementov, podatkovna struktura zapravlja vsaj $\sqrt{n}$ prostora (čeprav je to le za trenutek).

Predpostavimo, da začnemo s prazno podatkovno strukturo in dodamo n elementov vsakega posebej. Na koncu procesa je vseh n elementov shranjenih v strukturi in porazdeljenih med r kolekcijo spominskih blokov. Če velja $r \geq \sqrt{n}$, potem mora podatkovna struktura uporabljati r kazalcev (ali referenc), da sledi vsem r blokom. Te kazalci so zapravljen prostor. Na drugi strani če velja $r < \sqrt{n}$, potem morajo zaradi načela predalčkanja, določeni bloki biti vsaj $n/r > \sqrt{n}$ veliki. Vpoštevajoč moment v katerem je bil blok najprej alociran. Takoj po alociranju, je bil blok prazen in je zato zapravljal $\sqrt{n}$ prostora. Zaradi tega je bilo ob točno določenem času med vstavljanjem n elemntov, zapravljenega $\sqrt{n}$ prostora s strani podatkovne strukture.

2.6.3 Povzetek

Sledeč teorem povzema našo diskusijo o podatkovni strukturi `RootishArrayStack`:

Izrek 2.5. *`RootishArrayStack` implementira vmesnik `List`. `RootishArrayStack` ignorira cene klicev metod* `grow()` *in* `shrink()` *ter podpira operacije*

- `get(i)` *in* `set(i,x)` *z* $O(1)$ *časom na operacijo; in*
- `add(i,x)` *in* `remove(i)` *z* $O(1 + n - i)$ *časom na operacijo.*

Še več, če začnemo s praznim `RootishArrayStack`*, bo katerakoli sekvenca* m `add(i,x)` *in* `remove(i)` *operacij potrebovala v celoti* $O(m)$ *časa za vse klice teh dveh metod.*

Prostor (merjen v besedah),[3] *ki ga* `RootishArrayStack` *porabi za shrambo* n *elementov, je* $n + O(\sqrt{n})$.

2.6.4 Računanje Kvadratnih Korenov

Bralec ki je imel nekaj stika z modeli računanja, morda opazi da zgoraj opisan `RootishArrayStack`, ne spada v običajni model računanja besedni-RAM (1.4, ker zahteva računanje kvadratnih korenov. Operacija kvadratnega korena ni smatrana za navadno operacijo in navadno ni del besednega-RAM modela.

V tej sekciji pokažemo, da se lahko implementacijo kvadratnega korena učinkovito implementira. Še posebej pokažemo, da je vsako število $x \in \{0,\ldots,n\}$, $\lfloor\sqrt{x}\rfloor$lahko izračunano v konstantnem-času, nato ko $O(\sqrt{n})$ predpriprava ustvari dve tabeli dolžine $O(\sqrt{n})$. Sledeča lema kaže, da lahko zmanjšamo problem računanja kvadratnega korena spremenljivke x v kvadratni koren sorodne vrednosti x'.

Lema 2.3. *Naj bo* $x \geq 1$ *in* $x' = x - a$*, kjer je* $0 \leq a \leq \sqrt{x}$*. Potem sledi da* $\sqrt{x'} \geq \sqrt{x} - 1$.

Dokaz. Zadostuje pokazati da

$$\sqrt{x - \sqrt{x}} \geq \sqrt{x} - 1 \ .$$

Kvadriramo obe strani te neenačbe, da dobimo

$$x - \sqrt{x} \geq x - 2\sqrt{x} + 1$$

nato poračunamo do konca, da dobimo

$$\sqrt{x} \geq 1$$

kar drži za vsak $x \geq 1$. □

[3] Spomnimo se 1.4 za diskusijo kako se meri spomin.

Začnemo tako, da malo omejimo problem in predpostavimo da je $2^r \le x < 2^{r+1}$, tako da $\lfloor \log x \rfloor = r$, t.j., x je število z $r+1$ biti v binarni predstavitvi števil. Uzamemo $x' = x - (x \bmod 2^{\lfloor r/2 \rfloor})$. Sedaj, x' zadošča pogojem 2.3, zato je $\sqrt{x} - \sqrt{x'} \le 1$. Poleg tega ima x' vse spodnje $\lfloor r/2 \rfloor$ bite enake 0, zato obstaja samo ena

$$2^{r+1-\lfloor r/2 \rfloor} \le 4 \cdot 2^{r/2} \le 4\sqrt{x}$$

od možnih vrednosti x'. To pomeni da lahko uporabimo tabelo, `sqrttab`, ki shrani vrednost od $\lfloor \sqrt{x'} \rfloor$ za vsako možno vrednost spremenljivke x'. Bolj natančno, imamo

$$\texttt{sqrttab}[i] = \left\lfloor \sqrt{i 2^{\lfloor r/2 \rfloor}} \right\rfloor \ .$$

Na ta način je `sqrttab`$[i]$ znotraj dveh $\sqrt{x}$ za vsak $x \in \{i2^{\lfloor r/2 \rfloor}, \ldots, (i+1)2^{\lfloor r/2 \rfloor} - 1\}$. Drugače povedano, vhodni niz `s` = `sqrttab[x>>⌊r/2⌋]` je bodisi enak $\lfloor \sqrt{x} \rfloor$, $\lfloor \sqrt{x} \rfloor - 1$, ali $\lfloor \sqrt{x} \rfloor - 2$. S spremenljivko `s` lahko določimo vrednost $\lfloor \sqrt{x} \rfloor$ s povečevanjem `s` dokler $(s+1)^2 > x$.

FastSqrt

```
int sqrt(int x, int r) {
  int s = sqrtab[x>>r/2];
  while ((s+1)*(s+1) <= x) s++; // executes at most twice
  return s;
}
```

Vendarle to deluje samo pri $x \in \{2^r, \ldots, 2^{r+1} - 1\}$ in `sqrttab` je posebna tabela, ki deluje samo za določeno vrednost $r = \lfloor \log x \rfloor$. Da to rešimo, lahko izračunamo $\lfloor \log n \rfloor$ drugačnih `sqrttab` tabel, eno za vsako možno vrednost odf $\lfloor \log x \rfloor$. Velikosti teh tabel oblikujejo eksponentno zaporedje, katerega največja vrednost je kvečjemu $4\sqrt{n}$, tako da skupna velikost vseh tabel je $O(\sqrt{n})$.

Kakorkoli, izkaže se, da ne potrebujemo več kot ene `sqrttab` tabele; potrebujemo samo eno `sqrttab` tabelo za vrednost $r = \lfloor \log n \rfloor$. Vsaka vrednost x z $\log x = r' < r$ je lahko *upgraded* z množenjem x z $2^{r-r'}$ in uporabo enačbe

$$\sqrt{2^{r-r'}x} = 2^{(r-r')/2}\sqrt{x} \ .$$

Količina $2^{r-r'}x$ je v obsegu $\{2^r, \ldots, 2^{r+1} - 1\}$ zato lahko pogledamo njen kvadratni koren v `sqrttab`. Sledeča koda dopolni to idejo za izračun $\lfloor \sqrt{x} \rfloor$

za vsa ne-negativna števila x v obsegu $\{0,\ldots,2^{30}-1\}$ z uporabo tabele, sqrttab, velikosti 2^{16}.

FastSqrt
```
int sqrt(int x) {
  int rp = log(x);
  int upgrade = ((r-rp)/2) * 2;
  int xp = x << upgrade;  // xp has r or r-1 bits
  int s = sqrtab[xp>>(r/2)] >> (upgrade/2);
  while ((s+1)*(s+1) <= x) s++;  // executes at most twice
  return s;
}
```

Nekaj kar smo si vzeli za samoumnevno je vprašanje kako izračunati $r' = \lfloor \log x \rfloor$.Spet, to je problem ki ga lahko rešimo z tabelo, logtab, velikosti $2^{r/2}$. V tem primeru je koda še posebej enostavna, ker je $\lfloor \log x \rfloor$ samo kazalo pomembnega 1 bita v binarni predstavitni x. To pomeni, da za $x > 2^{r/2}$, lahko premaknemo v desno bite od x za r/2 pozicij, preden ga uporabimo za kazalo v logtab. Sledeča koda naredi to, z uporabo tabele logtab velikosti 2^{16} za izračun $\lfloor \log x \rfloor$ za vse x v obsegu $\{1,\ldots,2^{32}-1\}$.

FastSqrt
```
int log(int x) {
  if (x >= halfint)
    return 16 + logtab[x>>16];
  return logtab[x];
}
```

Nazadnje, z namenom dopolnitve vključimo sledečo kodo ki inicializira logtab in sqrttab:

FastSqrt
```
void inittabs() {
  sqrtab = new int[1<<(r/2)];
  logtab = new int[1<<(r/2)];
  for (int d = 0; d < r/2; d++)
    for (int k = 0; k < 1<<d; k++)
      logtab[1<<d+k] = d;
  int s = 1<<(r/4);                    // sqrt(2^(r/2))
  for (int i = 0; i < 1<<(r/2); i++) {
    if ((s+1)*(s+1) <= i << (r/2)) s++; // sqrt increases
    sqrtab[i] = s;
```

```
    }
}
```

Za povzetek, izračuni ki so nastali od `i2b(i)` metode lahko implementiramo v konstantnem času na besednem-RAM z uporabo $O(\sqrt{n})$ dodatnega spomina za shranjevanje `sqrttab` in `logtab` arrays. Te tabele lahko obnovimo ko `n`narase ali se zmanjša za faktor ali dva in strošek te obnovitve je lahko amortiziran čez števila od `add(i,x)` and `remove(i)` operaciji, ki sta povzročili spremembo v `n` enako kot je strošek `resize()` analiziran v `ArrayStack` implementaciji.

2.7 Razprava in vaje

Večina podatkovnih struktur, opisanih v tem poglavju je del folklore. Njihove implementacije so stare tudi več kot 30 let. Na primer, implementacije skladov, vrst in dvojnih vrst so lahko generalizirajo v `ArrayStack`, `ArrayQueue` and `ArrayDeque` opisani v tej knjigi, razloži Knuth [**?**, Section 2.2.2]

Brodnik *et al.* [**?**] je prvi opisal `RootishArrayStack` in dokazal $\sqrt{n}$ spodnjo omejenost, kot v 2.6.2. Prikazujejo tudi drugačne strukture, ki uporabljajo bolj sofisticirano izbiro velikosti bloka, zato da se izognejo računanju kvadratnih korenov v `i2b(i)` metodi. Znotraj njihove sheme, blok, ki vsebuje `i` je blok $\lfloor \log(i+1) \rfloor$, ki je indeks vodečega 1 bita v binarni representaciji $i+1$. Nekatere računalniške arhitekture imajo ukaz, ki izračuna indeks vodečega bita v integer-ju.

Struktura sorodna `RootishArrayStack` je dvo-nivojski *tiered-vector* of Goodrich and Kloss [**?**]. Ta struktura omogoča `get(i,x)` in `set(i,x)` operacije v konstantnem času, operacije `add(i,x)` in `set(i,x)` pa v $O(\sqrt{n})$ času. Ti časi so podobni časom, ki jih zmore bolj previdna implementacija `RootishArrayStack` opisana v 2.10.

Naloga 2.1. Metoda `addAll(i,c)` v `List` vstavi vse elemente iz `Collection` `c` v seznam na pozicijo `i`. (Metoda `add(i,x)` je poseben primer, kjer je `c` = {`x`}.) Razložite zakaj je za podatkovne strukture, opisane v tem poglavju, neučinkovito implementirati `addAll(i,c)` z zaporednimi klici `add(i,x)`. Razvijte bolj učinkovito implementacijo.

Naloga 2.2. Razvijte *RandomQueue*. To je implementacija Queue vmesnika v kateri operacija remove() odstrani naključen element izmed vseh elementov, ki so trenutno v vrsti (Razmislite o RandomQueue kot o torbi, v katero lahko dodajamo elemente ali odstranimo nnaključen element.). Operacije add(x) in remove() naj se v RandomQueue izvajajo v konstantnem času.

Naloga 2.3. Razvijte Treque (trojna vrsta). To je implementacija vmesnika List, v katerem se operacije get(i) and set(i,x) izvajajo v konstantnem času, operacije add(i,x) in remove(i) pa v času

$$O(1+\min\{i, n-i, |n/2-i|\}) \ .$$

Z drugimi besedami, modifikacije so hitre, če so blizu kateremukoli koncu ali če so blizu sredine seznama.

Naloga 2.4. Implementirajte metodo rotate(a,r) tako da, "rotira" polje a, tako da je a[i] premik v a[(i+r) mod a.length], za vsak i ∈ {0,...,a.length}.

Naloga 2.5. Implementirajte metodo rotate(r) tako da "rotira" seznam List, tako da element i postane element seznama na (i+r) mod n. Če se izvaja na ArrayDeque ali DualArrayDeque, potem naj se metoda rotate(r) izvaja v času $O(1+\min\{r, n-r\})$.

Naloga 2.6. Popravite implementacijo ArrayDeque tako, da bo se premikanje, ki ga sprožijo operacije add(i,x), remove(i), and resize(), izvajalo hitreje kot System.arraycopy(s,i,d,j,n) metoda.

Naloga 2.7. Popravite implementacijo ArrayDeque tako, da ne uporablja % operatorja (na nekaterih sistemih počasna operacija). Namesto tega naj se posluži dejstva, če je a.length potenca 2, potem

$$k\%a.length = k\&(a.length-1) \ .$$

(Operator & se tu smatra kot bitni.)

Naloga 2.8. Razvijte varijanto ArrayDeque, ki ne izvaja nobene modularne aritmetike. Namesto tega so vsi podatki v zaporednem bloku, urejeno znotraj polja. Ko podatki preplavijo začetek ali konec tega polja, se sproži prirejena operacija rebuild(). Amortizirana cena vseh operacij mora biti enaka kot ArrayDeque.

Namig: Ustreznost delovanja je tu povsem odvisna od tega, kako je implementirana operacija `rebuild()`. Želja je, da operacija `rebuild()` postavi podatkovno strukturo v stanje, kjer podatki ne morejo zbežati, proti kateremukoli koncu, dokler se ne izvede vsaj $n/2$ operacij.

Testirajte vašo implementacijo z primerjanjem performans z `ArrayDeque`. Optimizirajte vašo implemetacijo (z uporabo `System.arraycopy(a,i,b,i,n)`) in preverite če lahko deluje bolje kot implementacija `ArrayDeque` .

Naloga 2.9. Razvijte verzijo `RootishArrayStack`, ki ima samo $O(\sqrt{n})$ porabljenega prostora in lahko izvaja operacije `add(i,x)`, `remove(i,x)` v $O(1+\min\{i, n-i\})$ času.

Naloga 2.10. Razvijte verzijo `RootishArrayStack`, ki ima samo $O(\sqrt{n})$ porabljenega prostora in lahko izvaja operacije `add(i,x)`, `remove(i,x)` v $O(1+\min\{\sqrt{n}, n-i\})$ času. (Namig, glej 3.3.)

Naloga 2.11. Razvijte verzijo `RootishArrayStack`, ki ima samo $O(\sqrt{n})$ porabljenega prostora in lahko izvaja operacije `add(i,x)`, `remove(i,x)` v $O(1+\min\{i, \sqrt{n}, n-i\})$ času. (Namig, glej 3.3.)

Naloga 2.12. Razvijte `CubishArrayStack`. To je tri nivojska struktura, ki implementira `List` vmesnik, in porabi $O(n^{2/3})$ prostora. V tej strukturi se operacije `get(i)` in `set(i,x)` izvajajo v konstantnem času; medtem ko se operacije `add(i,x)` in `remove(i)` izvajajo $O(n^{1/3})$ amortizirano.

Poglavje 3

Povezani seznam

V tem poglavju nadaljujemo z implementacijo seznama `List`, s to razliko, da bomo uporabli podatkovne strukture, ki delujejo na osnovi kazalcev namesto polj. Strukture v tem poglavju so sestavljene iz vozlišč, ki vsebujejo elemente seznama. Z uporabo referenc (kazalcev) so vozlišča zaporedno povezana med seboj. Najprej bomo pogledali enostransko povezane sezname, s katerimi lahko implementiramo operacije `Sklada` in (FIFO) `Vrste`, ki se izvedejo v konstantnem času. Nato si bomo pogledali še obojestransko povezani seznam, s katerim lahko implementiramo `Deque` operacije tako, da se izvedejo v konstantnem času (Deque - vrsta pri kateri lahko dodajamo ter odstranjujemo elemente na začetku ali na koncu).

Povezani seznami imajo prednosti in slabosti v primerjavi z implementacijo seznama `List` z uporabo polja. Največja slabost je ta, da izgubimo zmožnost, da lahko v konstantem času dostopamo do kateregakoli elementa z uporabo metod `get(i)` ali `set(i,x)`. Namesto tega, se moramo sprehoditi po celotenem seznamu, element za elementom, dokler ne pridemo do `i`-tega elementa. Največja prednost pa je dinamičnost: z uporabo referenc vsakega vozlišča seznama `u`, lahko izbrišemo `u` ali vstavimo sosednje vozlišče vozlišču `u` v konstantnem času. To je vedno res ne glede na to, kje se nahaja vozlišče `u` v seznamu.

3.1 SLList: Enostransko povezani seznam

Enostransko povezani seznam SLList (singly-linked list) je zaporedje vozlišč Nodes. Vsako vozlišče u hrani vrednost u.x ter referenco u.next na naslednje vozlišče. Zadnje vozlišče w ima w.next = null

SLList
```
class Node {
public:
  T x;
  Node *next;
  Node(T x0) {
    x = 0;
    next = NULL;
  }
};
```

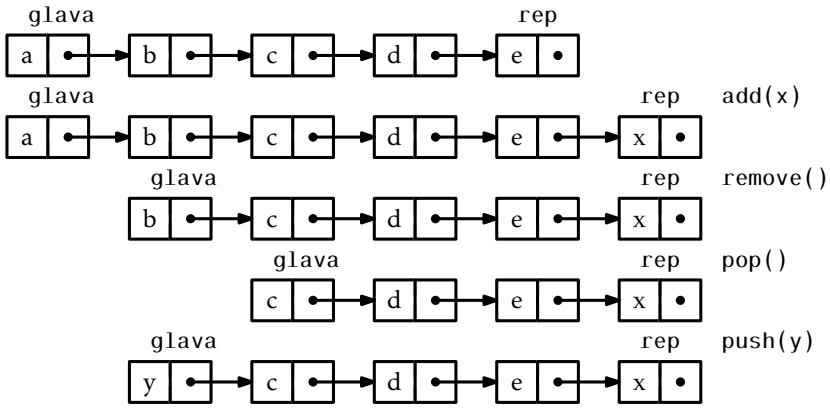
Za boljšo učinkovitost delovanja SLList uporablja spremnljivki head (glava) in tail (rep) za beleženje prvega ter zadnjega vozlišča. Za beleženje dolžine seznama, pa hrani še celoštevilsko spremenljivko n:

SLList
```
Node *head;
Node *tail;
int n;
```

Zaporedje ukazov Sklada in Vrste nad enostransko povezanim seznamom je prikazana na 3.1.

Enostransko povezani seznam lahko učinkovito implementira operaciji Sklada, to sta push(x) in pop(), s katerima dodajamo ter odstranjujemo elemente iz začetka seznama. Operacija push(x) kreira novo vozlišče u z vrednostjo x, nastavi u.next tako, da kaže na stari začetek seznama, novi začetek seznama pa postane u. Na koncu je potebno še povečati vrednost števca vozlišč n za 1.

SLList
```
T push(T x) {
  Node *u = new Node(x);
  u->next = head;
  head = u;
```



Slika 3.1: Zaporedje ukazov Vrste (add(x) in remove()) ter Sklada (pop() in push(y)) nad enostransko povezanim seznamom.

```
  if (n == 0)
    tail = u;
  n++;
  return x;
}
```

Operacija pop() najprej preveri ali je enostransko povezani seznam prazen. Če ni prazen, odstrani začetno vozlišče tako, da nastavi spremenljivko, ki kaže na začetek vozišča na `head = head.next` in zmanjša spremenljivko `n` za 1. Poseben primer je odstranjevanje zadnjega vozlišča, v tem primeru postavimo `tail` na `null`:

SLList

```
T pop() {
  if (n == 0)  return null;
  T x = head->x;
  Node *u = head;
  head = head->next;
  delete u;
  if (--n == 0) tail = NULL;
  return x;
}
```

Časovna zahtevnost operacij push(x) in pop() je $O(1)$.

3.1.1 Operacije Vrste

Enostransko povezani seznam lahko implementira tudi operaciji FIFO ("prvi noter, prvi ven") vrste, to sta add(x) in remove(). Operacija brisanja elementa je identična operaciji pop(), odstrani se torej začetno vozlišče. Obe operaciji se izvedeta v konstantnem času.

SLList

```
T remove() {
  if (n == 0)  return null;
  T x = head->x;
  Node *u = head;
  head = head->next;
  delete u;
  if (--n == 0) tail = NULL;
  return x;
}
```

Dodajanje pa je izvedeno tako, da se novo vozlišče pripne na konec seznama. V večini primerov to naredimo tako, da postavimo `tail.next` = `u`, kjer je `u` novo nastalo vozlišče in vsebuje vrednost `x`. Paziti je treba na poseben primer, ki se zgodi, kadar je seznam prazen, `n` = 0. To pomeni, da je `tail` = `head` = `null`. V tem primeru `tail` in `head` nastavimo tako, da kažeta na `u`.

SLList

```
bool add(T x) {
  Node *u = new Node(x);
  if (n == 0) {
    head = u;
  } else {
    tail->next = u;
  }
  tail = u;
  n++;
  return true;
}
```

Obe operaciji, add(x) in remove(), se izvedeta v konstantnem času.

3.1.2 Povzetek

Sledeči izrek povzame zmožnosti enostransko povezanega seznama SLList:

Izrek 3.1. *Enostransko povezani seznam SLList implementira operacije vmesnika Sklada in (FIFO) Vrste. Operacije* push(x), pop(), add(x) *in* remove() *se izvedejo v* $O(1)$.

Enostransko povezani seznam SLList implementira skoraj vse operacije Degue vrste. Edina manjkajoča operacija je odstranjevanje elementov iz konca enostransko povezanega seznama. Brisanje iz konca enojno povezanega seznama je težavno, saj moramo posodobiti vrednost tail, tako da kaže na vozlišče w, ki je predhodnik našega vozlišča tail. Naše vozlišče w izgleda tako w.next = tail. Na žalost pa je edina možnost da pridemo do vozlišča w ta, da se še enkrat sprehodimo čez celoten seznam, od začetka v vozlišču head, za kar pa potrebujemo n – 2 korakov.

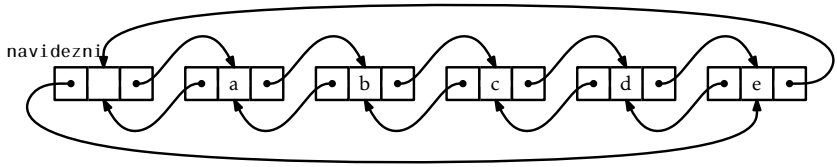
3.2 DLList: Obojestransko povezan seznam

(obojestransko povezan seznam) je zelo podoben SLList le, da ima vsako vozlišče u v DLList referenco na dve vozlišči, u.next, ki mu sledi ter vozlišče u.prev, ki je pred njim.

DLList

```
struct Node {
  T x;
  Node *prev, *next;
};
```

Pri implementaciji SLList, smo ugtovili, da imamo kar nekaj posebnih primerov, na katere moramo paziti. Na primer, pri odstranjevanju zadnjega elementa iz SLList ali pa dodajanju elementa v praznen SLList moramo zagotoviti, da se head (glava) in tail (rep) pravilno posodobita.



Slika 3.2: DLList, ki vsebuje a,b,c,d,e.

V DLList se število teh posebnih primerov znatno poveča. Morda najboljši način, da poskrbimo za vse te posebne primere v DLList je, da uvedemo dummy (navidezno) vozlišče. To je vozlišče brez vsebine, služi pa kot vsebovalnik, čeprav ne vsebuje vozlišč; vsako vozlišče ima next in prev, kjer dummy služi kot vozlišče, ki sledi zadnjemu vozlišču in razporeja prvo vozlišče v seznamu. Tako so vozlišča obojestransko povezava v cikel, kot je prikazano v 3.2.

DLList
```
Node dummy;
int n;
DLList() {
  dummy.next = &dummy;
  dummy.prev = &dummy;
  n = 0;
}
```

Iskanje vozlišče z določenim indeksom v DLList je enostavno; lahko bodisi začnemo pri glavi seznama (dummy.next) in se pomikamo naprej, ali pa začnemo pri repu seznama (dummy.prev) in se pomikamo nazaj. To nam omogoča, da dosežemo i-to vozlišče v času $O(1+\min\{i, n-i\})$:

DLList
```
Node* getNode(int i) {
  Node* p;
  if (i < n / 2) {
    p = dummy.next;
    for (int j = 0; j < i; j++)
      p = p->next;
  } else {
    p = &dummy;
```

```
    for (int j = n; j > i; j--)
      p = p->prev;
  }
  return (p);
}
```

get(i) in set(i,x) operacije so prav tako enostavne. Najprej moramo najti i-to vozlišče, nato pa dobimo ali nastavimo njegovo vrednost x:

DLList

```
T get(int i) {
    return getNode(i)->x;
}
T set(int i, T x) {
  Node* u = getNode(i);
  T y = u->x;
  u->x = x;
  return y;
}
```

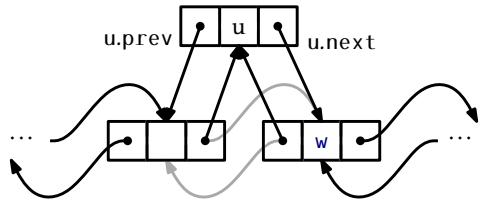
Čas izvajanja teh operacij je določen z strani časa, ki potrebujemo, da najdemo i-to vozlišče in je zato $O(1+\min\{i, n-i\})$.

3.2.1 Dodajanje in odstranjevanje

Če imamo referenco na vozlišče w v DLList in želimo vstaviti vozlišče u pred w, potem je potrebno le nastaviti u.next = w, u.prev = w.prev ter u.prev.next in u.next.prev. (Glej 3.3.) Zahvaljujoč navideznem vozlišču nam ni treba skrbeti, ali vozlišči w.prev in w.next sploh obstajata.

DLList

```
Node* addBefore(Node *w, T x) {
  Node *u = new Node;
  u->x = x;
  u->prev = w->prev;
  u->next = w;
  u->next->prev = u;
  u->prev->next = u;
  n++;
```



Slika 3.3: Dodajanje vozlišča u pred vozlišče w v DLList.

```
  return u;
}
```

Operacija seznama add(i,x) je trivialna za implementacijo. Najti moramo i-to vozlišče v DLList in nato vstavimo novo vozlišče u, ki vsebuje x, tik pred njim.

DLList

```
void add(int i, T x) {
    addBefore(getNode(i), x);
}
```

Edini nekonstantni del časa izvajanja časa izvajanja add(i,x), je čas, ki ga potrebujemo, da najdemo i-to vozlišče (z getNode(i)). Tako se add(i,x) izvede v času $O(1+\min\{i, n-i\})$.

Odstranjevanje vozlišča w iz DLList je enostavno. Potrebujemo samo nastaviti kazalec w.next in w.prev tako, da preskočijo vozlišče w. Uporaba navideznega vozlišča odpravi potrebo po upoštevanju posebnih primerov:

DLList

```
void remove(Node *w) {
  w->prev->next = w->next;
  w->next->prev = w->prev;
  delete w;
  n--;
}
```

Operacija remove(i) je prav tako enostavna. Najdemo vozlišče z indeksom i in ga odstranimo:

DLList

```
T remove(int i) {
  Node *w = getNode(i);
  T x = w->x;
  remove(w);
  return x;
}
```

Edini dragi del te operacije je iskanje `i`-tega vozlišča z operacijo `getNode(i)`. `remove(i)` se torej izvede v času $O(1+\min\{i, n-i\})$.

3.2.2 Povzetek

Naslednji izrek povzema uspešnost `DLList`:

Izrek 3.2. *`DLList` implementira vmesnik `List` (seznam). V tej izvedbi, je časovna zahtevnost operacij* `get(i)`, `set(i,x)`, `add(i,x)` *in* `remove(i)` $O(1+\min\{i, n-i\})$.

Treba je omeniti, da če odmislimo ceno operacije `getNode(i)`, se vse operacije v `DLList` izvedejo v konstantem času. Edina draga operacija v `DLList` je torej iskanje ustreznega vozlišča. Ko imamo dostop do ustreznega vozlišča, se dodajanje, odstranjevanje ali dostop do podatkov v tem vozlišču se izvede v konstantnem času.

To je v popolnem nasprotju z implementacijami `seznama` na osnovi polja 2; v teh izvedbi, lahko ustrezen element najdemo v konstantnem času. Vendar pa dodajanje ali odstranjevanje zahteva premikanje elementov v polju, kar pa načeloma ni operacija, ki se bi izvedla v konstantnem času.

Iz tega razloga, so povezani seznami primerni za primere, kjer lahko reference vozlišč pridobimo iz zunanjih virov. . Na primer kazalci na vozlišča povezanega seznama bi lahko bili shranjeni v `USet`. Za odstranitev elementa `x` iz povezanega seznama, lahko vozlišče, ki vsebuje `x`, hitro najdemo z uporabo `Uset` in vozlišče lahko odstranimo s seznama v konstantnem času.

3.3 SEList: Prostorsko učinkovit povezan seznam

Ena od slabosti povezanih seznamov (poleg časa, ki je potreben za dostop do elementov, ki so globoko v seznamu) je njihova poraba prostora. Vsak člen v DLList zahteva dodatni dve referenci do naslednjega in prejšnjega člena v seznamu. Dve polji v Node sta namenjeni vzdrževanju seznama, le eno polje pa shrambi podatkov.

SEList (Prostorsko-učikovit seznam) zmanjša porabo prostora v duhu preproste ideje. Namesto, da shrani posamezne elemente v DLList, shrani kar tabelo večih elementov. Podrobneje,SEList je parameteriziran s pomočjo *bloka velikosti* b. Vsak posamezen člen v SEList hrani blok, ki vsebuje $b+1$ elementov.

Zaradi kasnejših razlogov bo lažje, če lahko izvedemo Deque operacijo na vsakem bloku. Izbrali bomo podatkovno strukturo BDeque (omejen Deque), izpeljano iz strukture ArrayDeque structure described in 2.4. BDeque se le malo razlikuje od ArrayDeque. Ko se BDeque ustvari, je velikost tabele a kontantna in sicer $b+1$. Pomembna lastnost podatkovne strukture BDeque je možnost dodajanja in odstranjevanja od spredaj ali zadaj v konstantnem času. To je uporabno, ker se elementi prenašajo iz enega bloka v drugega.

SEList

```
class BDeque : public ArrayDeque<T> {
public:
  BDeque(int b) {
    n = 0;
    j = 0;
    array<int> z(b+1);
    a = z;
  }
  ~BDeque() { }
  // C++ Question: Why is this necessary?
  void add(int i, T x) {
    ArrayDeque<T>::add(i, x);
  }
  bool add(T x) {
    ArrayDeque<T>::add(size(), x);
    return true;
  }
```

```
  void resize() {}
};
```

SEList postane dvostransko povezan seznam blokov:

SEList

```
class Node {
public:
  BDeque d;
  Node *prev, *next;
  Node(int b) : d(b) { }
};
```

SEList

```
int n;
Node dummy;
```

3.3.1 Prostorske zahteve

SEList ima zelo tesne omejitve glede števila elementov v bloku. Razen zadnjega bloka vsebujejo najmanj $b-1$ in največ $b+1$ elementov. To pomeni, če SEList vsebuje n elementov, ima največ

$$n/(b-1)+1 = O(n/b)$$

blokov. Pri BDeque vsak blok vsebuje tabelo velikosti $b+1$, ampak vsi razen zadnjega elementa potrebujejo največ konstantno prostora. Prav tako je konstanten tudi neporabljen prostor bloka. To pomeni, da je poraba prostora podatkovne strukture SEList le $O(b+n/b)$. Z izbiro vrednosti b znotraj kontantnega faktorja $\sqrt{n}$, lahko prostorsko potrato približamo spodnji meji $\sqrt{n}$ predstavljeno v poglavju 2.6.2.

3.3.2 Iskanje elementov

Izziv pri podatkovni strukturi SEList je iskanje elementa z indeksom i.Pri čemer lokacija elementa predstavlja 2 dela:

1. Člen u, ki vsebuje blok z indeksom i; in

2. indeks elementa `j` znotraj bloka.

SEList
```
class Location {
public:
  Node *u;
  int j;
  Location() { }
  Location(Node *u, int j) {
    this->u = u;
    this->j = j;
  }
};
```

Pri iskanju bloka, ki vsebuje določen element uporabljamo isti postopek kot pri strukturi `DLList`. Lahko začnemo spredaj in potujemo naprej, ali pa začnemo zadaj in potujemo nazaj, do iskanega člena. Edina razlika je, da pri tej strukturi pri vsakem členu preskočimo celoten blok elementov.

SEList
```
void getLocation(int i, Location &ell) {
  if (i < n / 2) {
    Node *u = dummy.next;
    while (i >= u->d.size()) {
      i -= u->d.size();
      u = u->next;
    }
    ell.u = u;
    ell.j = i;
  } else {
    Node *u = &dummy;
    int idx = n;
    while (i < idx) {
      u = u->prev;
      idx -= u->d.size();
    }
    ell.u = u;
    ell.j = i - idx;
```

```
  }
}
```

Pomembno je, da si zapomnimo, da razen enega bloka, vsak blok vsebuje najmanj $b-1$ elementov, torej smo z vsakim korakom pri iskanju $b-1$ elementov bližje iskanemu elementu. Če iščemo od začetka naprej, lahko dosežemo iskani člen v $O(1+(n-i)/b)$ korakih. Algoritem je odvisen od indeksa i, torej je čas iskanja z indeksom i enak $O(1+\min\{i,n-i\}/b)$.

Ko enkrat vemo kako najti element z indeksom i, lahko z `get(i)` in `set(i,x)` operacijami dobimo ali nastavimo element z poljubnim indeksom v določenem bloku:

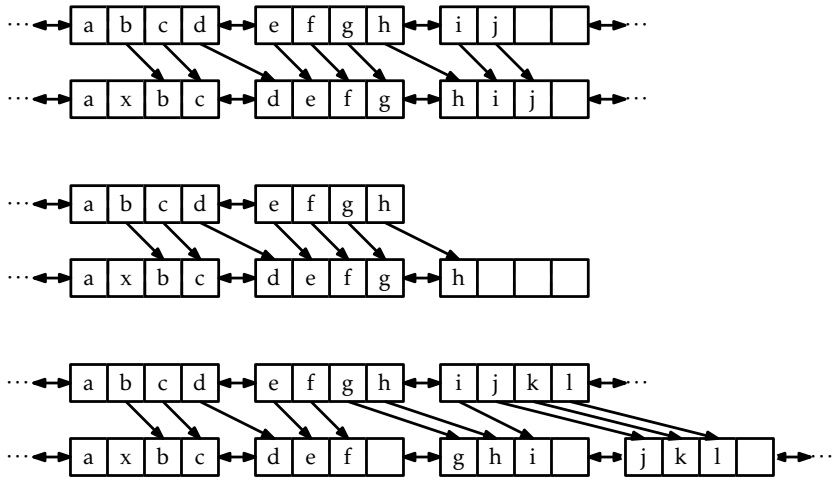
SEList

```
T get(int i) {
  Location l;
  getLocation(i, l);
  return l.u->d.get(l.j);
}
T set(int i, T x) {
  Location l;
  getLocation(i, l);
  T y = l.u->d.get(l.j);
  l.u->d.set(l.j, x);
  return y;
}
```

Čas izvajanja teh operacij sta odvisni od časa iskanja elementa, torej imata enako časovno zahtevnost $O(1+\min\{i,n-i\}/b)$.

3.3.3 Dodajanje elementov

Dodajanje elementov v podatkovno strukturo `SEList` je malo bolj kompleksno. Preden se lotimo splošnih primerov, si poglejmo najlažjo operacijo, `add(x)`, pri kateri se `x` doda na konec seznama. Če je zadnji blok poln(ali ne obstaja, ker še nimamo blokov), potem najprej naredimo nov blok in dodamo v seznam blokov. Sedaj, ko obstaja blok in ni prazen, dodamo `x` zadnjemu bloku.



Slika 3.4: 3 različni scenariji, ki se lahko zgodijo pri dodajanju elementa `x` v `SEList`. (`SEList` ima velikost bloka `b` = 3.)

SEList

```
void add(T x) {
  Node *last = dummy.prev;
  if (last == &dummy || last->d.size() == b+1) {
    last = addBefore(&dummy);
  }
  last->d.add(x);
  n++;
}
```

Dodajanje se malo bolj zakomplicira pri dodajanju v notranjost seznama s pomočjo metode `add(i,x)`. Najprej lociramo `i` da dobimo člen `u` čigar blok vsebuje `i`ti element. Problem nastane, ker hočemo vstaviti element `x` v blok `u`kjer blok `u` že vsebuje `b` + 1 elementov, torej je poln in ni prostora za `x`.

Naj $u_0, u_1, u_2, \ldots$ označujejo `u`, `u.next`, `u.next.next`, in tako naprej. Preiščemo $u_0, u_1, u_2, \ldots$ v iskanju člena, ki ima prostor za `x`. Možne so (glej 3.4):

1. Člen u_r, čigar blok ni poln, najdemo hitro(v $r+1 \leq b$ korakih). V tem primeru izvedemo r zamenjav elementa iz trenutnega v naslednji

blok, da prazen prostor v u_r postane prazen prostop v u_0. Nato vstavimo x v blok u_0.

2. Prav tako hitro (v $r+1 \leq b$ korakih) pridemo do konca seznama blokov. V tem primeru preprosto dodamo nov prazen blok na konec seznama in nadaljujemo s 1. scenarijem.

3. Po b korakih ne nardemo bloka, ki ni poln. V tem primeru, je $u_0,\ldots,u_{b-1}$ zaporedje $b$blokov, ki vsebujejo vsak po $b+1$ elementov. Vstavimo nov blok u_b na konec zaporedja in *razširimo* prvotnih $b(b+1)$ elementov tako, da vsak blok $u_0,\ldots,u_b$ vsebuje natanko b elementov. Sedaj blok u_0 vsebuje le b elementov in ima prostor za x, ki ga vstavljamo.

SEList

```
void add(int i, T x) {
  if (i == n) {
    add(x);
    return;
  }
  Location l; getLocation(i, l);
  Node *u = l.u;
  int r = 0;
  while (r < b && u != &dummy && u->d.size() == b+1) {
    u = u->next;
    r++;
  }
  if (r == b) {// b blocks each with b+1 elements
    spread(l.u);
    u = l.u;
  }
  if (u == &dummy) { // ran off the end - add new node
    u = addBefore(u);
  }
  while (u != l.u) { // work backwards, shifting elements
    u->d.add(0, u->prev->d.remove(u->prev->d.size()-1));
    u = u->prev;
  }
  u->d.add(l.j, x);
```

```
    n++;
}
```

Čas izvajanja operacije add(i,x) je različen, glede na to, kateri od treh scenarijev zgoraj se zgodi. Primera 1 in 2 vsebujeta preiskovanje in prestavljanje elementov pri največ b b blokih, torej je časovna zahtevnost $O(b)$. Primer 3 vsebuje spread(u) metodo, ki premakne $b(b+1)$ elementov, kar vzame $O(b^2)$ časa. Če ignoriramo ceno 3. scenarija(ki ga bomo upoštevali kasneje v amortizaciji), to pomeni, da je celotna časovna zahtevnost lociranja ija in izvajanja vstavljanja elementa x $O(b+\min\{i, n-i\}/b)$.

3.3.4 Odstranjevanje elementov

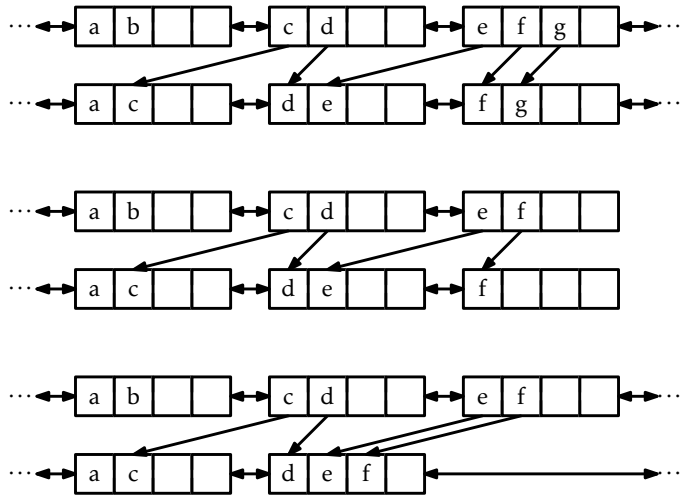
Odstranjevanje elementa iz podatkovne strukture SEList je podobno dodajanju elementov vanjo. Najprej lociramo vozlišče u, ki vsebuje element z indeksom i. Zdaj moramo biti pripravljeni na primer, ko elementa ne moremo zbrisati iz vozlišča u, ne da bi u-jev blok postal manjši od $b-1$.

Ponovno naj vozlišča $u_0, u_1, u_2, \ldots$ označujejo u, u.next, u.next.next in tako naprej. Med vozlišči poiščemo tisto, iz katerega si lahko sposodimo element, s katerim bo velikost bloka vozlišča u_0 vsaj $b-1$. To lahko storimo na 3 načine (3.5):

1. Hitro (v $r+1 \leq b$ korakih) najdemo vozlišče, čigar blok vsebuje več kot $b-1$ elementov. V tem primeru izvedemo r menjav elementa iz enega bloka v prejšnji blok, tako da dodaten element v u_r postane dodaten element v u_0. Nato lahko odstranimo ustrezen element iz bloka vozlišča u_0.

2. Hitro (v $r+1 \leq b$ korakih) se sprehajamo s konca seznama blokov. V tem primeru je u_r zadnji blok, zato zanj ni nujno, da vsebuje vsaj $b-1$ elementov. Nadaljujemo kot zgoraj. Sposodimo si element iz u_r in iz njega naredimo dodaten element v u_0. Če blok vozlišča u_r zaradi te menjave postane prazen, ga odstranimo.

3. Po b korakih ni več bloka, ki bi vseboval več kot $b-1$ elementov. V tem primeru je $u_0, \ldots, u_{b-1}$ zaporedje blokov b, kjer vsak izmed



Slika 3.5: Trije scenariji, ki se zgodijo ob odstranjevanju predmeta `x` znotraj podatkovne strukture `SEList`. (Velikost bloka tega `SEList`a je $b = 3$.)

njih vsebuje $b-1$ elementov. Teh $b(b-1)$ elementov *združimo* v $u_0, \ldots, u_{b-2}$, tako da vsak izmed novih $b-1$ blokov vsebuje natančno b elementov, vozlišče u_{b-1}, ki je zdaj prazno, pa zbrišemo. Blok vozlišča u_0 zdaj vsebuje b elementov, zato lahko iz njega odstranimo ustrezen element.

codeimportods/SEList.remove(i)

Operaciji `add(i,x)` in `remove(i)` imata enak čas izvajanja, $O(b+\min\{i, n-i\}/b)$, če ne poštevamo stroška metode `gather(u)`, ki jo uporabimo v 3. načinu odstranjevanja.

3.3.5 Amortizirana analiza širjenja in združevanja

Razmislimo o strošku metod `gather(u)` in `spread(u)`), ki sta lahko izvršeni preko metod `add(i,x)` in `remove(i)`. Metodi sta sledeči:

SEList

```
void spread(Node *u) {
  Node *w = u;
```

```
  for (int j = 0; j < b; j++) {
    w = w->next;
  }
  w = addBefore(w);
  while (w != u) {
    while (w->d.size() < b)
      w->d.add(0, w->prev->d.remove(w->prev->d.size()-1));
    w = w->prev;
  }
}
```

SEList

```
void gather(Node *u) {
  Node *w = u;
  for (int j = 0; j < b-1; j++) {
    while (w->d.size() < b)
      w->d.add(w->next->d.remove(0));
    w = w->next;
  }
  remove(w);
}
```

Čas izvajanja vsake metode je odvisen od dveh ugnezdenih zank. Obe, notranja in zunanja zanka, se izvršita največ $b+1$ krat. Celoten čas izvajanja vsake metode je tako $O((b+1)^2) = O(b^2)$. Ne glede na vse, naslednji izrek dokaže, da se metodi izvršita na največ enem izmed mnogih b klicev metod `add(i,x)` ali `remove(i)`.

Lema 3.1. *Če je ustvarjena prazna podatkovna struktura* SEList *in je izvršena katera koli ponovitev od $m \geq 1$ klicev metod* `add(i,x)` *in* `remove(i)`*, potem je celoten čas izvajanja vseh klicov metod* `spread()` *in* `gather()` *enak $O(bm)$.*

Dokaz. Uporabili bomo potencialno metodo amortiziranih analiz. Predpostavimo, da je vozlišče `u` *ranljivo*, če njegov blok ne vsebuje `b` elementov (`u` je ali zadnje vozlišče ali pa vsebuje $b-1$ ali $b+1$ elementov). Vozlišče je *robustno*, če njegov blok vsebuje `b` elementov. *Potencial* podatkovne strukture SEList določimo na podlagi števila ranljivih vozlišč, ki jih vsebuje. Osredotočili se bomo samo na metodo `add(i,x)` in njeno relacijo s

številom klicev metode spread(u). Analiza metod remove(i) in gather(u) je identična.

Opazimo, da se v primeru, ko se pri metodi add(i, x) izvrši scenarij 1, spremeni velikost bloka samo enemu vozlišču, vozlišču u_r. Zato se tudi največ eno vozlišče, vozlišče u_r, spremeni iz robustnega v ranljivo, ostala vozlišča pa ohranijo velikost, tako da se število ranljivih vozlišč poveča za 1. Sledi, da se potencial podatkovne strukture SEList poveča za največ 1 v scenarijih 1 in 2.

Če se izvrši scenarij 3, se izvrši, ker so vsa vozlišča $u_0, \ldots, u_{b-1}$ ranljiva. Nato se pokliče metoda spread(u_0), ki b ranljivih vozlišč zamenja z $b+1$ robustnimi vozlišči. Na koncu v blok vozlišča u_0 dodamo x, ki vozlišče naredi ranljivo. V splošnem se potencial zniža za $b-1$.

Potencial (ki šteje število ranljivih vozlišč) ni nikoli manjši od 0. Vsakič, ko se izvrši scenarij 1 ali scenarij 2, se potencial zviša za največ 1. Vsakič, ko se zgodi scenarij 3, se potencial zniža za $b-1$. V vsakem primeru scenarija 3 je vsaj $b-1$ primerov scenarija 1 ali scenarija 2. Tako je za vsak klic metode spread(u) vsaj b klicev metode add(i, x). To potrdi dokaz. □

3.3.6 Povzetek

Sledeči izrek povzema učinkovitest podatkovne strukture SEList:

Izrek 3.3. *Podatkovna struktura SEList implementira List vmesnik. Čeprav se ne ozira na stroška klicev metod* spread(u) *in* gather(u), *SEList z* b *velikostjo bloka podpira operacije*

- get(i) *in* set(i, x) *v času* $O(1+\min\{i, n-i\}/b)$ *na operacijo; in*
- add(i, x) *in* remove(i) *v času* $O(b+\min\{i, n-i\}/b)$ *na operacijo.*

Če začnemo s praznim SEList, bo skupno porabljen čas med vsemi klici metod spread(u) *in* gather(u) *za vsako ponovitev od m* add(i, x) *in* remove(i) *operacij enak* $O(bm)$.

Prostor (merjen v besedah)[1] *porabljen za podatkovno strukturo SEList, ki hrani* n *elementov je* $n + O(b + n/b)$.

[1] Poglavje 1.4 za razlago o merjenju spomina.

SEList je kompromis med podatkovnima strukturama ArrayList in DLList, kjer je njuna relativna mešanica odvisna od bloka velikosti b. Pri skrajnosti b = 2, vsako vozlišče v SEList (in tudi v DLList)hrani največ 3 vrednosti. Pri drugi skrajnosti b > n, so vsi elementi shranjeni v eni tabeli, tako kot pri ArrayList. Med tema skrajnostma je kompromis v času, ki je potreben za dodajanje ali odstranjevanje elementa in časom, ki je potreben za lociranje točno določenega predmeta.

3.4 Razprave in vaje

Tako enosmerno-povezani kot dvosmerno-povezani seznami so uveljavljene tehnike, uporabljene v programih že več kot 40 let. O njih naprimer razpravlja Knuth [**?**, Sections 2.2.3–2.2.5].Tudi podatkovna struktura SEList je uveljavljena kot dobro poznana vaja podatkovnih struktur.SEList včasih imenujemo tudi *Odvit povezan seznam* [**?**].

Na prostoru v dvosmerno-povezanem seznamu lahko prihranimo z uporabo t.i. XOR-seznamov. V XOR-seznamu vsako vozlišče u vsebuje samo en kazalec, imenovan u.nextprev, ki vsebuje bitna XOR kazalca u.prev in u.next. Seznam potrebuje za delovanje dva kazalca, eden kaže na dummy vozlišče, drug pa na dummy.next (prvo vozlišče, ali dummy vozlišče, če je seznam prazen). Ta tehnika izrablja dejstvo, da če imamo dva kazalca na u in u.prev, lahko izluščimo u.next s pomočjo naslednje formule

$$\texttt{u.next} = \texttt{u.prev}\hat{}\texttt{u.nextprev} \ .$$

(Tukaj nam operator ^ izračuna bitni XOR dveh argumentov.) Ta tehnika programsko kodo zakomplicira in implementacija v vseh programskih jezikih, kot je naprimer Java ali Python, ki imajo mehanizme za sproščanje pomnilnika (garbage collector) ni možna. Tukaj podamo dvosmerno-povezan seznam, ki za delovanje potrebuje samo en kazalec na vozlišče.

Za referenco o podrobnejši razpravi XOR seznamov si poglej članek Sinhe [**?**] .

Naloga 3.1. Zakaj ni možna uporaba praznega vozlišča v SLList za izogib posebnih primerov, ki se zgodijo pri operacijah push(x), pop(), add(x), and remove()?

Naloga 3.2. Napišite `SLList` (enosmerno-povezan seznam) metodo `secondLast()`, ki vrne predzadnji element v `SLList`. Metodo implementirajte brez uporabe članovske spremenljivke `n`, ki skrbi za velikost seznama.

Naloga 3.3. Na enosmerno-povezanem seznamu implementirajte naslednje `List` operacije: `get(i)`, `set(i,x)`, `add(i,x)` in `remove(i)`. Vse metode se naj izvedejo v $O(1+i)$ časovni zahtevnosti.

Naloga 3.4. Na enosmerno-povezanem seznamu `SLLIST` implementirajte metodo `reverse()`, ki obrne vrstni red elementov v seznamu. Metoda naj teče v $O(n)$ časovni zahtevnosti. Ni dovoljena uporaba rekurzije in implementacija z drugimi časovnimi strukturami. Prav tako ni dovoljeno ustvarjati nova vozlišča.

Naloga 3.5. Napišite metodo za enosmerno `SLList` in dvosmerno `DLList` povezan seznam `checkSize()`. Metoda naj se sprehodi skozi seznam in prešteje število vozlišč. Če se prešteto število vozlišč ne ujema z vrednostjo shranjeno v spremenljivki `n`, naj metoda vrže izjemo. V primeru da se števila ujemata, metoda ne vrača ničesar.

Naloga 3.6. Ponovno napišite kodo za `addBefore(w)` operacijo, ki ustvari novo vozlišče `u` in ga doda v dvosmerno-povezan seznam tik pred vozliščem `w`. Tudi, če se vaša koda ne popolnoma ujema s kodo iz te knjige, je metoda še vseeno lahko pravilna. Najbolje, da metodo stestirate in preverite.

Z naslednjimi vajami bomo izvajali manipulacije na dvosmerno-povezanih seznamih. Vse vaje morate dokončati brez dodeljevanja novih vozlišč ali začasnih seznamov. Vse naloge se lahko rešijo s spreminjanjem vrednosti `prev` in `next` v že obstoječih vozliščih.

Naloga 3.7. Napišite metodo za dvosmerno-povezan seznam `isPalindrome()`, ki vrne `true`, če je seznam *palindrom*, npr., element na poziciji `i` je enak elementu na poziciji $n-i-1$ za vsak $i \in \{0,\ldots,n-1\}$. Metoda se naj izvede v $O(n)$ časovni zahtevnosti.

Naloga 3.8. Napišite novo metodo `rotate(r)`, ki obrne dvosmerno-povezan seznam tako, da element na poziciji `i` postane element $(i+r) \bmod n$. Ta metoda se običajno izvaja v $O(1+\min\{r, n-r\})$ časovni zahtevnosti in ne spreminja vozlišč v seznamu.

Naloga 3.9. Napišite metodo `truncate(i)`, ki odseka dvojno-povezan seznam na poziciji `i`. Po izvedbi metode naj bo velikost seznama `i`, vsebuje pa naj samo elemente na intervalu $0,\ldots,i-1$. Metoda naj vrne dvojno-povezan seznam `DLList` in vsebuje elemente na intervalu $i,\ldots,n-1$. Metoda naj se izvede v $O(\min\{i, n-i\})$ časovni zahtevnosti.

Naloga 3.10. Napišite metodo dvojno-povezanega seznama `DLList absorb(l2)`, ki za vhodni parameter prejme dvojno-povezan seznam `DLList l2`, ter sprazni njegovo vsebino in jo pripne na konec svojega seznama. Naprimer, če $l1$ vsebuje a,b,c in $l2$ vsebuje d,e,f, po klicu $l1$.absorbe($l2$) $l1$ vesbuje a,b,c,d,e,f, $l2$ pa bo prazen.

Naloga 3.11. Napišite metodo `deal()`, ki iz pod. strukture `DLList` odstrani vse elemente z lihimi indeksi in vrne `DLList`, ki vsebuje izbrisane elemente. Naprimer, če $l1$ vsebuje a,b,c,d,e,f, potem bo po klicu $l1$.deal() vseboval a,c,e, metoda pa bo vrnila seznam , ki vsebuje elemente b,d,f.

Naloga 3.12. Napišite metodo `reverse()`, ki obrne vrstni red elementov v pod. strukturi `DLList`.

Naloga 3.13. V tej vaji boste implementirali urejanje pod. strukture `DLList` z zlivanjem, kot je opisano v poglavju 11.1.1.

1. Napišite metodo pod. strukture `DLList takeFirst(l2)`, ki odstrani prvo vozlišče iz $l2$ ter ga doda na konec seznama, nad katerim je bila metoda klicana. Metoda je enakovredna klicu `add(size(),l2.remove(0))`, vendar pri tem ne ustvari novega vozlišča.

2. Napišite statično metodo pod. strukture `DLList merge(l1,l2)`, ki kot argument dobi dva urejena seznama $l1$ in $l2$, ju združi ter vrne nov urejen seznam. Seznama $l1$ ter $l2$ se v metodi izpraznita. Naprimer, če $l1$ vsebuje a,c,d in $l2$ vsebuje b,e,f , metoda vrne nov seznam, ki vsebuje a,b,c,d,e,f.

3. Napišite metodo pod. strukture `DLList sort()`, ki uredi elemente v seznamu z uporabo urejanja z zlivanjem. Ta rekurzivni algoritem deluje tako:

(a) Če je velikost seznama 0 ali 1, je seznam urejen. V nasprotnem primeru...

(b) Z uporabo metode `truncate(size()/2)`, razdeli seznam v dva seznama `l1` in `l2` , ki sta približno enake velikosti.

(c) Rekurzivno uredi `l1`.

(d) Rekurzivno uredi `l2`.

(e) Združi `l1` in `l2` v en urejen seznam.

Naslednje vaje so naprednejše ter zahtevajo jasno razumevanje kaj se dogaja z najmanjšo vrednostjo shranjeno v skladu ali vrsti, ko dodajamo ter odstranjujemo elemente.

Naloga 3.14. Zasnuj ter implementiraj podatkovno strukturo `MinStack`, ki hrani primerljive elemente in podpira skladovne operacije `push(x)`, `pop()` ter `size()`. Poleg tega podpira tudi operacijo `min()`, ki vrne trenutno najmanjšo vrednost v skladu. Vse operacije naj se izvedejo v konstantnem času.

Naloga 3.15. Zasnuj ter implementiraj podatkovno strukturo `MinQueue`, ki hrani primerljive elemente in podpira operacije vrste: `add(x)`, `remove()` in `size()`. Poleg tega vsebuje tudi operacijo `min()`, ki vrne trenutno najmanjšo vrednost v vrsti. Vse operacije naj se izvedejo v konstantnem amortiziranem času.

Naloga 3.16. Zasnuj ter implementiraj podatkovno strukturo `MinDeque`, ki hrani primerljive elemente in podpira operacije obojestranske vrste: `addFirst(x)`, `addLast(x)`, `removeFirst()`, `removeLast()` in `size()`. Poleg tega vsebuje tudi operacijo `min()`, ki vrne trenutno najmanjšo vrednost v obojestranski vrsti. Vse operacijo nase se izvedejo v konstantnem amortiziranem času.

Naslednje vaje preverijo razumevanje implementacije in analize prostorsko učinkovitega povezanega seznama(`SEList`).

Naloga 3.17. Dokaži, da se operacije pod. strukture `SEList` uporabljene kot sklad (`SEList` spreminjata le operaciji `push(x)` ≡ `add(size(), x)` in `pop()` ≡ `remove(size() – 1)`)), izvedejo v konstantnem amortiziranem času neodvisno od vrednosti b.

Naloga 3.18. Zasnuj ter implementiraj različico pod. strukture `SEList`, ki izvede vse operacije pod. strukture `DLList` v konstantnem amortiziranem času na vsako operacijo, neodvisno od vrednosti b.

Naloga 3.19. Kako bi uporabil bitno operacijo ekskluzivni ali(XOR) za zamenjavo vrednosti dveh celoštevilskih(`int`) spremenljivk brez, da bi uporabil tretjo spremenljivko?

g

Poglavje 4

Preskočni seznami

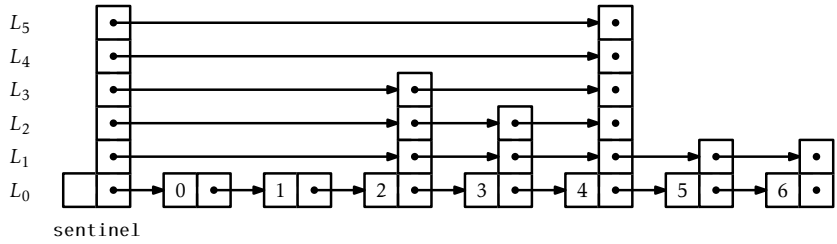
V tem poglavju bomo govorili o lepi podatkovni strukturi: preskočnem seznamu, ki ima veliko možnosti uporabe. Z uporabo preskočnega seznama lahko implementiramo `List`, ki ima časovne zahtevnosti operacij `get(i)`, `set(i,x)`, `add(i,x)`, in `remove(i)` $O(\log n)$. Prav tako lahko implementiramo `SSet`, v katerem vse operacije potrebujejo $O(\log n)$ pričakovanega časa.

Učinkovitost preskočnega seznama je povezana z njegovo naključnostjo. Ko je nov element dodan preskočnemu seznamu, ta uporabi metodo metanja kovanca za določitev višine novega elementa. Učinek preskočnega seznama je odvisen od pričakovanih izvajanj in dolžine poti. To pričakovanje pa je povezano z uporabo metode meta kovanca. V implementaciji je metoda meta kovanca simulirana z uporabo generatorja naključnih števil.

4.1 Osnovna struktura

Konceptualno je preskočni seznam zaporedje enojno povezanih seznamov $L_0,\ldots,L_h$.Vsak seznam L_r vsebuje podniz elementov v L_{r-1}. Začnimo z vhodnim seznamom L_0, ki vsebuje `n` elementov in naredimo L_1 iz L_0, L_2 iz L_1, in tako naprej. Elementi v L_r so pridobljeni z metanjem kovanca za vsak element, `x`, v L_{r-1} in dodajo `x` v L_r, če kovanec "pokaže"glavo. To delamo, dokler ne naredimo praznega seznama L_r. Primer preskočnega seznama je prikazan na sliki 4.1.

Za vsak element `x`, v preskočnem seznamu imenujemo *višina* `x` največjo



Slika 4.1: Preskočni seznam s sedmimi elementi.

vrednost r, kjer se x pojavi v L_r. Tako imajo na primer elementi, ki se pojavijo samo v L_0, višino 0. Če pomislimo, ugotovimo, da je višina x ustreza naslednjemu eksperimentu: Mečimo kovanec tako dolgo, dokler ne bo pokazal cifre. Kolikokrat je pokazal glavo? Odgovor, ne presenetljivo, je, da je pričakovana višina vozlišča enaka 1. (Pričakovali smo, da bomo kovanec vrgli dvakrat, da dobimo cifro, vendar nismo šteli zadnjega meta). *Višina* preskočnega seznama je višina njegovega najvišjega vozlišča.

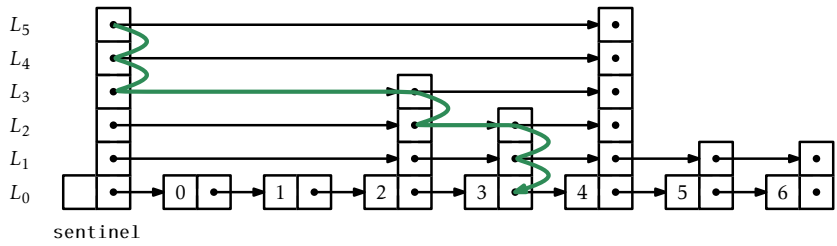
Na koncu vsakega seznama je posebno vozlišče, imanovano *stražar* , ki predstavlja statista za seznam. Glavna lastnost preskočnega seznama je, da obstaja kratka pot, imanovana *pot iskanja*, od stražarja v L_h do vsakega vozlišča v L_0. Narediti pot iskanja za posamezno vozlišče uje preprosto (glej 4.2) : Začnemo v zgornjem levem kotu preskočnega seznama (stražar je v L_h) in se premikamo desno toliko časa, dokler ne gremo preko vozlišča u, nato pa se premaknemo korak nižje v spodnji seznam.

Natančneje, za izdelati pot iskanja za vozlišče u v L_0, začnemo pri stražarju w v L_h. Nato pregledamo w.next. Če w.next vsebuje element, ki se pojavi pred u v L_0, nastavimo w = w.next, sicer se premaknemo navzdol in nadaljujemo iskanje pojavitve w v seznamu L_{h-1}. Postopek ponavljamo dokler na dosežemo predhodnika od u v L_0.

Rešitev, ki si jo bomo podrobneje pogledali v 4.4, nam pokaže, da je pot iskanja dokaj kratka:

Lema 4.1. *Pričakovana dolžina poti iskanja za vsako vozlišče* u *v* L_0 *je največ* $2\log n + O(1) = O(\log n)$.

Prostorsko učinkovit način za implementacijo preskočnega seznama je ta, da definiramo Vozlisce, u, ki je sestavljen iz podatka x in polja



Slika 4.2: The search path for the node containing 4 in a skiplist.

kazalcev `next`, kjer `u.next[i]` kaže na naslednika `u`-ja v seznamu L_i. Na ta način je podatek `x` v vozlišču stored samo enkrat, čeprav se `x` pojavlja v različnih seznamih.

SkiplistSSet

```
struct Node {
  T x;
  int height;      // length of next
  Node *next[];
};
```

V naslednjih dveh podpoglavjih bomo govorili o dveh različnih uporabah preskočnih seznamov. Pri obeh je L_0 shranjena glavna struktura (seznam elementov ali sortiran niz elementov). Glavna razlika med tema dvema strukturama je v načinu premikanja po poti iskanja; drugače povedano, razlikujeta se v tem, kako se odločajo, ali gre pot iskanja do L_{r-1} ali le do L_r.

4.2 SkiplistSSet: Učinkovit SSet

`SkiplistSSet` uporablja preskočni seznam za implementirati `SSet` vmesnik. Ko ga uporabljamo na ta način, so v seznamu L_0 shranjeni elementi `SSet`-a v urejenem vrstnem redu. Metoda `find(x)` deluja tako, da sledi poti iskanja za najmanjšo vrednostjo `y`, kjer je `y` $\geq$ `x`:

SkiplistSSet

```
Node* findPredNode(T x) {
```

```
  Node *u = sentinel;
  int r = h;
  while (r >= 0) {
    while (u->next[r] != NULL
                && compare(u->next[r]->x, x) < 0)
      u = u->next[r]; // go right in list r
    r--; // go down into list r-1
  }
  return u;
}
T find(T x) {
  Node *u = findPredNode(x);
  return u->next[0] == NULL ? null : u->next[0]->x;
}
```

Sledenje poti iskanja za `y` je preprosto: ko se nahajamo v določenem vozlišču `u` v L_r, pogledamo v desno z `u.next[r].x`. Če je `x` > `u.next[r].x`, se premaknemo za eno mesto v desno v L_r; sicer se premaknemo navzdol v L_{r-1}. Vsak korak (desno ali navzdol) v takem iskanju potrebuje konstanten čas; potemtakem, po 4.1, je pričakovani čas izvajanja `find(x)` enak $O(\log n)$.

Preden lahko dodamo element v `SkipListSSet`, potrebujemo metodo, ki nam bo simulirala met kovanca za določitev višine `k` novega vozlišča. To naredimo tako, da si izberemo poljubno število `z` in štejemo število zaporednih enic v dvojiškem zapisu števila `z`:[1]

SkiplistSSet

```
int pickHeight() {
  int z = rand();
  int k = 0;
  int m = 1;
  while ((z & m) != 0) {
    k++;
    m <<= 1;
  }
  return k;
```

[1] Ta metoda ne ponazarja popolnoma eksperiment metanja kovanca saj bo vrednost `k` vedno manjša od števila bitov v `int`. Kakorkoli, to bo imelo malenkosten vpliv dokler ne bo število elementov v strukturi veliko večje kot $2^{32} = 4294967296$.

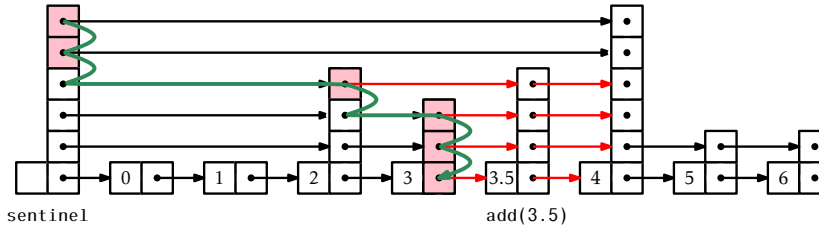
```
}
```

Pri izvedbi metode add(x) v SkiplistSSet smo najprej poiskali x in ga nato dodali v več seznamov $L_0,\ldots,L_k$, kjer je k izbran s pomočjo pickHeight() metode. Najlažji način za narediti to je s pomočjo polja, sklad, ki hrani sled vozlišč, kjer se je pot iskanja spustila iz seznama L_r v L_{r-1}. Natančneje, sklad[r] je vozlišče v L_r kjer se je pot iskanja nadaljevala en nivo nižje, v seznamu L_{r-1}. Vozlišča, ki smo jih prilagodili za vstaviti x so točno vozlišča stack[0],...,stack[k]. Koda v nadaljevanju prikazuje implementacijo algoritma za add(x):

SkiplistSSet

```
bool add(T x) {
  Node *u = sentinel;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u->next[r] != NULL
               && (comp = compare(u->next[r]->x, x)) < 0)
      u = u->next[r];
    if (u->next[r] != NULL && comp == 0)
      return false;
    stack[r--] = u;        // going down, store u
  }
  Node *w = newNode(x, pickHeight());
  while (h < w->height)
    stack[++h] = sentinel; // height increased
  for (int i = 0; i < w->height; i++) {
    w->next[i] = stack[i]->next[i];
    stack[i]->next[i] = w;
  }
  n++;
  return true;
}
```

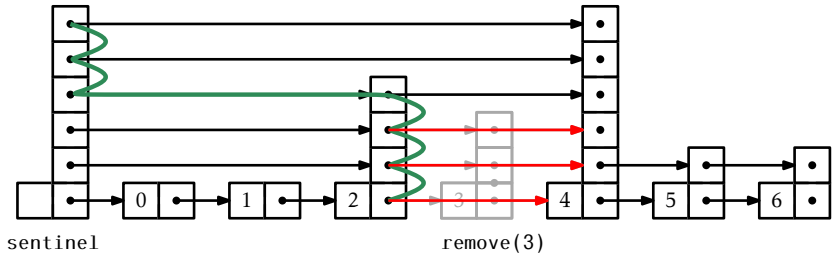
Brisanje elementa x je podobno vstavljanju, le da pri tej metodi ni potrebe po skladu za hranjenje poti iskanja. Brisanje je lahko opravljeno s sledenjem poti iskanja. Ko iščemo x, vedno ko se premaknemo korak navzdol iz vozlišča u, preverimo, če je u.next.x = x in če je, odstranimo u iz seznama:



Slika 4.3: Dodajanje vozlišča 3.5 v preskočni seznam. Vozlišča shranjena v `sklad` so označena.

SkiplistSSet

```
bool remove(T x) {
  bool removed = false;
  Node *u = sentinel, *del;
  int r = h;
  int comp = 0;
  while (r >= 0) {
    while (u->next[r] != NULL
               && (comp = compare(u->next[r]->x, x)) < 0) {
      u = u->next[r];
    }
    if (u->next[r] != NULL && comp == 0) {
      removed = true;
      del = u->next[r];
      u->next[r] = u->next[r]->next[r];
      if (u == sentinel && u->next[r] == NULL)
        h--; // skiplist height has gone down
    }
    r--;
  }
  if (removed) {
    delete del;
    n--;
  }
  return removed;
}
```



Slika 4.4: Brisanje vozlišča 3 iz preskočnega seznama.

4.2.1 Povzetek

Naslednji teorem povzema uporabnost preskočnega seznama, ko ga uporabljamo za implementacijo sortiranih nizov:

Izrek 4.1. *`SkiplistSSet` je uporabljen za izvedbo vmesnika `SSet`. `SkiplistSSet` opravi operacije* `add(x)` *(dodaj),* `remove(x)` *(odstrani), and* `find(x)` *(najdi) v pričakovanem času* $O(\log n)$ *na operacijo.*

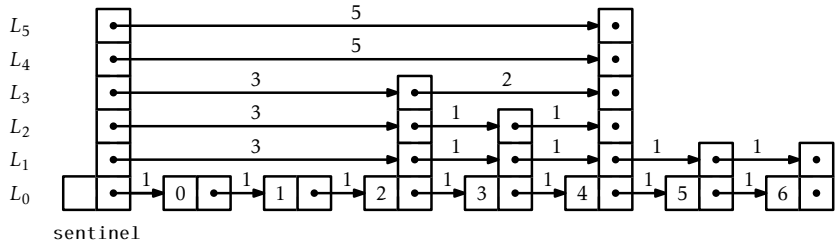
4.3 `SkiplistList`: Učinkovit naključni dostop `List`

`SkiplistList` implementira vmesnik `List` z uporabo preskočnega seznama. V `SkiplistList`, L_0 vsebuje elemente seznama v istem zaporedju, kot so ti razvrščeni v seznamu. Tako kot v `SkiplistSSet`, lahko elemente dodajamo, brišemo ali do njih dostopamo v $O(\log n)$ časa.

Da to lahko dosežemo, je potrebno najti iskalno pot do `i`-tega elementa v L_0. Najlažji način je opredeliti pojem *dolžine* nivoja v nekem seznamu L_r. Vsak nivo v seznamu L_0 definiramo kot 1. Dolžina nivoja, `e`, v L_r, `r` > 0, je definirana kot vsota dolžin nivojev, ki so pod `e` v L_{r-1}. Dolžina `e`-ja je ekvivalenčna številu nivojev v L_0, ki so pod `e`. Poglej 4.5 za primer preskočnega seznama z dolžino njegovih nivojev. Ker so nivoji preskočnega seznama shranjeni v polju, lahko na enak način shranjujemo tudi dolžino:

SkiplistList

```
struct Node {
```



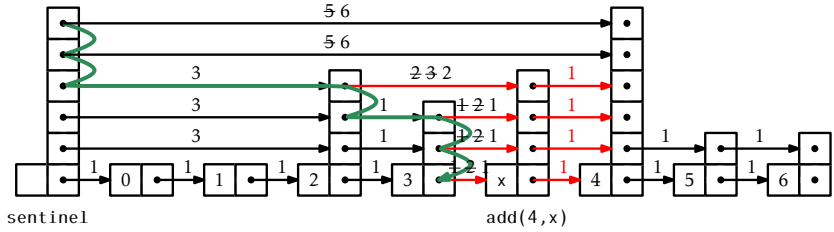
Slika 4.5: Dolžine nivojev v preskočnem seznamu.

```
  T x;
  int height;     // length of next
  int *length;
  Node **next;
};
```

Uporabna lastnost opredelitve dolžin je, da če smo trenutno v vozlišču, ki se nahaja na poziciji `j` v L_0 in sledimo nivoju dolžine ℓ, se potem premaknemo v vozlišče, ki se nahaja na mestu $j+\ell$ v seznamu L_0. Tako lahko, ko sledimo iskalni poti, ohranjamo vrednost pozicije, `j`, trenutnega vozlišča v L_0. Ko smo v vozlišču, `u`, v L_r, gremo desno če `j` plus dolžina nivoja `u.next[r]` manj kot `i`. V nasprotnem primeru, se pomaknemo navzdol v L_{r-1}.

SkiplistList

```
Node* findPred(int i) {
  Node *u = sentinel;
  int r = h;
  int j = -1;   // the index of the current node in list 0
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    r--;
  }
  return u;
}
```



Slika 4.6: Dodajanje v `SkiplistList`.

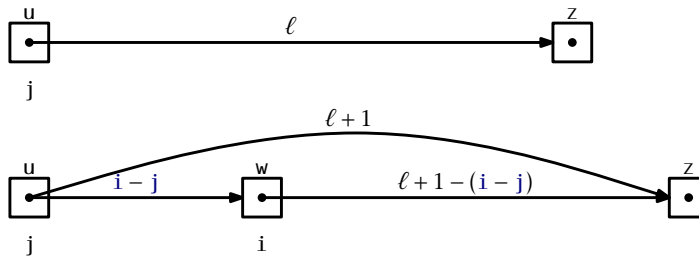
SkiplistList

```
T get(int i) {
  return findPred(i)->next[0]->x;
}
T set(int i, T x) {
  Node *u = findPred(i)->next[0];
  T y = u->x;
  u->x = x;
  return y;
}
```

Ker je najtežji del operacij `get(i)` in `set(i,x)` iskanje `i`-tega vozlišča v L_0, se operacije izvedejo v $O(\log n)$ časa.

Dodajanje elementa v `SkiplistList` na pozicijo, `i`, je enostavno. Za razliko od dodajanja v `SkiplistSSet`, vemo da bo vozlišče dejansko dodano, zato lahko hkrati dodajamo in iščemo lokacijo za novo vozlišče. Najprej izberemo višino, `k`, novega vozlišča, `w`, nato sledimo iskalni poti `i`. Vsakič ko se iskalna pot premakne navzdol od L_r z $r \le k$, spojimo `w` v L_r. Dodatno moramo biti pozorni, da se dolžina nivojev pravilno osvežuje. Poglej 4.6.

Pozorni moramo biti, da vsakič ko se iskalna pot v vozlišču premakne nivo nižje, `u`, v L_r, se dolžina nivoja `u.next[r]` poveča za ena, ker dodajamo element pod nivo na poziciji `i`. Spoj vozlišča `w` med vozlišča, `u` in`z`, deluje kot je prikazano v 4.7. Ko sledimo iskalni poti, shranjujemo tudi pozicijo, `j`, od `u` v L_0. Zato, vemo da je dolžina nivoja od `u` do `w` enaka `i` − `j`. Sklepamo lahko da je dolžina nivoja od `w` do `z` iz dolžine, ℓ, od nivoja `u` do `z`. Potemtakem, lahko spojimo v `w` in osvežimo dolžine nivojev



Slika 4.7: Posodabljanje dolžine nivojev, med spajanjem vozlišča w v preskočni seznam.

v konstantnem času.

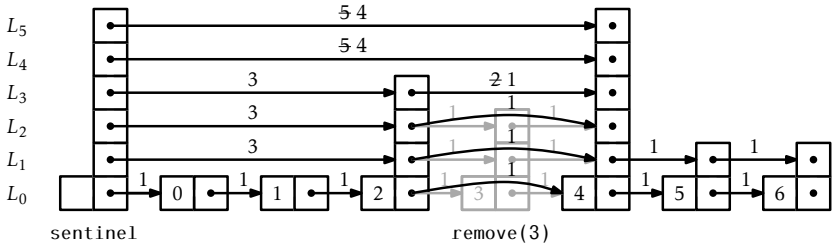
Postopek izgleda veliko bolj kompleksen kot v resnici je. Koda je pravzaprav zelo enostavna:

SkiplistList

```
void add(int i, T x) {
  Node *w = newNode(x, pickHeight());
  if (w->height > h)
    h = w->height;
  add(i, w);
}
```

SkiplistList

```
Node* add(int i, Node *w) {
  Node *u = sentinel;
  int k = w->height;
  int r = h;
  int j = -1; // index of u
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    u->length[r]++; // to account for new node in list 0
    if (r <= k) {
      w->next[r] = u->next[r];
      u->next[r] = w;
```



Slika 4.8: Brisanje elementa iz SkiplistList.

```
      w->length[r] = u->length[r] - (i - j);
      u->length[r] = i - j;
    }
    r--;
  }
  n++;
  return u;
}
```

Do sedaj bi morala biti implementacija operacije remove(i) v SkiplistList jasna. Sledimo iskalni poti vozlišča na poziciji i. Vsakič ko se iskalna pot zmanjša za ena od vozlišča, u, na nivoju r zmanjšamo dolžino nivoja, ki izstopa iz u-ja na tistem nivoju. Pregledati moramo tudi, da je u.next[r] element ranga i in v kolikor drži, ga premaknemo iz seznama na tisti nivo. Primer si lahko ogledate tukaj 4.8.

SkiplistList

```
T remove(int i) {
  T x = null;
  Node *u = sentinel, *del;
  int r = h;
  int j = -1; // index of node u
  while (r >= 0) {
    while (u->next[r] != NULL && j + u->length[r] < i) {
      j += u->length[r];
      u = u->next[r];
    }
    u->length[r]--; // for the node we are removing
    if (j + u->length[r] + 1 == i && u->next[r] != NULL) {
      x = u->next[r]->x;
```

```
      u->length[r] += u->next[r]->length[r];
      del = u->next[r];
      u->next[r] = u->next[r]->next[r];
      if (u == sentinel && u->next[r] == NULL)
        h--;
    }
    r--;
  }
  deleteNode(del);
  n--;
  return x;
}
```

4.3.1 Povzetek

Naslednji teorem povzema učinkovitost podatkovne strukture `SkiplistList`:

Izrek 4.2. *`SkiplistList` izvede vmesnik `List`. `SkiplistList` podpira operacije* `get(i)`, `set(i,x)`, `add(i,x)`, *ter* `remove(i)` *v* $O(\log n)$ *pričakovanem času na operacijo.*

4.4 Analiza preskočnega seznama

V sledečem delu bomo analizirali pričakovano višino, velikost ter dolžino Iskalne poti v preskočnem seznamu. Za razumevanje potrebujemo osnovno ozadnje verjetnosti. Nekateri dokazi so osnovani na metu kovanca.

Lema 4.2. *Naj bo T število, kadar se pošten kovanec obrne navzgor, vključno s primerom kadar kovanec pade z glavo navzgor. Takrat* $\mathrm{E}[T] = 2$.

Dokaz. Recimo da nehamo metati kovanec prvič kadar pade z glavo navzgor. Definirajmo indikacijsko spremenljivko

$$I_i = \begin{cases} 0 & \text{če je kovanec vržen navzgor } i \text{ kar} \\ 1 & \text{če je kovanec vržen } i \text{ ali več krat} \end{cases}$$

Upoštevajte da $I_i = 1$ če in samo če edini $i-1$ met kovanca postane rep, torej $\mathrm{E}[I_i] = \Pr\{I_i = 1\} = 1/2^{i-1}$. Opazimo da T, vse mete kovanca lahko

zapišemo kot $T = \sum_{i=1}^{\infty} I_i$. Sledi,

$$\begin{aligned}
\mathrm{E}[T] &= \mathrm{E}\left[\sum_{i=1}^{\infty} I_i\right] \\
&= \sum_{i=1}^{\infty} \mathrm{E}[I_i] \\
&= \sum_{i=1}^{\infty} 1/2^{i-1} \\
&= 1 + 1/2 + 1/4 + 1/8 + \cdots \\
&= 2 \ .
\end{aligned}$$

□

Naslednji hipotezi nam pokažeta da ima preskočni seznam linearno velikost:

Lema 4.3. *Pričakovano število vozlišč v preskočnem seznamu vsebuje* n *elementov, če ne upoštevamo kontrolnih pojavljanj, je* $2n$.

Dokaz. Verjetnost, da je kateri koli element, x, vsebovan v seznamu L_r is $1/2^r$, so the expected number of nodes in L_r je $n/2^r$.[2] Sledi, da je skupno število pričakovanih vozlišč v seznamu

$$\sum_{r=0}^{\infty} n/2^r = n(1 + 1/2 + 1/4 + 1/8 + \cdots) = 2n \ .$$

□

Lema 4.4. *Pričakovana višina preskočnega seznama, ki vsebuje* n *elementov je največ* $\log n + 2$.

Dokaz. Za vsak $r \in \{1, 2, 3, \ldots, \infty\}$, Definiramo indicator naključnih spremenljivk

$$I_r = \begin{cases} 0 & \text{if } L_r \text{ je prazen} \\ 1 & \text{if } L_r \text{ ni prazen} \end{cases}$$

Višina, h, preskočnega seznama je

$$h = \sum_{i=1}^{\infty} I_r \ .$$

[2] Poglej 1.3.4 za obrazložitev kako pridemo do rezultata z uporabo indikatorja spremenljivk in linearnosti pričakovanja.

Upoštevajte, da I_r ni nikoli večji kot dolžina, $|L_r|$, od L_r, zato

$$\mathrm{E}[I_r] \leq \mathrm{E}[|L_r|] = n/2^r \ .$$

Zato imamo

$$\begin{aligned}
\mathrm{E}[h] &= \mathrm{E}\left[\sum_{r=1}^{\infty} I_r\right] \\
&= \sum_{r=1}^{\infty} E[I_r] \\
&= \sum_{r=1}^{\lfloor \log n \rfloor} E[I_r] + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} E[I_r] \\
&\leq \sum_{r=1}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} n/2^r \\
&\leq \log n + \sum_{r=0}^{\infty} 1/2^r \\
&= \log n + 2 \ . \qquad \square
\end{aligned}$$

Lema 4.5. *Pričakovano število vozlišč v preskočnem seznamu vsebuje n elementov, z vsemi pojavitvami "opazovalca", je $2n + O(\log n)$.*

Dokaz. Po 4.3, sledi da je pričakovano število vozlišč, brez "opazovalca" $2n$. Število pojavitev "opaovalca" je enako višini, h, preskočnega seznama,torej 4.4 the expected number of occurrences of the je "opazovalec" največ $\log n + 2 = O(\log n)$. $\square$

Lema 4.6. *Pričakovana dolžina iskalne poti v preskočnem seznamu je največ $2 \log n + O(1)$.*

Dokaz. Najlažje dokažemo hipotezo tako da uporabimo *reverse search path* za vozlišče, x. Ta pot začne pri predhodniku x v L_0. Kadarkoli , če grelahko pot eno nadstropje višje takrat lahko. V kolikor nemore iti eno nadstropje višje, gre levo. Če nekaj trenutkov premišljujemo o tem nas bo prepričalo da je vzvratna iskalna pot za x enaka iskalni poti za x, z razliko da je vzvratna.

Število vozlišč, ki obiščejo vzvratno pot v nekem nadstropju, r, je povezana z naslednjim eksperimentom: Vržimo kovanec. Če pade glava,se premakni navzgor, nato ustavi. V nasprotnem primeru se premakni levo in ponovi eksperiment. Številov metov kovanca, preden pade glava predstavlja število korakov v levo, ki ki jih vzvratna iskalna pot porabi v nekem nadstropju.

Bodite pozorni da lahko pride do "overcounta" števila korakov na levo, saj se mora eksperiment končati. Končati mora ob prvi glavi ali ko iskalna pot doseže "opazovalca",kateri pride prvi. To ne predstavlja problema saj leži hipoteza na zgornji meji. 4.2 nam prikazuje, da je pričakovano število metov kovanca preden pade prva "glava", 1.

Naj S_r označuje število korakov ki jih porabi iskalna pot naprej na nadstropju r ki gre levo. Pravkar smo trdili da $E[S_r] \leq 1$. Poleg tega, $S_r \leq |L_r|$, ker nemoremo narediti več korakov v L_r kot je dolžina L_r, zato

$$E[S_r] \leq E[|L_r|] = n/2^r \ .$$

Sedaj lahko dokončamo dokaz 4.4. Naj bo S dolžina iskalne poti nekega vozlišča, u, v preskočnem seznamu in naj bo h višina preskočnega seznama. Sledi

$$\begin{aligned}
E[S] &= E\left[h + \sum_{r=0}^{\infty} S_r\right] \\
&= E[h] + \sum_{r=0}^{\infty} E[S_r] \\
&= E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} E[S_r] + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} E[S_r] \\
&\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=\lfloor \log n \rfloor + 1}^{\infty} n/2^r \\
&\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r \\
&\leq E[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r \\
&\leq E[h] + \log n + 3
\end{aligned}$$

$$\leq 2\log n + 5 \ . \qquad \square$$

Sledeči teorem povzema rezultat sekcije:

Izrek 4.3. *Preskočni senam, ki vsebuje* n *elementov je pričakoval velikost* $O(n)$ *in pričakovana dolžina iskalne poti nekega elementa je največ:* $2\log n + O(1)$.

4.5 Razprava in vaje

Preskočne sezname je predstavil Pugh [**?**] ki je tudi predstavil veliko aplikacij in razširitev preskočnih seznamov [**?**]. Od takrat se jih je veliko preučevalo. Veliko raziskovalcev je naredilo veliko natančnih analiz pričakovane dolžine in variance dolžine iskanja poti za `i`ti element v preskočnem seznamu [**?**, **?**, **?**]. Deterministične različice [**?**], pristranske različice [**?**, **?**], in samo-prilagodljive različice [**?**] preskočnih seznamov so se razvile. Implementacije preskočnih seznamov so bile napisane za različne jezike in ogrodja in so uporabljeni v odprtokodnih podatkovnih sistemih [**?**, **?**]. Različica preskočnih seznamov je uporabljena v strukturah upravljanja procesov jedra operacijskega sistema HP-UX [**?**].

Naloga 4.1. Narišite iskalne poti za 2.5 in 5.5 v preskočnem seznamu v 4.1.

Naloga 4.2. Narišite dodajanje vrednosti 0.5 (z višino 1)in nato 3.5 (z višino 2) v preskočni seznam v 4.1.

Naloga 4.3. Narišite odstranjevanje vrednosti 1 in nato 3 iz preskočnega seznama v 4.1.

Naloga 4.4. Narišite izvedbo `remove`(2) v `SkiplistList` v 4.5.

Naloga 4.5. Narišite izvedbo `add`(3, `x`) v `SkiplistList` v 4.5. Predpostavi, da `pickHeight`() izbere višino 4 za novo ustvarjeno vozlišče.

Naloga 4.6. Pokažite da je med izvajanjem `add`(`x`) ali `remove`(`x`) operacij, pričakovano število kazalcev v `SkiplistSet` ki se spremenijo konstanta.

Naloga 4.7. Predpostavite da, namesto povišanja elementa iz L_{i-1} v L_i na osnovi meta kovanca, element povišamo z neko verjetnostjo p, $0 < p < 1$.

1. Pokažite, da je s to modifikacijo pričakovana dolžina iskalne poti največ $(1/p)\log_{1/p} \mathtt{n} + O(1)$.

2. Kakšna je vrednost p ki zmanjša prejšnji izraz?

3. Kakšna je pričakovana višina preskočnega seznama?

4. Kakšno je pričakovano število vozlišč v preskočnem seznamu?

Naloga 4.8. Metoda `find(x)` v `SkiplistSet` včasih izvede *odvečne primerjave*; Te se pojavijo kadar je `x` primerjan z isto vrednostjo več kot enkrat. Pojavijo se lahko za neko vozlišče, `u`, `u.next[r]` = `u.next[r-1]`. Pokažite kako se te odvečne primerjave zgodijo in priredite `find(x)` tako da se jih izognete. Analizirajte pričakovano število primerjav izvedenih z vašo prirejeno `find(x)` metodo.

Naloga 4.9. Zasnujte in implementirajte različico preskočnega seznama, ki implementira `SSet` interface, pa tudi dovoljuje hiter dostop do elementov po rangu. To pomeni, da tudi podpira funkcijo `get(i)`, ki vrača element katerega rang je `i` v $O(\log \mathtt{n})$ pričakovani časovni zahtevnosti. (Rang elementa `x` v `SSet` je število elementov v `SSet` ki so manjši od `x`.)

Naloga 4.10. *prst* v preskočnem seznamu je polje ki shranjuje zaporedje vozlišč v iskalni poti kjer se iskalna pot spušča. (Spremenljivka `stack` v `add(x)` koda na strani 93 je prst; osenčena vozlišča v 4.3 kažejo na vsebino enega prsta.) Na prst lahko gledamo kot na nekaj kar kaže pot do vozlišča v najnižjem seznamu, L_0.

finger search implementira `find(x)` operacijo z uporabo prsta, s sprehajanjem po seznamu navzgor z uporabo prsta dokler ne doseže vozlišča `u` tako da je `u.x` < `x` in `u.next` = `null` ali `u.next.x` > `x` in nato izvajanjem noramlnega iskanja `x` začenši z `u`. Mogoče je dokazati da je pričakovano število potrebnih korakov za finger search $O(1 + \log r)$, kjer je r število vrednosti v L_0 med `x` in vrednostjo na katero kaže prst.

Implementirajte podrazred od `Skiplist`, ki se imenuje `SkiplistWithFinger`, ki implementira `find(x)` operacije z uporabo notranjega prsta. Podrazred naj hrani prst, ki je uporabljen tako da je vsaka operacija `find(x)` implementirana kot prstno iskanje (finger search). Med vsako `find(x)` operacijo je prst posodobljen tako da vsaka operaacija `find(x)`

uporabi, kot začtno točko, prst ki kaže na rezultat prejšnje `find(x)` operacije.

Naloga 4.11. Zapišite metodo `truncate(i)`, ki skrajša `SkiplistList` na poziciji `i`. Po izvedbi metode, je velikost seznama `i` in vsebuje samo elemente na indexih $0,\ldots,i-1$. Vrnjena vrednost je nek drug `SkiplistList`, ki vsebuje elemente na indexih $i,\ldots,n-1$. Metoda mora imeti časovno zahtevnost $O(\log n)$.

Naloga 4.12. Napišite `SkiplistList` metodo, `absorb(l2)`, ki sprejme argument `SkiplistList`, `l2`, ga izprazni in pripne njegovo vsebino, , urejeno, prejemniku. Naprimer, če `l1` vsebuje a,b,c in `l2` vsebuje d,e,f, potem bo po klicu `l1.absorb(l2)`, `l1` vseboval a,b,c,d,e,f in `l2` bo prazen. Metoda naj ima časovno zahtevnost $O(\log n)$.

Naloga 4.13. Z uporabo pristopov prostorsko učinkovitega seznama `SEList`, zasnujte in implementirajte prostorsko učinkovit `SSet`, `SESSet`. Da bi to stroili, shranite urejene podatke v `SEList`, in bloke tega `SEList` v `SSet`. Če prvotna implementacija `SSet` porabi $O(n)$ prosotra za shranjevanje n elementov, potem bo `SESSet` imel dovolj prostora za n elementov plus $O(n/b+b)$ odvečnega prostora.

Naloga 4.14. Z uporabo `SSet` kot vašo osnovno strukturo, zasnujte in implementirajte aplikacijo, ki prebere (veliko) besedilno datoteko in dovoljuje interaktivno iskanje, za katerikoli podniz vsebovan v besedilu. Ko uporabnik vnaša svojo iskalno zahtevo naj se kot rezultat prikazuje ujemajoč del besedila (če obstaja).
Namig 1: Vsak podniz je predpona neki priponi, tako da zadošča shraniti vse pripone besedilne datoteke.
Namig 2: Vsaka pripona je lahko predstavljena strnjeno kot samojstojna števka, ki predstavlja kje v besedilu se pripona začne.
Preizkusite svojo aplikacijo na veleikih besedilih, kot so na primer knjige dostopne na Project Gutenberg [**?**]. If done correctly, your applications will be very responsive; there should be no noticeable lag between typing keystrokes and seeing the results.

Naloga 4.15. (Ta vaja naj bo opravljena po branju o binarnih iskalnih drevesih.) in 6.2.) Primerjajte preskočne sezname z binarnimi iskalnimi drevesi po naslednjih kriterijih:

1. Razložite kako odstranjevanje robnih elementov preskočnega sezname vodi k strukturi ki izgleda kot binarno drevo in je enaka binarnemu iskalnemu drevesu.

2. Preskočni seznami in dvojiška iskalna drevesa oboji porabijo približno enako število kazalcev (2 na vozlišče). Preskočni seznami bolje uporabijo te kazalce. Razložite zakaj.

Poglavje 5

Zgoščevalne tabele

Zgoščevalne tabele predstavljajo učinkovito metodo za shranjevanje majhnega števila celih števil n, iz velikega obsega $U = \{0, \ldots, 2^w - 1\}$. Izraz *zgoščevalna tabela* sicer označuje širok spekter podatkovnih struktur. Prvi del poglavja se osredotoča na dve najbolj pogosti implementaciji: zgoščevanje z veriženjem in linearno naslavljanje.

Zelo pogosto se uporabljajo za shranjevanje podatkov, katerih tip niso cela števila. V tem primeru je celoštevilska *zgoščevalna koda* povezana z vsako podatkovno enoto in uporabljena v zgoščevalni tabeli. Drugi del predstavi, kako so zgoščevalne kode ustvarjene.

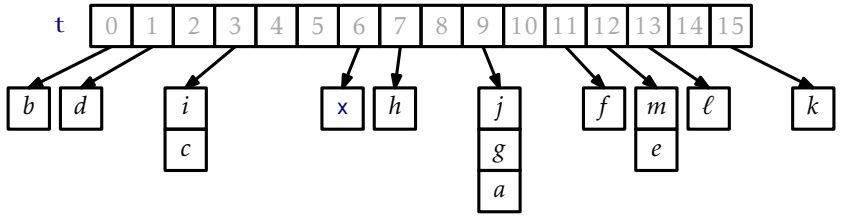
Nekatere uporabljene metode iz tega poglavja potrebujejo naključno izbrana števila v določenem razponu. V primerih kode, so nekatera "naključna" cela števila enolično določena z uporabo naključnih bitov generiranih iz atmosferskega šuma.

5.1 Zgoščevalna tabela z veriženjem

Podatkovna struktura zgoščevalna tabela z veriženjem za shranjevanje tabele t seznamov uporablja zgoščevanje z veriženjem. Za hranjenje skupnega števila podatkov v vseh seznamih se uporablja celo število n. (glej 5.1):

ChainedHashTable

```
array<List> t;
int n;
```



Slika 5.1: Primer zgoščevalne tabele z veriženjem z `n` = 14 in `t.length` = 16. V tem primeru je `hash(x)` = 6

Zgoščena vrednost podatkovnega elementa `x`, označena z `hash(x)` predstavlja vrednost v razponu $\{0,\ldots,$ `t.length` $-1\}$. Vsi podatki z zgoščeno vrednostjo `i` so shranjeni v seznamu na lokaciji `t[i]`. Da se izognemo prevelikim seznamom, ohranjamo invarianto

$$\texttt{n} \leq \texttt{t.length}$$

tako da je povprečno število elementov shranjenih v posameznem seznamu `n/t.length` ≤ 1.

Pri dodajanju elementa `x`, v zgoščevalno tabelo, najprej preverimo če je potrebno povečati `t.length`. V kolikor je to potrebno ga povečamo. Potem razpršimo `x`, da dobimo število `i`, v razponu $\{0,\ldots,$ `t.length` $-1\}$, in pripnemo `x` seznamu `t[i]`:

ChainedHashTable

```
bool add(T x) {
  if (find(x) != null) return false;
  if (n+1 > t.length) resize();
  t[hash(x)].add(x);
  n++;
  return true;
}
```

Povečevanje tabele, v kolikor je le-to potrebno, vključuje podvojitev dolžine tabele `t` in ponovno vstavljanje elementov vanjo. Ta strategija je popolnoma enaka kot pri implementaciji ArrayStacka in tudi tu velja enako pravilo: Cena rasti je amortizirano po nekaj sekvencah vstavljanja samo konstantna (glej 2.1 na strani 34).

Poleg rasti je edino potrebno opravilo ob vstavljanju nove vrednosti `x` v zgoščevalno tabelo z veriženjem dodajanje `x`-a seznamu `t[hash(x)]`. Za katerokoli od implementacij seznama opisanih v poglavjih 2 in 3, potrebujemo le konstanten čas.

Za odstranitev elementa `x` iz zgoščevalne tabele se sprehodimo čez seznam `t[hash(x)]`, dokler n najdemo elementa `x`, tako da ga lahko odstranimo:

ChainedHashTable

```
T remove(T x) {
  int j = hash(x);
  for (int i = 0; i < t[j].size(); i++) {
    T y = t[j].get(i);
    if (x == y) {
      t[j].remove(i);
      n--;
      return y;
    }
  }
  return null;
}
```

Časovna zahtevnost je $O(n_{\texttt{hash(x)}})$, pri čemer n_i označuje dolžino seznama shranjenega v `t[i]`.

Iskanje elementa `x` v zgoščevalni tabeli poteka podobno. Izvedemo linearno iskanje nad seznamom `t[hash(x)]`:

ChainedHashTable

```
T find(T x) {
  int j = hash(x);
  for (int i = 0; i < t[j].size(); i++)
    if (x == t[j].get(i))
      return t[j].get(i);
  return null;
}
```

Podobno tudi tu potrebujemo čas sorezmeren z dolžino seznama `t[hash(x)]`.

Hitrosti zgoščevalnih tabel so odvisne predvsem od izbire zgoščevalne funkcije. Dobra zgoščevalna funkcija razprši elemente enakomerno med `t.length` seznamov, tako da je pričakovana velikost seznama `t[hash(x)]` $O(n/t.length) = O(1)$. Po drugi strani pa slaba zgoščevalna funkcija zgosti vse vrednosti(vključno z `x`) na isto lokacijo v tabeli. V tem primeru bo

2^w (4294967296)	10000000000000000000000000000000
z (4102541685)	11110100100001111101000101110101
x (42)	00000000000000000000000000101010
$z \cdot x$	10100000011110010010000101110100110010
$(z \cdot x) \bmod 2^w$	00011110010010000101110100110010
$((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d}$	00011110

Slika 5.2: Operacija večkratne zgoščevalne funkcije z $w = 32$ in $d = 8$.

velikost seznama $t[\mathrm{hash}(x)]$ n. V naslednjem poglavju je opisan primer dobre zgoščevalne funckije.

5.1.1 Zgoščevanje z množenjem

Zgoščevanje z množenjem je učinkovita metoda tvorbe zgoščevalnih vrednosti osnovana na kongruenci(opisana v poglavju 2.3) in celoštevilskemu deljenju. Uporablja operator div , ki obdrži celoštevilski del kvocienta, ostanek pa zanemari. Praktično za vsako število velja $a \geq 0$ in $b \geq 1$, $a \operatorname{div} b = \lfloor a/b \rfloor$.

Pri zgoščevanju z množenjem uporabljamo tabele velikosti 2^d pri čemer je d neko celo število(imenovano *dimenzija*). Formula za zgoščevanje celega števila $x \in \{0, \ldots, 2^w - 1\}$ je

$$\mathrm{hash}(x) = ((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d} \ .$$

Pri tem je z neko naključno izbrano *celo* število v $\{1, \ldots, 2^w - 1\}$. Zgoščevalna funkcija je lahko realizirana zelo učinkovito, z obzirom na to, da so operacije nad celimi števili že v osnovi izvedene nad 2^w biti, kjer je w število bitov v celem številu. (Glej 5.2.) Poleg tega je celoštevilsko deljenje z 2^{w-d} enako izločanju skrajno desnih $w-d$ bitov v binarni predstavitvi (kar uredimo s premikom za $w-d$ bitov). S tem dosežemo, da ima koda lažjo implementacijo kot matematična formula:

ChainedHashTable

```
int hash(T x) {
  return ((unsigned)(z * hashCode(x))) >> (w-d);
}
```

Pri naslednjem primeru, čigar dokaz je prikazan kasneje v poglavju,

pokažemo, da igra zgoščevalna funkcija z množenjem odlično vlogo pri izmikanju trkov.

Lema 5.1. *Naj bosta* x *in* y *dve vrednosti izmed* $\{0,\ldots,2^w-1\}$ *in* $x \neq y$. *Potem sledi, da* $\Pr\{\texttt{hash}(x) = \texttt{hash}(y)\} \leq 2/2^d$.

Pri primeru 5.1, je učinkovitost funkcij `odstrani(x)` in na `jdi(x)` možno preprosto analizirati:

Lema 5.2. *Za katerokoli podatkovno vrednost* x *je pričakovana dolžina seznama* $t[\texttt{hash}(x)]$ *največ* n_x+2, *pri čemer je* n_x *število pojavitev* x *v zgoščevalni tabeli.*

Dokaz. Naj bo S (večkratna-) zbirka elementov shranjenih v zgoščevalni tabeli, ki ni enaka x. Za element $y \in S$ definiramo indikatorsko spremenljivko

$$I_y = \begin{cases} 1 & \text{če je } \texttt{hash}(x) = \texttt{hash}(y) \\ 0 & \text{drugače} \end{cases}$$

in opazimo, da je po primeru 5.1, $\mathrm{E}[I_y] \leq 2/2^d = 2/\texttt{t.length}$ pričakovana dolžina lista $t[\texttt{hash}(x)]$ podana v naslednji obliki

$$\begin{aligned} \mathrm{E}[t[\texttt{hash}(x)].\texttt{size}()] &= \mathrm{E}\left[n_x + \sum_{y \in S} I_y\right] \\ &= n_x + \sum_{y \in S} \mathrm{E}[I_y] \\ &\leq n_x + \sum_{y \in S} 2/\texttt{t.length} \\ &\leq n_x + \sum_{y \in S} 2/n \\ &\leq n_x + (n - n_x)2/n \\ &\leq n_x + 2 \ , \end{aligned}$$

□

Sedaj bi želeli dokazati primer 5.1, a za slednje, najprej potrebujemo rezultat iz teorije števil. Pri naslednjem dokazu uporabljamo notacijo $(b_r,\ldots,b_0)_2$ pri označevanju $\sum_{i=0}^{r} b_i 2^i$, kjer je vsak b_i bitna vrednost, ali 0 ali 1. Z drugimi besedami je $(b_r,\ldots,b_0)_2$ celo številčna vrednost, čigar

dvojiška predstavitev je podana kot $b_r, \ldots, b_0$. Z uporabo $\star$ označimo neznano bitno vrednost.

Lema 5.3. *Naj bo S zbirka lihih celih števil na intervalu $\{1, \ldots, 2^w - 1\}$; prav tako naj bosta q in i dva, katera koli, elementa izmed vseh elementov v S. Potem takem obstaja točno ena vrednost $z \in S$ za katero velja $zq \bmod 2^w = i$.*

Dokaz. Ker je število izbira za z in i enaka, je zadostljivo dokazati, *največ* ena vrednost $z \in S$ za katero velja $zq \bmod 2^w = i$.

Predpostavimo da sta, za voljo nasprotij, dve vrednosti z and z', kjer velja $z > z'$. Potem je

$$zq \bmod 2^w = z'q \bmod 2^w = i$$

Kar sledi k

$$(z - z')q \bmod 2^w = 0$$

Slednje pomeni, da je

$$(z - z')q = k2^w \tag{5.1}$$

za neko celo število k. V smislu dvojiških števil, bi slednje pomenilo da imamo

$$(z - z')q = k \cdot (1, \underbrace{0, \ldots, 0}_{w})_2 \ ,$$

tako da so w zadnje bitne vrednosti v dvojiški predstavitvi $(z - z')q$ vse ničle (0).

Poleg tega velja tudi da je $k \neq 0$, ker velja da je $q \neq 0$ in $z - z' \neq 0$. Ker je q liho število, nima ničel kot zadnje vrednosti v bitni predstavitvi:

$$q = (\star, \ldots, \star, 1)_2 \ .$$

Ker velja da ima $|z - z'| < 2^w$, $z - z'$ manj, kot w, ničelnih zadnjih vrednosti v bitni predstavitvi slednjega:

$$z - z' = (\star, \ldots, \star, 1, \underbrace{0, \ldots, 0}_{<w})_2 \ .$$

□

Uporabnost 5.3 izhaja iz sledeče predpostavke: Če je z izbran enakomerno naključno iz S, potem je zt enakomerno porazdeljen nad S. V sledečem dokazu, si pomagamo z dvojiško predstavitvijo z, katera sestoji iz $w-1$ naključnih bitov s pripono 1.

Dokaz za 5.1. Začnemo z ugotovitvijo da je $\mathtt{hash}(x) = \mathtt{hash}(y)$ ekvivalenten trditvi " d najpomembnejših bitov v $zx \bmod 2^w$ in d najpomembnejših bitov $zy \bmod 2^w$ je enakih." Pri prejšnji trditvi je potrebno poudariti, da je d najpomembnejših bitov v dvojiški predstavitvi $z(x-y) \bmod 2^w$ vseh enakih 1 ali enakih 0. Torej velja,

$$z(x-y) \bmod 2^w = (\underbrace{0,\ldots,0}_{d},\underbrace{\star,\ldots,\star}_{w-d})_2 \tag{5.2}$$

ko velja $zx \bmod 2^w > zy \bmod 2^w$ ali

$$z(x-y) \bmod 2^w = (\underbrace{1,\ldots,1}_{d},\underbrace{\star,\ldots,\star}_{w-d})_2 \ . \tag{5.3}$$

ko velja $zx \bmod 2^w < zy \bmod 2^w$. Potemtakem, ugotavljamo le verjetnost, da $z(x-y) \bmod 2^w$ izgleda kot (5.2) or (5.3).

Naj bo q enolično liho število, za katero velja $(x-y) \bmod 2^w = q2^r$ za neko število $r \geq 0$. Po 5.3, ima dvojiška predstavitev $zq \bmod 2^w$ $w-1$ naključnih bitov, zaključenih z 1:

$$zq \bmod 2^w = (\underbrace{b_{w-1},\ldots,b_1}_{w-1},1)_2$$

Iz tega sledi, da ima dvojiška predstavitev $z(x-y) \bmod 2^w = zq2^r \bmod 2^w$ $w-r-1$ naključnih bitov, zaključenih z 1, zaključenih z r ponovitvami 0:

$$z(x-y) \bmod 2^w = zq2^r \bmod 2^w = (\underbrace{b_{w-r-1},\ldots,b_1}_{w-r-1},1,\underbrace{0,0,\ldots,0}_{r})_2$$

S tem zaključimo dokaz. Če je $r > w-d$, potem d najpomembnejših bitov $z(x-y) \bmod 2^w$ vsebuje tako ničle kot enice , tako da je verjetnost da $z(x-y) \bmod 2^w$ izgleda kot (5.2) ali (5.3) nična. Če je $r = w-d$, potem je verjetnost da izgleda kot (5.2) nična, vendar je verjetnost da izgleda kot (5.3) $1/2^{d-1} = 2/2^d$ (ker moramo imeti $b_1,\ldots,b_{d-1} = 1,\ldots,1$).

Če velja $r < w - d$, potem moramo imeti $b_{w-r-1},\ldots,b_{w-r-d} = 0,\ldots,0$ ali $b_{w-r-1},\ldots,b_{w-r-d} = 1,\ldots,1$. Verjetnost posamezne od teh možnosti je $1/2^d$ pri čemer so vse vzajemno izključujoče, tako da je verjetnost da se zgodi katerakoli $2/2^d$. S tem zaključimo dokaz. □

5.1.2 Povzetek

Naslednji izrek povzema uspešnost `ChainedHashTable` podatkovne strukture:

Izrek 5.1. *`ChainedHashTable` implementira vmesnik `USet`. Če ignoriramo ceno klicev metode* `grow()`, *`ChainedHashTable` podpira operacije* `add(x)`, `remove(x)`, `find(x)`, *v pričakovanem $O(1)$ času na operacijo.*

Poleg tega, da je začetna `ChainedHashTable` prazna, vsaka sekvenca od m `add(x)` *in* `remove(x)` *operacije rezultira v skupni porabi $O(m)$ časa za vse klice na* `grow()`.

5.2 `LinearHashTable`: Linearno naslavljanje

Podatkovna struktura `ChainedHashTable` uporablja polje seznamov, kjer `i` seznam shrani vse elemente `x` tako da je `hash(x) = i`. Alternativa po imenu *odprto nasljavljanje* je namenjena shranjevanju elementov neposredno v polje, `t`, z vsako lokacijo polja v `t` pa shrani največ eno vrednost. Tak pristop se uporablja v `LinearHashTable` in je opisan v tem poglavju. Ponekod je ta podatkovna struktura opisana kot *odprto naslavljanje*.

Glavna ideja `LinearHashTable` je da bi mi lahko, idealno, shranili element `x` z razpršilno vrednostjo `i = hash(x)` v lokacijo tabele `t[i]`. Če tega ne moremo storiti (ker je nek element že shranjen tam) potem ga skušamo shraniti v lokaciji `t[(i + 1) mod t.length]`; če tudi to ni mogoče, potem poskusimo z `t[(i + 2) mod t.length]`, in tako naprej, dokler ne najdemo mesta za `x`.

V `t` imamo shranjene tri tipe vhodov:

1. podatkovne vrednosti: dejanske vrednosti iz `USet` katere predstavljamo;

2. `null` vrednosti: na lokacijah v tabeli kjer ni in ni bilo nikoli kakršnihkoli podatkov; in

3. `del` vrednosti: na lokacijah tabele kjer so bili podatki nekoč shranjeni ampak so od takrat bili izbrisani.

Poleg števca, `n`, ki skrbi za spremljanje številov elementov v `LinearHashTable`, imamo še števec, `q`, ki skrbi za spremljanje števila elementov Tipov 1 in 3. To pomeni, `q` je enak `n` z dodanimi števili `del` vrednosti v `t`. Za učinkovito delovanje potrebujemo da je `t` precej večji od `q`, tako da je veliko `null` vrednosti v `t`. Operacije na `LinearHashTable` torej ohranjajo invarianto, da je $t.length \geq 2q$.

Torej, `LinearHashTable` hrani tabelo, `t`, ki hrana podatkovne elemente in cela števila ali integers `n` in `q` ki spremljata število dejanski podatkovnih elementov in ne-`null` vrednosti v `t`. Ker vrsta zgoščevalnih funkcij deluje le za tabele katerih velikosti potence števila 2, prav tako hranimo celo število `d` in ohranjamo invarianto da je $t.length = 2^d$.

LinearHashTable
```
array<T> t;
int n;   // number of values in T
int q;   // number of non-null entries in T
int d;   // t.length = 2^d
```

Delovanje iskanja `find(x)` je v `LinearHashTable` preprosto. Začnemo z vpisom v tabelo `t[i]` kjer je `i = hash(x)` in iskanih elementov `t[i]`, `t[(i + 1) mod t.length]` , `t[(i + 2) mod t.length]`, in tako naprej dokler ne najedmo indeksa `i'` tako, da je bodisi `t[i'] = x`, ali `t[i'] = null`. V prvem primeru bomo vrnili `t[i']`. V drugem primeru pa lahko ugotovimo, da x ni vsebovan v razpršeni tabeli in vrnemo `null`.

LinearHashTable
```
T find(T x) {
  int i = hash(x);
  while (t[i] != null) {
    if (t[i] != del && t[i] == x) return t[i];
    i = (i == t.length-1) ? 0 : i + 1; // increment i
  }
  return null;
}
```

Delovanje add(x) je tudi dokaj enostavno izvajati. Po preverjanju, da x slučajno že ni shranjena v tabeli (uporabimo find(x)), iščemo t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], in tako naprej, dokler ne najdemo null ali del in shranimo x na lokaciji, če je potrebno povečamo n in q.

LinearHashTable

```
bool add(T x) {
  if (find(x) != null) return false;
  if (2*(q+1) > t.length) resize();   // max 50% occupancy
  int i = hash(x);
  while (t[i] != null && t[i] != del)
    i = (i == t.length-1) ? 0 : i + 1; // increment i
  if (t[i] == null) q++;
  n++;
  t[i] = x;
  return true;
}
```

Do sedaj naj bi bilo delovanje izvajanja remove(x) očitno. Iščemo t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], in tako naprej dokler ne najdemo indeksa i′ tako, da bo t[i′] = x ali t[i′] = null. V prvem primeru nastavimo t[i′] = del in vrnemo true. V drugem primeru ugotovimo, da x wni bil shranjen v tabeli (zato ga ne moremo odstraniti) in vrnemo false.

LinearHashTable

```
T remove(T x) {
  int i = hash(x);
  while (t[i] != null) {
    T y = t[i];
    if (y != del && x == y) {
      t[i] = del;
      n--;
      if (8*n < t.length) resize(); // min 12.5% occupancy
      return y;
    }
    i = (i == t.length-1) ? 0 : i + 1;  // increment i
  }
  return null;
}
```

Pravilnost metod `find(x)`, `add(x)` in `remove(x)` je lahko preveriti, čeprav temelji na uporabi `del` vrednosti. Opazimo lahko, da nobena od teh operacij nikoli ne postavi ne-`null` vnosa na `null`. Zato ko dosežemo indeks `i`′, kot je recimo `t[i]` = `null`, je to dokaz da element `x`, ki ga iščemo, ni shranjen v tabeli; `t[i]` je bil vedno `null`, zato ni razloga da bi prejšnja operacija `add(x)` nadaljevala čez indeks `i`′.

Metodo resize() pokliče metoda `add(x)` ko število ne-`null` vnosov preseže `t.length`/2 ali pa metoda `remove(x)`, ko je število podatkovnih vnosov manjše od `t.length`/8. Metoda deluje enako kot v drugih podatkovih strukturah, ki temeljijo na tabelah. Najdemo najmanjše pozitivno število `d`, tako da je $2^d \geq 3n$. Tabelo `t` dodelimo tako da dobimo tabelo velikosti 2^d in nato vse elemente iz stare verzije tabele `t` vstavimo v novo ustvarjeno kopijo tabele `t`. Medtem ponastavimo `q` na vrednost `n`, saj nova tabela `t` ne vsebuje `del` vrednosti.

LinearHashTable

```
void resize() {
  d = 1;
  while ((1<<d) < 3*n) d++;
  array<T> tnew(1<<d, null);
  q = n;
  // insert everything into tnew
  for (int k = 0; k < t.length; k++) {
    if (t[k] != null && t[k] != del) {
      int i = hash(t[k]);
      while (tnew[i] != null)
        i = (i == tnew.length-1) ? 0 : i + 1;
      tnew[i] = t[k];
    }
  }
  t = tnew;
}
```

5.2.1 Analiza odprtega naslavljanja

Vsaka od operacij `add(x)`, `remove(x)` in `find(x)` se konča najkasneje takoj ko odkrije prvi `null` vnos v `t`. Intuicija za to analizo temelji na tem, da je najmanj polovica elementov v tabeli `t` enakih `null`, zato operacija ne

bi smela potrebovati veliko časa za zaključitev, saj zelo hitro naleti na `null` vnos. Na to intuicijo se ne smemo preveč trdno zanašati, ker bi nas pripeljala do (napačnega) sklepa da je pričakovano število lokacij v tabeli `t`, ki jo poda ta operacija, največ 2.

Za preostanek tega poglavja bomo domnevali, da so vse razpršene vrednosti neodvisno in enotno porazdeljene v $\{0,\ldots,\texttt{t.length}-1\}$. To ni realistišna domneva, vendar nam bo omogočila analizo linearnega naslavljanja. Kasneje v tem poglavju bomo opisali metodo imenovano tabelarno zgoščevanje, ki ustvari razpršeno funkcijo, ki je "dovolj dobra" za linearno naslavljanje. Prav tako bomo predpostavili, da so vsi indeksi v položajih `t` celoštevilsko deljeni z `t.length`, tako da je `t[i]` okrajšava za `t[i mod t.length]`.

Pravimo da se *izvršitev dolžine k*, *ki se začne pri* `i` zgodi, kadar noben od elementov $\texttt{t}[\texttt{i}],\texttt{t}[\texttt{i}+1],\ldots,\texttt{t}[\texttt{i}+k-1]$ ni `null` in $\texttt{t}[\texttt{i}-1]=\texttt{t}[\texttt{i}+k]=$ `null`. Število elementov tabele `t` ki niso `null` je enako `q`, metoda `add(x)` pa zagotavlja, da vedno velja $\texttt{q}\le\texttt{t.length}/2$. Obstaja `q` elementov $\texttt{x}_1,\ldots,\texttt{x}_\texttt{q}$ ki so bili vstavljeni v `t` po zadnji `rebuild()` operaciji. Po naši domnevi ima vsak izmed teh elementov zgočevalno vrednost $\texttt{hash}(\texttt{x}_j)$, ki je enotna in neodvisna od drugih. S tako nastavitvijo lahko dokažemo glavno trditev potrebno za analiziranje linearnega naslvljanja.

Lema 5.4. *Določimo vrednost* $\texttt{i}\in\{0,\ldots,\texttt{t.length}-1\}$. *Potem je možnost, da se izvršitev dolžine k začne pri* `i`, *enaka* $O(c^k)$ *za konstanto* $0<c<1$.

Dokaz. Če se začetek dolžine k začne pri `i`, je natanko k elementov $\texttt{x}_j$, ki so $\texttt{hash}(\texttt{x}_j)\in\{\texttt{i},\ldots,\texttt{i}+k-1\}$. Verjetnost za to je točno

$$p_k=\binom{\texttt{q}}{k}\left(\frac{k}{\texttt{t.length}}\right)^k\left(\frac{\texttt{t.length}-k}{\texttt{t.length}}\right)^{\texttt{q}-k},$$

ker za vsako izbiro k elementov, teh k elementov mora zgostiti k eni izmed k lokacij. Preostalih $\texttt{q}-k$ pa mora zgostiti k preostalim $\texttt{t.length}-k$ lokacijam v tabeli.[1]

V naslednji izpeljavi bomo pogoljufali in zamenjali $r!$ z $(r/e)^r$. Stirlingova aproksimacija (1.3.2) nam pove da je to le faktor $O(\sqrt{r})$ od pravilnosti. To naredimo zato, da si poenostavimo izpeljavo; 5.4 od bralca

[1] Upoštevajte, da je p_k večje kot verjetnost, da se izvajanje dolžine k začne pri `i`, ker definicija od p_k ne upošteva pogoja $\texttt{t}[\texttt{i}-1]=\texttt{t}[\texttt{i}+k]=$ `null`.

zahteva, da natančneje in v celoti ponovi izračun z uporabo Stirlingove aproksimacije.

Vrednost p_k je maksimalna, ko je `t.length` minimum in podatkovna struktura obdrži nespremnjen `t.length` $\geq 2q$, torej

$$\begin{aligned}
p_k &\leq \binom{q}{k}\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\
&= \left(\frac{q!}{(q-k)!k!}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\
&\approx \left(\frac{q^q}{(q-k)^{q-k}k^k}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \qquad \text{[Stirlingova aproksimacija]} \\
&= \left(\frac{q^k q^{q-k}}{(q-k)^{q-k}k^k}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\
&= \left(\frac{qk}{2qk}\right)^k\left(\frac{q(2q-k)}{2q(q-k)}\right)^{q-k} \\
&= \left(\frac{1}{2}\right)^k\left(\frac{(2q-k)}{2(q-k)}\right)^{q-k} \\
&= \left(\frac{1}{2}\right)^k\left(1+\frac{k}{2(q-k)}\right)^{q-k} \\
&\leq \left(\frac{\sqrt{e}}{2}\right)^k .
\end{aligned}$$

(V zadnjem koraku uporabimo neenakost $(1+1/x)^x \leq e$, ki drži za vse $x > 0$.). Ker je $\sqrt{e}/2 < 0.824360636 < 1$, dokaz lahko potrdimo. □

Uporaba 5.4 za dokaz zgornje meje na času izvajanja `find(x)`, `add(x)` in `remove(x)` je sedaj enostavna. Upoštevajmo najenostavnješi primer, kjer izvršimo `find(x)` za neko vrednost `x`, ki ni bila nikoli shranjena v `LinearHashTable`. V tem primeru `i` = `hash(x)` dobi naključno vrednost v $\{0,\ldots,$ `t.length` $-1\}$, ki je neodvisna od vsebine `t`. Če je `i` del izvajanja dolžine k, potem je čas izvajanja operacije `find(x)` v najboljšem primeru $O(1+k)$. Potemtakem, zgornja meja pričakovanega časa izvajanja je

$$O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k\Pr\{\texttt{i} \text{ je del obhoda dolžine } k\}\right) .$$

Upoštevajte, da vsako izvajanje dolžine k prispeva k notranji vsoti k-krat za končni prispevek k^2, torej lahko navedeno vsoto ponovno napišemo kot

$$
\begin{aligned}
&O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty}k^2\Pr\{\texttt{i začne obhod dolžine } k\}\right)\\
&\leq O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty}k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty}k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty}k^2\cdot O(c^k)\right)\\
&= O(1)\ .
\end{aligned}
$$

Zadnji korak v tej izpeljavi prihaja iz dejstva, da $\sum_{k=0}^{\infty}k^2\cdot O(c^k)$ eksponentno zmanjšuje vrsto.[2] Potemtakem lahko sklepamo, da je pričakovan čas izvajanja operacije `find(x)` za vrednost `x`, ki ni vsebovana v `LinearHashTable` enaka, $O(1)$.

Če zanemarimo ceno operacije `resize()`, potem nam gornja analiza poda vse kar potrebujemo za analiziranje cene ostalih operacij v `LinearHashTable`.

Analiza gornje operacije `find(x)` velja pri operaciji `add(x)` kadar, `x` ni v tabeli. Za analizo operacije `find(x)` kadar, `x` je vsebovan v tabeli moramo upoštevati samo to, da je cena enaka operaciji `add(x)` s katero smo dodali `x` v tabelo. Za konec, cena operacije `remove(x)` je enaka ceni operacije `find(x)`.

V povzetku, če zanemarimo ceno klicev operacije `resize()`, so vse ostale operacije v `LinearHashTable` izvršene v pričakovanem času $O(1)$. Da upoštevamo ceno operacije resize, lahko uporabimo enako amortizirano analizo izvedeno za podatkovno strukturo `ArrayStack` v 2.1.

[2] V terminologiji več matematičnih učbenikov nam ta vsota poda razmerje: Obstaja pozitivno celo število k_0, ki velja za vse $k \geq k_0$, $\frac{(k+1)^2 c^{k+1}}{k^2 c^k} < 1$.

5.2.2 Povzetek

Spodnji izrek je povzetek časovnih zahtevnosti, metod, podatkovne strukture `LinearHashTable`:

Izrek 5.2. *`LinearHashTable` implementira vmesnik `USet`. Če ignoriramo ceno klicev metode* `resize()`*, je pričakovana časovna zahtevnost metod* `add(x)`*,* `remove(x)`*, in* `find(x)`*, podatkovne strukture `LinearHashTable`, enaka $O(1)$.*

Če začenjamo s prazno `LinearHashTable`, velja, da za katerokoli zaporedje m operacij metod `add(x)` *in* `remove(x)`*, porabimo $O(m)$ časa za klice metode* `resize()`*.*

5.2.3 Tabelarno zgoščevanje

Med analizo podatkove strukture `LinearHashTable`, smo naredili zelo močno predpostavko: Da so za katerokoli množico elementov, $\{x_1,\ldots,x_n\}$, zgoščevalne vrednosti $\texttt{hash}(x_1),\ldots,\texttt{hash}(x_n)$ neodvisno in enakomerno razporejene po množici $\{0,\ldots,\texttt{t.length}-1\}$. En način, kako to doseči je, da hranimo ogromno polje, `tab`, dolžine 2^w, kjer je vsak zapis naključno `w` bitno celo število, neodvisno od vseh ostalih zapisov. Na ta način bi lahko implementirali `hash(x)`, tako da bi izbrali `d` bitno celo število iz tabele `tab[x.hashCode()]`:

LinearHashTable

```
int idealHash(T x) {
  return tab[hashCode(x) >> w-d];
}
```

Na žalost je hranjenje polja velikosti 2^w neoptimalna rešitev, kar se tiče prostorske porabe. Pristop, ki ga uporablja *tabelarno zgoščevanje* je, da `w` bitna cela števila obravnava kot cela števila, ki so sestavljena iz `w/r` celih števil, ki imajo dolžino le `r` bitov. Tako pri tabelarnem zgoščevanju potrebujemo samo `w/r` polj velikosti 2^r. Vsi zapisi v teh poljih so neodvisna `w`-bitna cela števila. Da pridobimo vrednost `hash(x)`, razdelimo `x.hashCode()` v `w/r` `r`-bitnih celih števil ter jih uporabimo kot indekse za polja. Nato vse te vrednosti združimo z bitnim operatorjem izključni ali(XOR), da pridobimo `hash(x)`. Spodnja programska koda prikazuje kako to deluje za $w = 32$ in $r = 4$:

LinearHashTable

```
int hash(T x) {
  unsigned h = hashCode(x);
  return (tab[0][h&0xff]
        ^ tab[1][(h>>8)&0xff]
        ^ tab[2][(h>>16)&0xff]
        ^ tab[3][(h>>24)&0xff])
      >> (w-d);
}
```

V temu primeru je `tab` dvodimenzionalno polje s štirimi stolpci in $2^{32/4} = 256$ vrsticami.

Enostavno lahko preverimo, da je, za poljubni `x`, `hash(x)` enakomerno razporejen po intervalu $\{0,\ldots,2^d-1\}$. Z malo dodatnega dela lahko tudi preverimo, da ima poljubni par vrednosti neodvisne zgoščene vrednosti. To pomeni, da bi se za implementacijo `ChainedHashTable`, namesto zgoščevalne funkcije - metode množenja uporabilo tabelarno zgoščevanje.

Dejstvo, da ima poljubna množica `n` različnih vrednosti množico `n` neodvisnih zgoščenih vrednosti ne velja. Ne glede na to, pa velja, da ko uporabljamo tabelarno zgoščevanje, še vedno velja meja 5.2. Reference za to lahko najdete na koncu tega poglavja.

5.3 Zgoščene vrednosti

Zgoščene tabele, ki smo si jih pogledali v prejšnjem podpoglavju se uporabljajo za povezovanje podatkov s celoštevilskimi ključi sestavljenimi iz `w` bitov. Velikokrat pa uporabljamo ključe, ki niso cela števila. Lahko so nizi znakov, objekti, tabele ali ostale sestavljene strukture. Da lahko uporabimo zgoščevalne funkcije na takih tipih podatkov moramo prej preslikati te podatke v `w`-bitne zgoščene vrednosti. Preslikave zgoščevalnih funkcij morajo imeti naslednje lastnosti:

1. Če sta `x` in `y` enaka, potem morata biti enaka tudi `x.hashCode()` in `y.hashCode()`.
2. Če `x` in `y` nista enaka, potem mora biti verjetnost, da sta `x.hashCode()` = `y.hashCode()` majhna (blizu $1/2^w$).

Prva lastnost nam zagotavlja, da če v zgoščeni tabeli hranimo `x` in kasneje iščemo vrednost `y` (ki je enaka `x`), da bomo našli `x` . Druga lastnost pa nam preprečuje izgubo podatkov pri pretvarjanju objektov v cela števila. Zagotavlja nam, da bodo različni objekti imeli različno zgoščeno vrednost in bodo tako zelo verjetno shranjeni na različnih mestih v naši zgoščeni tabeli.

5.3.1 Zgoščene vrednosti osnovnih podatkovnih tipov

Za majhne osnovne podatkovne tipe kot so `char`, `byte`, `int`, in `float` lahko ponavadi hitro najdemo zgoščeno vrednost. Ti podatkovni tipi imajo vedno binarno predstavitev sestavljeno iz `w` ali manj bitov. (V C++ `char` ponavadi 8-bitni in `float` 32-bitni.) . V teh primerih te bite obravnavamo kot cela števila na intervalu $\{0,\ldots,2^{\mathtt{w}}-1\}$. Če sta dve vrednosti različni potem dobijo različni zgoščeni vrednosti. Če sta vrednosti enaki pa dobita enako zgoščeno vrednost.

Nekateri podatkovni tipi pa so sestavljeni iz več kot `w` bitov. Ponavadi c`w` bitov za neko konstantno celo število c . (V Javi sta `long` in `double` primera tipov pri katerih je $c = 2$.) Te podatkovne tipe lahko obravnavamo kot objekte sestavljene iz c delov, kot je opisano v naslednjem podpoglavju.

5.3.2 Zgoščene vrednosti sestavljenih podatkovnih tipov

Za sestavljene objekte si želimo zgraditi zgoščevalno funkcijo, ki bi kombinirala zgoščene vrednosti podatkovnih tipov, ki ta objekt sestavljajo. Vendar pa to ni tako enostavno kot zveni. Kljub temu, da lahko najdemo kar nekaj bljižnic s katerimi to lahko naredimo (na primer sestavljanje zgoščenih vrednosti z operacijo XOR) pa to ni rešitev problema, saj lahko hitro pridemo do primerov kjer take bljižnice odpovedo (glej naloge 5.7–5.9). A vendar obstajajo hitri in robustni načini reševanja tega problema, če si lahko privoščimo računanje z 2`w` bitno natančnostjo. Zamislimo si objekt sestavljen iz delov $P_0,\ldots,P_{r-1}$ katerih zgoščene vrednosti so $\mathtt{x}_0,\ldots,\mathtt{x}_{r-1}$. Potem si lahko izberemo neodvisna in naključna `w`-bitna števila $\mathtt{z}_0,\ldots,\mathtt{z}_{r-1}$ in eno liho in naključno celo število `z` sestavljeno iz 2`w` bitov. Iz tega lahko izračunamo zgoščeno vrednost za naš objekt na

naslednji način:

$$h(x_0,\ldots,x_{r-1}) = \left(\left(z\sum_{i=0}^{r-1} z_i x_i\right) \bmod 2^{2w}\right) \operatorname{div} 2^w \ .$$

Upoštevajte, da ima ta zgoščena vrednost zadnji korak (deljenje z z in deljenje z 2^w), ki uporablja multiplikativno zgoščevalno funkcijo iz 5.1.1, da vzame $2w$-bitni vmesni rezultat in ga pomanjša v w-bitni končni rezultat. Tukaj je primer te metode uporabljene na enostavnemu sestavljenemu podatkovnemu tipu s tremi deli `x0`, `x1`, and `x2`:

Point3D

```
unsigned hashCode() {
  // random number from random.org
  long long z[] = {0x2058cc50L, 0xcb19137eL, 0x2cb6b6fdL};
  long zz = 0xbea0107e5067d19dL;
  long h0 = ods::hashCode(x0);
  long h1 = ods::hashCode(x1);
  long h2 = ods::hashCode(x2);
  return (int)(((z[0]*h0 + z[1]*h1 + z[2]*h2)*zz) >> 32);
}
```

Naslednji izrek nam pokaže, da je metoda, ob tem da je enostavna za implementacijo, tudi dokazano dobra:

Izrek 5.3. *Naj bosta* $x_0,\ldots,x_{r-1}$ *in* $y_0,\ldots,y_{r-1}$ *sekvenci* w *bitnih integerjev v* $\{0,\ldots,2^w-1\}$ *in predvidevamo, da* $x_i \neq y_i$ *za vsaj en indeks* $i \in \{0,\ldots,r-1\}$. *Potem*

$$\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \leq 3/2^w \ .$$

Dokaz. Najprej bomo ignorirali zadnji multiplikativni zgoščevalni korak in si kasneje pogledali kako ta korak prispeva. Opredeli:

$$h'(x_0,\ldots,x_{r-1}) = \left(\sum_{j=0}^{r-1} z_j x_j\right) \bmod 2^{2w} \ .$$

Predvidevamo, da $h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$. To lahko zapišemo kot:

$$z_i(x_i - y_i) \bmod 2^{2w} = t \tag{5.4}$$

kjer

$$t = \left(\sum_{j=0}^{i-1} z_j(y_j - x_j) + \sum_{j=i+1}^{r-1} z_j(y_j - x_j) \right) \bmod 2^{2w}$$

Če predvidevamo, da je brez izgube splošnosti $x_i > y_i$, potem (5.4) postane

$$z_i(x_i - y_i) = t \ , \tag{5.5}$$

saj je vsak od z_i in $(x_i - y_i)$ največ $2^w - 1$, torej je njun produkt največ $2^{2w} - 2^{w+1} + 1 < 2^{2w} - 1$. Po domnevi, $x_i - y_i \neq 0$, torej (5.5) ima največ eno rešitev v z_i. Zato, ker sta z_i in t neodvisna ($z_0, \ldots, z_{r-1}$ sta medsebojno neodvisna), verjetnost, da izberemo z_i tako da je $h'(x_0, \ldots, x_{r-1}) = h'(y_0, \ldots, y_{r-1})$ največ $1/2^w$.

Zadnji korak zgoščevalne funkcije se uporablja za multiplikativno zgoščevanje, da zmanjšamo naše $2w$-bitne vmesne rezultate $h'(x_0, \ldots, x_{r-1})$ v w-bitni končni rezultat $h(x_0, \ldots, x_{r-1})$. Po teoremu 5.3, če $h'(x_0, \ldots, x_{r-1}) \neq h'(y_0, \ldots, y_{r-1})$, potem $\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq 2/2^w$.

Če povzamemo,

$$\begin{aligned}
&\Pr\left\{ \begin{array}{l} h(x_0, \ldots, x_{r-1}) \\ \quad = h(y_0, \ldots, y_{r-1}) \end{array} \right\} \\
&= \Pr\left\{ \begin{array}{l} h'(x_0, \ldots, x_{r-1}) = h'(y_0, \ldots, y_{r-1}) \text{ ali} \\ h'(x_0, \ldots, x_{r-1}) \neq h'(y_0, \ldots, y_{r-1}) \\ \quad \text{in } z h'(x_0, \ldots, x_{r-1}) \operatorname{div} 2^w = z h'(y_0, \ldots, y_{r-1}) \operatorname{div} 2^w \end{array} \right\} \\
&\leq 1/2^w + 2/2^w = 3/2^w \ .
\end{aligned}$$

□

5.3.3 Zgoščevalne funkcije za polja in nize

Metoda iz prejšnjega dela deluje dobro za objekte, ki imajo stalno število komponent. Vendar ne deluje dobro, ko jo želimo uporabiti za objekte, ki imajo spremenljivo število komponent, saj potrebuje naključno w-bitno celo število za vsako komponento. Lahko bi uporabili psevdonaključno zaporedje za generiranje toliko števil z kolikor jih potrebujemo, toda števila z niso medsebojno neodvisna, zaradi česar bi težko dokazali da psevdonaključna števila ne vplivajo na zgoščevalno funkcijo, ki jo uporabljamo. Vrednosti t in z v dokazu 5.3 nista več neodvisni.

Bolj temeljit pristop je, da uporabimo polinome nad praštevili. To pomeni le, da uporabimo običajne polinomske funkcije, ki dajo ostanek deljenja z nekim praštevilom p. Ta metoda sloni nad sledečim teoremom, ki pravi, da se takšne funkcije obnašajo podobno kot običajne polinomske funkcije:

Izrek 5.4. *Naj bo p praštevilo in $f(z) = x_0 z^0 + x_1 z^1 + \cdots + x_{r-1} z^{r-1}$ netrivialni polinom s koeficienti $x_i \in \{0, \ldots, p-1\}$. Takrat ima enačba $f(z) \bmod p = 0$ največ $r-1$ rešitev za $z \in \{0, \ldots, p-1\}$.*

Da izkoristimo **??** , uporabimo zgoščevalno funkcijo nad zaporedjem celih števil $x_0, \ldots, x_{r-1}$ kjer je vsak $x_i \in \{0, \ldots, p-2\}$ z uporabo naključnega celega števila $z \in \{0, \ldots, p-1\}$ in funkcije

$$h(x_0, \ldots, x_{r-1}) = \left(x_0 z^0 + \cdots + x_{r-1} z^{r-1} + (p-1) z^r \right) \bmod p \ .$$

Ste opazili $(p-1)z^r$ na koncu formule? To si lahko predstavljate kot zadnji element, x_r, v zaporedju $x_0, \ldots, x_r$. Ta element se razlikuje od vseh ostalih (ki so v $\{0, \ldots, p-2\}$). $p-1$ je kot znak, ki označuje konec zaporedja.

Sledeči teorem, ki upošteva primer, ko sta obe zaporedji enako dolgi, dokazuje, da ta zgoščevalna funkcija daje dober rezutat pri majhni meri naključnosti pri izbiri z:

Izrek 5.5. *Vzemimo $p > 2^w + 1$ da je naravno število, vzemimo $x_0, \ldots, x_{r-1}$ in $y_0, \ldots, y_{r-1}$ vsako je sekvenca w-bit celih števil v $\{0, \ldots, 2^w - 1\}$ in predpostavimo $x_i \neq y_i$ za vsaj en indeks $i \in \{0, \ldots, r-1\}$. Potem*

$$\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq (r-1)/p\} \ .$$

Dokaz. Enačba $h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})$ je lahko napisana kot

$$\left((x_0 - y_0) z^0 + \cdots + (x_{r-1} - y_{r-1}) z^{r-1} \right) \bmod p = 0. \tag{5.6}$$

Ker $x_i \neq y_i$, je ta polinom netrivialen. Potemtakem, po **??**, ima največ $r-1$ rešitev v z. Verjetnost, da izberemo z, ki je ena od teh rešitev, je potemtakem v najboljšem primeru $(r-1)/p$. □

Opozorimo, da ima ta zgoščevalna funkcija, prav tako opravka s primeri v katerih imata dve sekvenci različno dolžino, čeprav je ena od sekvenc predpona drugi. To je zaradi tega, ker ta funkcija efektivno razpršuje

neskončno sekvenco

$$x_0, \ldots, x_{r-1}, p-1, 0, 0, \ldots \ .$$

To zagotavlja, da če imamo dve sekvenci dolžine r in r' z $r > r'$, potem se ti dve sekvenci razlikujeta v indeksu $i = r$. V tem primeru (5.6) postane

$$\left(\sum_{i=0}^{i=r'-1} (x_i - y_i) z^i + (x_{r'} - p + 1) z^{r'} + \sum_{i=r'+1}^{i=r-1} x_i z^i + (p-1) z^r \right) \bmod p = 0 \ ,$$

katero, po **??**, ima največ r rešitev v z. Skupaj z 5.5 to zadostuje za dokaz naslednjega bolj splošnega teorema:

Izrek 5.6. *Vzemimo* $p > 2^w + 1$ *da je naravno število, vzemimo* $x_0, \ldots, x_{r-1}$ *in* $y_0, \ldots, y_{r'-1}$ *, da sta unikatne sekvence* w*-bit celih števil* v $\{0, \ldots, 2^w - 1\}$*. Potem*

$$\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq \max\{r, r'\}/p \ .$$

Sledeči primer kode prikazuje kako je ta zgoščevalna funkcija uporabljena na objektu, ki vsebuje polje `x`, ki vsebuje vrednosti:

GeomVector

```
unsigned hashCode() {
  long p = (1L<<32)-5;   // prime: 2^32 - 5
  long z = 0x64b6055aL;  // 32 bits from random.org
  int z2 = 0x5067d19d;   // random odd 32 bit number
  long s = 0;
  long zi = 1;
  for (int i = 0; i < x.length; i++) {
    // reduce to 31 bits
    long long xi = (ods::hashCode(x[i]) * z2) >> 1;
    s = (s + zi * xi) % p;
    zi = (zi * z) % p;
  }
  s = (s + zi * (p-1)) % p;
  return (int)s;
}
```

Predstavljena koda žrtvuje nekaj verjetnosti kolizije zaradi implementacijske uporabnosti. Zlasti zaradi tega, ker aplicira multiplikativno razpr-ši-tveno funkcijo iz 5.1.1, z `d` = 31 za zmanjšanje `x[i].hashCode()` v

31-bit vrednost. To je zaradi tega, da seštevanje in množenje, ki sta narejena po operaciji modula naravnega števila $p = 2^{32} - 5$, se lahko izvede z uporabo nepodpisane 63-bit aritmetike. Zaradi tega, je verjetnost dveh različnih sekvenc, od tega ima daljša dolžino r, da imata enako razpršitveno kodo v najslabšem primeru

$$2/2^{31} + r/(2^{32} - 5)$$

za razliko od $r/(2^{32} - 5)$ specificirano v 5.6.

5.4 Razprave in primeri

Ideja *zgoščevanja z množenjem* je zelo stara in je del zgoščevalne folklore [**?**, Razdelek 6.4]. Vendar je ideja, da izberemo množitelja z kot naključno *sodo* število in analiza 5.1.1, zastarela po mnenju Dietzfelbingerja *et al.* [**?**]. Ta različica zgoščevanja z množenjem je ena od najpreprostejših, ampak njena verjetnost kolizije $2/2^d$ je za dva faktorja večja kot pri naključni funkciji $2^w \to 2^d$. Zgoščevalna metoda *zmnoži-seštej* uporablja funkcijo

$$h(x) = ((zx + b) \bmod 2^{2w}) \operatorname{div} 2^{2w-d}$$

kjer sta z in b naključno izbrana iz $\{0, \ldots, 2^{2w}-1\}$. Zmnoži-seštej zgoščevanje ima verjetnost kolizije samo $1/2^d$ [**?**], ampak zahteva $2w$-bitne aritmetične operacije.

Obstaja kar nekaj metod za pridobivanje zgoščenih vrednosti iz zaporedja fiksne dolžine, vsebujoč w-bitnih celih števil. Še posebej hitra metoda [**?**] je funkcija

$$\begin{aligned} &h(x_0, \ldots, x_{r-1}) \\ &\quad = \left(\textstyle\sum_{i=0}^{r/2-1} ((x_{2i} + a_{2i}) \bmod 2^w)((x_{2i+1} + a_{2i+1}) \bmod 2^w)\right) \bmod 2^{2w} \end{aligned}$$

kjer je r parno število in $a_0, \ldots, a_{r-1}$ naključno izbrani iz $\{0, \ldots, 2^w\}$. To ustvari $2w$-bitno zgoščeno vrednost, katere možnost kolizije je $1/2^w$. To se lahko zmanjša na w-bitno zgoščeno vrednost z uporabo zgoščevanja z množenjem. Ta metoda je hitra, ker zahteva samo $r/2$ $2w$-bitnih množenj, metoda omenjena v 5.3.2 pa zahteva r množenj. (mod operacije se dogajajo zaporedno z uporabo w in $2w$-bitne aritmetične operacije za seštevanje in množenje.)

Metoda iz 5.3.3, ki uporablja polinome in polja praštevil za zgoščevanje tabel in nizov spremenljive dolžine je zastarela po mnenju Dietzfelbingerja *et al.* [**?**]. Zaradi njene uporabe mod operatorja, ki se zanaša na potrošne strojne ukaze je na žalost počasna. Nekatere različice te metode določijo praštevilo p iz obrazca $2^w - 1$. V tem primeru se lahko operator mod zamenja s prištevanjem (+) in logično in(&) operacijo [**?**, Section 3.6]. Druga možnost je uporaba hitrejše metode za nize fiksne velikosti pri blokih dolžine c za neko konstanto $c > 1$ in potem metode s polji praštevil za zaporedje $\lceil r/c \rceil$ zgoščenih vrednosti.

Naloga 5.1. Nekatere univerze vsakemu študentu določijo študentsko številko, ko se prvič prijavijo za katerikoli predmet. Te številke so zaporedna cela števila, ki so se začela z 0 mnogo let nazaj in so sedaj zapisana že v milijonih. Recimo, da imamo razred stotih novih študentov in bi radi vsakemu študentu dodelili zgoščeno vrednost, ki je odvisna od njihovih študentskih številk. Ali ima več smisla uporabiti prvi dve števki ali zadnji dve števki študentske številke? Pojasni svoj odgovor.

Naloga 5.2. Upoštevajte zgoščevalno funkcijo iz odstavka 5.1.1, in predpostavite, da je $n = 2^d$ and $d \leq w/2$.

1. Pokažite, da za vsakega izbranega množitelja, z, obstajajo vrednosti n, ki imajo enako zgoščeno vrednost. (Namig: Gre za preprosto rešitev, ki ne zahteva teorije števil).

2. Glede na podanega množitelja, z, opišite tiste vrednosti n, ki imajo enako zgoščeno vrednost. (Hint: Ta primer je zahtevnejši in zahteva poznavanje osnov teorije števil.)

Naloga 5.3. Pokažite, da je meja dovoljene vrednosti $2/2^d$ v trditvi 5.1 najboljša možna meja, če je $x = 2^{w-d-2}$ in $y = 3x$, then $\Pr\{\mathtt{hash}(x) = \mathtt{hash}(y)\} = 2/2^d$. (Namig: Poglejte si binarni prikaz za zx in $z3x$ in upoštevajte dejstvo, da je $z3x = zx + 2zx$.)

Naloga 5.4. Dokažite trditev 5.4 z uporabo Stirlingove aproksimacije iz poglavja 1.3.2.

Naloga 5.5. Upoštevajte spodnjo poenostavljeno verzijo kode za dodajanje elementa x v LinearHashTable (linearno razpršeno tabelo), ki element x shrani v prvo polje v tabeli, ki vsebuje vrednost null. Opišite

zakaj je ta način dodajanja elementov zelo počasen. Pokažite to na primeru zaporednega izvajanja operacij $O(n)$ `add(x)`, `remove(x)`, in `find(x)`, ki za izvedbo porabijo n^2 časa.

LinearHashTable

```
bool addSlow(T x) {
  if (2*(q+1) > t.length) resize();   // max 50% occupancy
  int i = hash(x);
  while (t[i] != null) {
      if (t[i] != del && x.equals(t[i])) return false;
      i = (i == t.length-1) ? 0 : i + 1; // increment i
  }
  t[i] = x;
  n++; q++;
  return true;
}
```

Naloga 5.6. Zgodnejše verzije metode Java `hashCode()` za razred `String` ni delovala tako, da bi uporabila vse znake v dolgem nizu. Naprimer, za 16 znakov dolg niz se je koda razpršitve izračunala glede na osem sodo indeksiranih znakov. Na primeru pojasnite zakaj to ni bila pametna ideja. Primer naj sestoji iz večjega nabora nizov, pri čemer naj imajo vsi enako kodo razpršitve.

Naloga 5.7. Predpostavite da imate objekt sestavljen iz dveh `w`-bitnih števil, `x` in `y`. Pokažite zakaj $x \oplus y$ ni dobra koda razpršitve za vaš objekt. Pokažite tudi primer večje množice objektov, kjer bi vsi imeli kodo raz-pr-ši-tve 0.

Naloga 5.8. Predpostavite da imate objekt sestavljen iz dveh `w`-bitnih števil, `x` in `y`. Pokažite zakaj $x + y$ ni dobra koda razpršitve za vaš objekt. Pokažite tudi primer večje množice objektov, kjer bi vsi imeli enako kodo razpršitve.

Naloga 5.9. Predpostavite da imate objekt sestavljen iz dveh `w`-bitnih števil, `x` in `y`. Predpostavite tudi, da je koda razpršitve za vaš objekt definirana z deterministično funkcijo $h(x, y)$, ki ustvari eno samo `w`-bitno število. Dokažite da obstaja večja množica objektov, ki imajo enako kodo razpršitve.

Naloga 5.10. Naj za neko pozitivno število `w` velja $p = 2^w - 1$. Razložite zakaj za pozitivno število x velja

$$(x \bmod 2^w) + (x \operatorname{div} 2^w) \equiv x \bmod (2^w - 1) \ .$$

(Dobimo algoritem za računanje $x \bmod (2^w - 1)$ s pomočjo zaporednega nastavljanja

$$\texttt{x} = \texttt{x}\&((1 << \texttt{w}) - 1) + \texttt{x} >> \texttt{w}$$

dokler ne velja $\texttt{x} \leq 2^w - 1$.)

Naloga 5.11. Izberite neko pogostokrat uporabljeno implementacijo zgoščе-ne tabele kot je recimo The C++ STL `unordered_map` ali `HashTable` oziroma `LinearHashTable` iz te knjige in napišite program, ki v to podatkovno strukturo shranjuje števila, `x`, tako da je časovna zahtevnost funkcije `find(x)` linearna. Se pravi, poiščite množico `n` števil v kateri je c`n` elementov, katerih koda razpršitve je na isti lokaciji v tabeli. Odvisno od kvalitete implementacije boste to mogoče lahko dosegli že samo z natančnim pregledom kode ali pa boste morali napisati nekaj vrstic kode, ki bo poskušala z vstavljanjem in iskanjem elementov ter merjenjem časa za dodajanje in iskanje posameznih vrednosti. (To se lahko, se tudi že je, uporabi za napad DOS(denial of service) na strežnike [**?**].)

Poglavje 6

Dvojiška drevesa

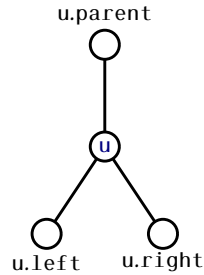
To poglavje vpeljuje eno izmed najbolj temeljnih struktur v računalništvu: dvojiška drevesa. Uporaba besede *drevo* prihaja iz dejstva da je, ko jih rišemo, končna risba podobna drevesom iz gozda. Obstaja veliko načinov definiranja dvojiških dreves. Matematično je *dvojiško drevo* povezan, neusmerjen, končni graf brez ciklov, katerih stopnja bi bila večja od tri.

Za večino aplikacij v računalništvu, so dvojiška drevesa *zakoreninjena:* Posebno vozlišče r, v največ drugi stopnji, se imenuje *koren* drevesa. Za vsako vozlišče u ≠ r, se drugo vozlišče na poti od u do r imenuje *starš* od u . Vsako drugo vozlišče, ki meji na u imenujemo *otrok* od u. Večina dvojiških dreves, ki nas zanimajo, je *urejena* in tako lahko ločimo med *levi otrok* in *desni otrok* od u.

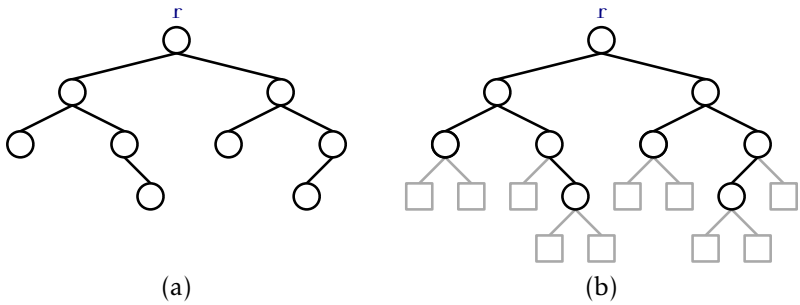
V ilustracijah, so dvojiška drevesa običajno narisana iz korena navzdol, s korenom na vrhu slike in levim ter desnim otrokom tako, da je levi otrok na levi strani, desni pa na desni strani (6.1). Na primer 6.2. kaže dvojiško drevo z devetimi vozlišči.

Ker so dvojiška drevesa tako pomembna, so za njih razvili določeno terminologijo: *globina* vozlišča, u, je v dvojiškem drevesu dolžina poti od u do korena drevesa. Če je vozlišče w, na poti od u do r, potem w imenujemo *prednik* od u in u pa imenujemo *naslednik* od w. *poddrevo* od vozlišča u je dvojiško drevo, ki ima korenine v u in vsebuje vse naslednike od u. *višina* vozlišča u, je dolžina najdaljše poti od u do enega od njenih naslednikov. *višina* drevesa je višina njegovega korena. Vozlišče u, je *list* če nima nobenega otroka.

Včasih mislimo, da so drevesa utrjena z *zunanjimi vozlišči*. Vsako vo-



Slika 6.1: Starš, levi otrok, desni otrok vozlišča u v BinaryTree.



Slika 6.2: Dvojiško drevo (a) devet vozlišč in (b) deset zunanjih vozlišč.

zlišče, ki nima levega otroka ima zunanje vozlišče kot za svojega levega otroka in podobno vsako vozlišče, ki nima desnega otroka ima zunanje vozlišče kot za svojega desnega otroka (glej 6.2.b). Z indukcijo lahko enostavno preverimo, da ima dvojiško drevo z $n \geq 1$ pravimi vozlišči $n+1$ zunanjih vozlišč.

6.1 BinaryTree: Osnovno dvojiško drevo

Najenostavnejši način, predstavitve vozlišča u, v dvojiškem drevesu je izrecno shranjevanje (največ treh) sosedov od u:

BinaryTree

```
class BTNode {
  N *left;
  N *right;
  N *parent;
  BTNode() {
    left = right = parent = NULL;
  }
};
```

Ko eden od treh sosedov ni prisoten, ga nastavimo na `nil`. Na ta način sta oba zunanja vozlišča drevesa in starš korena vrednosti `nil`.

Dvojiško drevo se lahko zastopa kot pointer do svojega vozlišča korena r:

BinaryTree

```
Node *r;    // root node
```

Globino vozlišča u, lahko izračunamo tako, da štejemo korake od u do korena:

BinaryTree

```
int depth(Node *u) {
  int d = 0;
  while (u != r) {
    u = u->parent;
    d++;
  }
  return d;
}
```

6.1.1 Rekurzivni algoritmi

Z uporabo rekurzivnih algoritmov je izračun o dvojiških drevesih enostaven. Na primer, za izračun velikosti (število vozlišč) dvojiškega drevesa, ki je zakoreninjen v vozlišču u, naredimo tako da rekurzivno izračunamo velikost dveh poddreves, ki so zakoreninjena na otroke od u, nato povzamemo te velikosti, in dodamo eno:

BinaryTree
```
int size(Node *u) {
  if (u == nil) return 0;
  return 1 + size(u->left) + size(u->right);
}
```

Za izračun višine vozlišča u moremo izračunati višino u- jevih dveh poddreves, vzeti največjega in mu dodati:

BinaryTree
```
int height(Node *u) {
  if (u == nil) return -1;
  return 1 + max(height(u->left), height(u->right));
}
```

6.1.2 Obiskovanje dvojiškega drevesa

Prejšnja algoritma iz prejšnjega odseka uporabljata rekurzijo, za obisk vseh vozlišč v dvojiškem drevesu. Vsak od njih obišče vozlišča dvojiškega drevesa v istem vrstnem redu kot naslednja koda:

BinaryTree
```
void traverse(Node *u) {
    if (u == nil) return;
    traverse(u->left);
    traverse(u->right);
}
```

Z uporabo rekurzije, lahko na ta način proizvajamo zelo kratko in preprosto kodo, lahko pa je taka koda zelo problematična. Največja globina rekurzije je podana z največjo globino vozlišča v dvojiškem drevesu, tj.

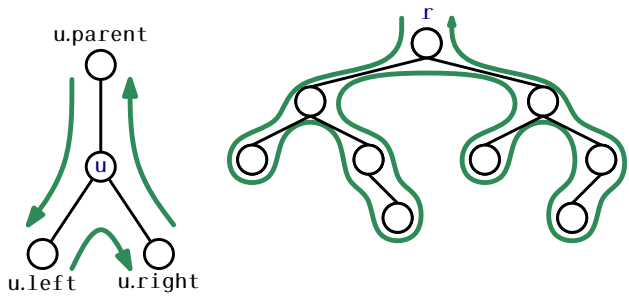
višina drevesa. Če je višina drevesa zelo velika, potem lahko taka rekurzija porabi veliko več pomnilnika na skladu, kot ga je na voljo.

Za obhod dvojiškega drevesa brez rekurzije lahko uporabimo algoritem, ki se zanaša na to, da ve, iz kje je prišel in kam bo odšel. Glej 6.3. Če pridemo do vozlišča `u` od `u.parent`, potem obiščemo `u.left`. Če pridemo do `u` od `u.left`, potem obiščemo `u.right`. Če prispemo na `u` iz `u.right`, potem smo končali z obiskovanjem `u` -jevih poddreves in se tako vrnemo na `u.parent`. Naslednja koda izvaja idejo, ki vključuje ravnanje v primerih, ko katera koli od `u.left`, `u.right` ali `u.parent` je `nil`:

BinaryTree
```
void traverse2() {
  Node *u = r, *prev = nil, *next;
  while (u != nil) {
    if (prev == u->parent) {
      if (u->left != nil) next = u->left;
      else if (u->right != nil) next = u->right;
      else next = u->parent;
    } else if (prev == u->left) {
      if (u->right != nil) next = u->right;
      else next = u->parent;
    } else {
      next = u->parent;
    }
    prev = u;
    u = next;
  }
}
```

Enake primere, ki jih lahko izračunamo z rekurzivnimi algoritmi, lahko izračunamo z iterativnimi algoritmi. Na primer, za izračun velikosti drevesa hranimo števec `n`, in nižamo `n` vsakič ko obiščemo novo vozlišče.

BinaryTree
```
int size2() {
    Node *u = r, *prev = nil, *next;
    int n = 0;
    while (u != nil) {
      if (prev == u->parent) {
        n++;
        if (u->left != nil) next = u->left;
        else if (u->right != nil) next = u->right;
```

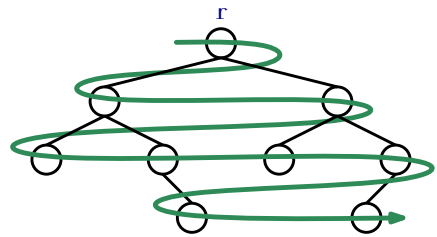


Slika 6.3: Trije primeri, ki se pojavijo na vozlišču u kadar obhodimo dvojiškega drevesa,ki niso rekurzivna

```
        else next = u->parent;
      } else if (prev == u->left) {
        if (u->right != nil) next = u->right;
        else next = u->parent;
      } else {
        next = u->parent;
      }
      prev = u;
      u = next;
    }
    return n;
  }
```

V nekaterih implementacijah dvojiških dreves, se `parent` ne uporablja. V takih primerih, lahko še vedno uporabimo iterativno izvedbo, vendar mora taka izvedba uporabljati `List` (ali `Stack`), saj bi tako lahko spremljali pot od trenutnega vozlišča do korena.

Posebna vrsta prečkanja, ki ne ustreza vzorcu zgoraj navedene funkcije je *prvi-v-širino*. V prvi-v-širino obhodu, so vozlišča obiskana stopnja-postopnjo, pri kateremu začnemo v korenu in nadaljujemo navzdol, kjer obiskujemo vsako vozlišče od levega proti desni (glej 6.4). To je podobno načinu branja strani v Angleškem jeziku. Prvi-v-širino obhod je implementiran z uporabo vrste `q`, ki na začetku vsebuje samo koren `r`. Na vsakem koraku, vzamemo naslednje vozlišče `u` iz `q` , nato procesiramo `u`, in dodamo `u.left` in `u.right` (če niso `nil`) v `q`:



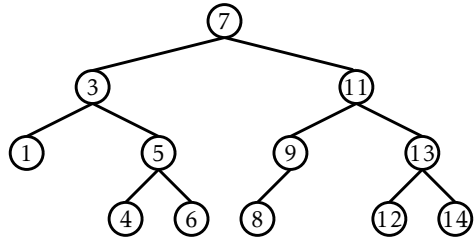
Slika 6.4: Med prvi-v-širino obhodu, so vozlišča v dvojiškem drevesu obiskana po principu stopnja-po-stopnjo in levo-proti-desni za vsako stopnjo.

BinaryTree

```
void bfTraverse() {
  ArrayDeque<Node*> q;
  if (r != nil) q.add(q.size(),r);
  while (q.size() > 0) {
    Node *u = q.remove(q.size()-1);
    if (u->left != nil) q.add(q.size(),u->left);
    if (u->right != nil) q.add(q.size(),u->right);
  }
}
```

6.2 BinarySearchTree: Neuravnoteženo dvojiško iskalno drevo

BinarySearchTree je posebna oblika dvojiškega drevesa, pri katerem vsako vozlišče u hrani tudi podatek u.x iz nekega skupnega vrstnega reda. Podatki dvojiškega iskalnega drevesa upoštevajo *lastnost dvojiških iskalnih dreves*: Za vozlišče u velja, da vsak podatek shranjen v poddrevesu u.left je manjši od u.x ter vsak podatek shranjen v poddrevesu u.right je večji od u.x. Primer BinarySearchTree je prikazan v 6.5.



Slika 6.5: Dvojiško iskalno drevo.

6.2.1 Iskanje

Lastnost dvojiškega iskalnega drevesa je zelo uporabna, ker nam omogoča hitro iskanje vrednosti `x` v dvojiškem iskalnem drevesu. To naredimo tako, da začnemo z iskanjem vrednosti `x` v korenu `r`. Ko pregledamo vozlišče `u`, imamo tri možnosti:

1. Če je $x < u.x$, nadaljujemo z iskanjem v `u.left`;
2. Če je $x > u.x$, nadaljujemo z iskanjem v `u.right`;
3. Če je $x = u.x$, pomeni, da smo našli vozlišče `u`, ki hrani `x`.

Iskanje se zaključi, ko dosežemo Možnost 3 ali ko je $u = \texttt{nil}$ (prazen). V prvem primeru smo našli `x`. V drugem pa sklenemo, da `x` ni v dvojiškem iskalnem drevesu.

BinarySearchTree

```
T findEQ(T x) {
  Node *w = r;
  while (w != nil) {
    int comp = compare(x, w->x);
    if (comp < 0) {
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
```

```
      return w->x;
    }
  }
  return null;
}
```

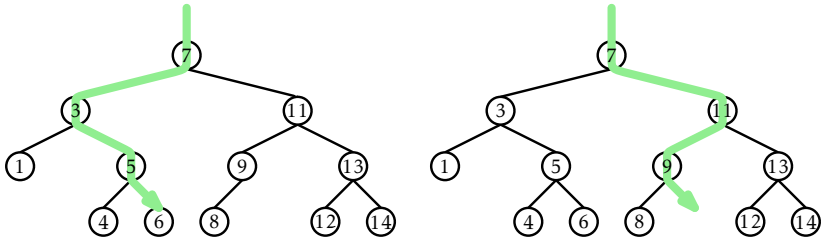
V 6.6 sta prikazana dva primera iskanj v dvojiškem iskalnem drevesu. Drugi primer prikazuje, da tudi če ne najdemo x v drevesu, vseeno pridobimo nekaj pomembnih informacij. Če pogledamo zadnje vozlišče u pri katerem se je zgodila Možnost 1, vidimo, da je u.x najmanjša vrednost v drevesu, ki je večja od x. Podobno, zadnje vozlišče kjer se je zgodila Možnost 2 hrani največjo vrednost v drevesu, ki je manjša od x. Torej, ob spremljanju zadnjega vozlišča z pri katerem se je zgodila Možnost 1, lahko BinarySearchTree implementira find(x) funkcijo, ki vrne najmanjšo vrednost v drevesu, ki je večja ali enaka x:

BinarySearchTree

```
T find(T x) {
  Node *w = r, *z = nil;
  while (w != nil) {
    int comp = compare(x, w->x);
    if (comp < 0) {
      z = w;
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
      return w->x;
    }
  }
  return z == nil ? null : z->x;
}
```

6.2.2 Vstavljanje

Pri vstavljanju nove vrednosti x v BinarySearchTree, najprej poiščemo x v drevesu. Če ga najdemo, potem vstavljanje ni potrebno. V nasprotnem primeru shranimo x v otroka zadnjega vozlišča p, ki smo ga obiskali med iskanjem za vrednostjo x. Ali je novo vozlišče levi ali desni otrok vozlišča



Slika 6.6: Primer (a) uspešnega iskanja (za 6) ter (b) neuspešnega iskanja (za 10) v dvojiškem iskalnem drevesu.

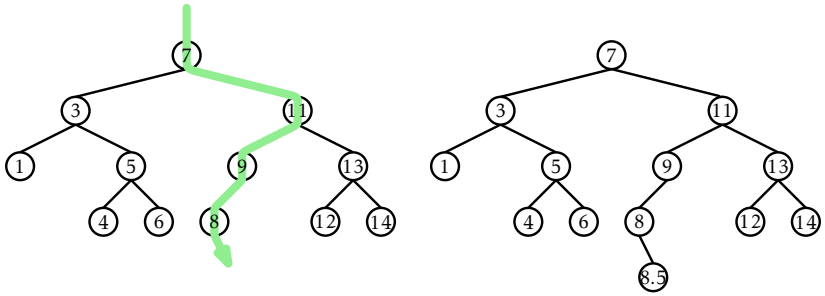
p, je odvisno od rezultata primerjave med x ter p.x.

BinarySearchTree

```
bool add(T x) {
  Node *p = findLast(x);
  Node *u = new Node;
  u->x = x;
  return addChild(p, u);
}
```

BinarySearchTree

```
Node* findLast(T x) {
  Node *w = r, *prev = nil;
  while (w != nil) {
    prev = w;
    int comp = compare(x, w->x);
    if (comp < 0) {
      w = w->left;
    } else if (comp > 0) {
      w = w->right;
    } else {
      return w;
    }
  }
  return prev;
}
```



Slika 6.7: Vstavljanje vrednosti 8.5 v dvojiško iskalno drevo.

BinarySearchTree

```
bool addChild(Node *p, Node *u) {
    if (p == nil) {
      r = u;               // inserting into empty tree
    } else {
      int comp = compare(u->x, p->x);
      if (comp < 0) {
        p->left = u;
      } else if (comp > 0) {
        p->right = u;
      } else {
        return false;   // u.x is already in the tree
      }
      u->parent = p;
    }
    n++;
    return true;
  }
```

Primer je prikazan v 6.7. Najbolj časovno požrešen del tega procesa je začetno iskanje vozlišča `x`, ki porabi količino časa sorazmerno z višino novo vstavljenega vozlišča `u`. V najslabšem primeru je ta enaka višini `BinarySearchTree`.

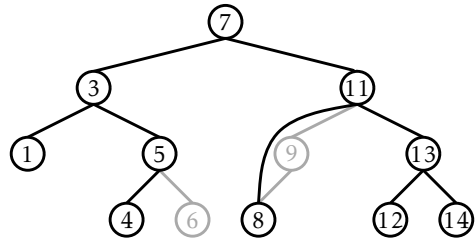
6.2.3 Brisanje

Brisanje vrednosti, ki jo hrani vozlišče u v strukturi BinarySearchTree je malce težje. Če je u list potem preprosto odstranimo u iz seznama otrok njegovega starša. V primeru, da ima u samo enega otroka lahko odstranimo u iz drevesa tako, da u.parent posvoji u-jevega otroka(glej 6.8):

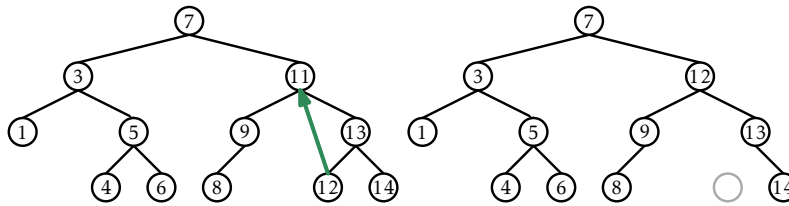
BinarySearchTree

```
void splice(Node *u) {
  Node *s, *p;
  if (u->left != nil) {
    s = u->left;
  } else {
    s = u->right;
  }
  if (u == r) {
    r = s;
    p = nil;
  } else {
    p = u->parent;
    if (p->left == u) {
      p->left = s;
    } else {
      p->right = s;
    }
  }
  if (s != nil) {
    s->parent = p;
  }
  n--;
}
```

Brisanje pa se zakomplicira, ko ima u dva otroka. V tem primeru je najlažje poiskati neko vozlišče w, ki ima manj kot dva otroka, ter da w.x lahko zamenja u.x. Za ohranjanje lastnosti dvojiškega iskalnega drevesa mora biti vrednost w.x blizu vrednosti u.x. Na primer: če bi izbrali w tako, da je w.x najmanjša vrednost, ki je večja od u.x, bi delovalo. Iskanje primernega vozlišča w je preprosto: to je najmanjša vrednost, ki se nahaja v poddrevesu u.right. To vozlišče lahko brez skrbi odstranimo, ker nima levega otroka (glej 6.9).



Slika 6.8: Brisanje lista (6) ali vozlišča z enim otrokom (9) je preprosto.



Slika 6.9: Brisanje neke vrednosti (11) iz nekega vozlišča u, ki ima dva otroka, počnemo z zamenjavo u-jeve vrednosti z najmanjšo vrednostjo v u-jevem desnem poddrevesu.

BinarySearchTree

```
void remove(Node *u) {
  if (u->left == nil || u->right == nil) {
    splice(u);
    delete u;
  } else {
    Node *w = u->right;
    while (w->left != nil)
      w = w->left;
    u->x = w->x;
    splice(w);
    delete w;
  }
}
```

6.2.4 Povzetek

Vsaka izmed funkcij `find(x)`, `add(x)` ter `remove(x)` v strukturi `BinarySearchTree` vključuje sledenje neki poti od korena pa do nekega vozlišča v drevesu. Brez dodatnega znanja o obliki drevesa je težko karkoli povedati o dolžini te poti, razen tega, da je pot manjša kot `n` - število vseh vozlišč v drevesu. Sledeči izrek povzame zmožnosti podatkovne strukture `BinarySearchTree`:

Izrek 6.1. *`BinarySearchTree` implementira `SSet` vmesnik ter podpira funkcije* `add(x)`, `remove(x)` *ter* `find(x)` *v* $O(n)$ *časa na operacijo.*

6.1 se slabo primerja z 4.1, ki prikazuje, da struktura `SkiplistSSet` lahko implementira `SSet` vmesnik z pričakovanim časom $O(\log n)$ na operacijo. Problem strukture `BinarySearchTree` tiči v tem, da lahko postane *neuravnoteženo*. Namesto da drevo izgleda kot na 6.5, lahko izgleda kot dolga veriga z `n` vozlišči, ki imajo po točno enega otroka, razen zadnjega, ki nima nobenega.

Obstaja več načinov, kako se izogniti strukturi `BinarySearchTree`, ki je neuravnotežena. Vsi načini vodijo v podatkovne strukture, ki imajo operacije s časom $O(\log n)$. V 7 pokažemo, kako lahko dosežemo operacije z *pričakovanim* časom $O(\log n)$ s pomočjo naključnosti. V 8 pokažemo, kako dosežemo operacije z *amortiziranim* časom $O(\log n)$ s pomočjo delnih obnovitvenih operacij. V 9 pokažemo, kako dosežemo operacije z *najslabšim* časom $O(\log n)$ s pomočjo simulacije dreves, ki niso dvojiška: eno v katerem imajo vozlišča lahko do štiri otroke.

6.3 `BinaryTree`: Razprava in vaje

Dvojiška drevesa se že tisočletja uporabljajo za predstavitev razmerij med elementi. Med drugim se uporabljajo tudi za prikaz družinskih dreves (rodovnika). Vzemimo primer, koren drevesa je oseba A. Levi in desni otrok osebe A sta njena starša in sta vozlišči drevesa; zgodba se naprej ponavlja za vsako vozlišče rekurzivno v globino. V zadnjih stoletjih se uporabljajo tudi v biologiji, natančneje za prikaz vrst v drevesni strukturi, kjer listi drevesa predstavljajo obstoječe vrste, vozlišča znotraj drevesa pa

dogodke v razvoju, kjer se iz ene razvijeta dve novi vrsti (*angl: speciation event*).

V petdesetih letih 19. st. so raziskovalci odkrili dvojiška iskalna drevesa. Več o dvojiških iskalnih drevesih lahko preberete v nadaljevanju.

Ko se srečamo z implementacijo dvojiških dreves, zlasti če le-te gradimo z ničle, se moramo dogovoriti za nekaj pravil. Eno izmed pravil je vprašanje: naj vozlišča drevesa vsebujejo kazalce na svoje starše ali ne? Če večina operacij na drevesu poteka od korena do listov, potem kazalcev ne potrebujemo. Po drugi strani pa to pomeni, da moramo t.i. sprehode po drevesu implementirati rekurzivno ali pa z uporabo posebnih skladov. Pomanjkanje kazalcev se pozna tudi v nekaterih drugih metodah, kot sta npr. vstavljanje in brisanje elementov iz dvojiškega drevesa, kjer se brez kazalcev njuna implementacija močno zaplete.

Drugo pravilo govori o tem kako v vozlišču hraniti kazalce na starša ter levega in desnega otroka. Ena možnost je, da so kazalci shranjeni kot spremenljivke. Druga možnost je, da jih hranimo v tabeli p, ki je dolžine 3 tako, da je vrednost $u.p[0]$ levi otrok vozlišča u, vrednost $u.p[1]$ je desni otrok vozlišča u, vrednost $u.p[2]$ pa je starš vozlišča u. Z uporabo tabele kazalcev tudi omogočimo poenostavitev nekaterih zaporedij `if` stavkov v algebraične izraze.

Takšno poenostavitev lahko opazimo ob sprehodu po drevesu. Ko naletimo na vozlišče u iz tabele $u.p[i]$, je naslednje vozlišče v sprehodu $u.p[(i+1) \bmod 3]$. Podoben primer nastopi, ko v drevesu nimamo levo-desne simetrije. Npr. sorojenec (tj. brat) vozlišča $u.p[i]$ je $u.p[(i+1) \bmod 2]$. Takšen trik deluje, če je vozlišče $u.p[i]$ levi otrok ($i = 0$) ali desni otrok ($i = 1$) vozlišča u. S tem nam ni več potrebno pisati *leve* in *desne* kode (tj. dve različni kodi za urejanje leve in desne strani drevesa), temveč lahko vse združimo v en sam košček kode. Kot primer si lahko ogledate metodi `rotateLeft(u)` in `rotateRight(u)` na strani 162.

Vaja 6.1. Dokažite, da ima dvojiško drevo s številom vozlišč $n \geq 1$ $n-1$ povezav.

Vaja 6.2. Dokažite, da ima dvojiško drevo s številom notranjih vozlišč $n \geq 1$ $n+1$ zunanjih vozlišč (listov).

Vaja 6.3. Privzemimo, da imamo dvojiško drevo T z vsaj enim listom. Dokažite bodisi, da: a) ima koren drevesa največ enega otroka bodisi b), da ima drevo T več kot en list.

Vaja 6.4. Implementirajte nerekurzivno metodo *size2*(u), ki izračunava velikost podrevesa vozlišča u. **Vaja 6.5.** Napišite nerekurzivno metodo *height2*(u), ki izračunava višino vozlišča u v dvojiškem drevesu. **Vaja 6.6.** Dvojiško drevo je uravnoteženo, če za vsako vozlišče u velja, da se višina njegovih poddreves z vozliščem `u.left` in `u.right` razlikuje za največ ena. Napišite rekurzivno metodo *isBalanced*() katera testira, če je drevo uravnoteženo. Metoda mora imeti časovno zahtevnost $O(n)$. (Priporočeno je, da kodo testirate na nekaj velikih drevesih z različnimi oblikami. Metodo s časovno zahtevnostjo veliko večjo od $O(n)$ je dosti lažje napisati.)

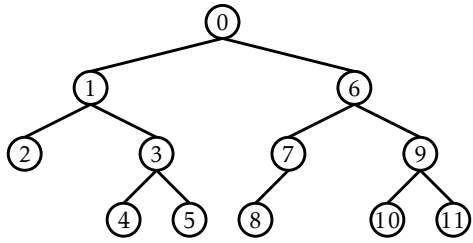
Premi pregled (pre-order) po dvojiškem drevesu je sprehod, ki obišče vsako vozlišče u pred svojimi otroci. Vmesni pregled (in-order) pogleda najprej levega otroka potem koren in nato še desnega otroka. Dobljeni vrstni red je sortiran. Obratni pregled (post-order) obišče koren u šele po tem, ko obišče vsa vozlišča v svojih poddrevesih. Glej sliko 6.10.

Vaja 6.7. Ustvarite podrazred BinaryTree, čigar vozlišča imajo polja za shranjevanje števil premega, obratnega in vmesnega pregleda. Napišite rekurzivne metode *preOrderNumber*(), *inOrderNumber*() in *postOrderNumber*(), ki ta števila pravilno dodelijo. Vse te metode morajo imeti časovno zahtevnost $O(n)$.

Vaja 6.8. Napišite nerekurzivno metode *nextPreOrder*(u), *nextInOrder*(u) in *nextPostOrder*(u), ki vračajo vozlišče katero sledi u v premem, vmesnem in obratnem pregledu. Te metode bi morale vzeti amortiziran konstanten čas. Če začnemo pri katerem koli izbranem vozlišču, ter večkrat kličemo eno izmed teh funkcij dokler u ni enak *null*, bi časovna zahtevnost morala biti $O(n)$.

Vaja 6.9. Recimo, da imamo dvojiško drevo s pre-, in- in post-order številkami dodeljenimi vozliščem. Pokažite, kako se lahko te številke uporabijo za odgovor na vsako izmed naslednjih vprašanj, v konstantnem času.

1. Ob danem vozlišču u, določite velikost poddrevesa, ki ima koren v



Slika 6.10: Pre-order, post-order, and in-order numberings of a binary tree.

u.

2. Ob danem vozlišču u, določite globino v u.

3. Ob danih dveh vozliščih u in w, določite ali je u prednik w.

Naloga 6.1. Recimo, da imamo seznam vozlišč, ki so premo in vmesno oštevilčena. Dokažite, da obstaja največ eno premo/vmesno oštevilčeno drevo in pokažite, kako ga sestavimo.

Naloga 6.2. Pokažite, da je lahko oblika kateregakoli dvojiškega drevesa z n vozlišči predstavljena z največ $2(n-1)$ biti. (Namig: poskusite zabeležiti, kaj se zgodi ob sprehodu, nato podatke uporabite za ponovno postavitev drevesa.)

Naloga 6.3. Narišite, kaj se zgodi, ko dodamo vrednosti 3.5 in nato 4.5 dvojiškemu iskalnemu drevesu v 6.5.

Naloga 6.4. Narišite, kaj se zgodi, ko odstranimo vrednosti 3 in nato 5 iz dvojiškega iskalnega drevesa v 6.5.

Naloga 6.5. Izvedite `BinarySearchTree` metodo, `getLE(x)`, ki vrne seznam vseh članov, ki so manjši ali enaki x. Čas izvajanja vaše metode, bi moral biti $O(n'+h)$, kjer je n' število članov, ki so manjši ali enaki x in h je višina drevesa.

Naloga 6.6. Opišite, kako dodamo elemente $\{1,\ldots,n\}$ prvotno praznemu `BinarySearchTree` tako, da ima končno drevo višino $n-1$. Na koliko načinov je to mogoče narediti?

Naloga 6.7. Če imamo neko `BinarySearchTree` in izvedemo operaciji `add(x)`, ki ji sledi `remove(x)` (z enako vrednostjo `x`), ali se nujno povrnemo v prvotno drevo?

Naloga 6.8. Ali lahko `remove(x)` operacija poveča višino kateregakoli vozlišča v `BinarySearchTree`? In če da, za koliko?

Naloga 6.9. Ali lahko `add(x)` operacija poveča višino kateregakoli vozlišča v `BinarySearchTree`? Ali lahko poveča višino drevesa? Če da, za koliko?

Naloga 6.10. Oblikujte in izvedite različico `BinarySearchTree`, v kateri vsako vozlišče `u`, vzdržuje vrednosti `u.size` (velikost poddrevesa, ki ima koren v vozlišču `u`), `u.depth` (globino od `u`), in `u.height` (višino poddrevesa, s korenom v `u`).

Te vrednosti vzdržujemo tudi ob klicu `add(x)` in `remove(x)` operacij, ampak to ne sme povečati ceno operacij za več kakor neko konstanto vrednost.

Poglavje 7

Naključna iskalna binarna drevesa

V tem poglavju bomo predstavili binarno iskalno strukturo, ki uporablja naključje, da doseže pričakovani čas $O(\log n)$ za vse operacije.

7.1 Naključna iskalna binarna drevesa

Premislimo o dveh binarnih iskalnih drevesih, ki sta prikazani na 7.1, od katerih ima vsak $n = 15$ vozlišč. Tista na levi strani je seznam ta druga pa je popolnoma uravnoteženo binarno iskalno drevo. Tista na levi strani ima višino $n - 1 = 14$ in tista na desni ima višino tri.

Predstavljajte si, kako bi lahko bili zgrajeni ti dve drevesi. Tista na levi se zgodi, če začnemo s praznim `BinarySearchTree` in dodamo zaporedje

$$\langle 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 \rangle \ .$$

Nobeno drugo dodatno zaporedje ne bo ustvarilo to drevo (kot lahko dokažete z indukcijo po n). Po drugi strani, pa je drevo na desni lahko ustvarjeno z zaporedjem

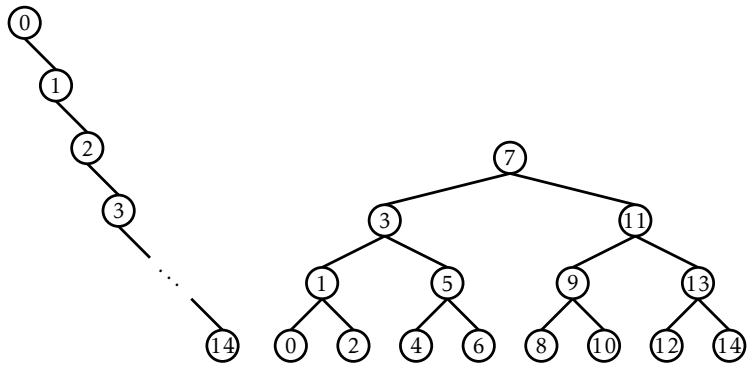
$$\langle 7, 3, 11, 1, 5, 9, 13, 0, 2, 4, 6, 8, 10, 12, 14 \rangle \ .$$

Ostala zaporedja tudi delujejo dobro, vključno z

$$\langle 7, 3, 1, 5, 0, 2, 4, 6, 11, 9, 13, 8, 10, 12, 14 \rangle \ ,$$

in

$$\langle 7, 3, 1, 11, 5, 0, 2, 4, 6, 9, 13, 8, 10, 12, 14 \rangle \ .$$



Slika 7.1: Dva binarna iskana drevesa vsebujeta cela števila $0,\ldots,14$.

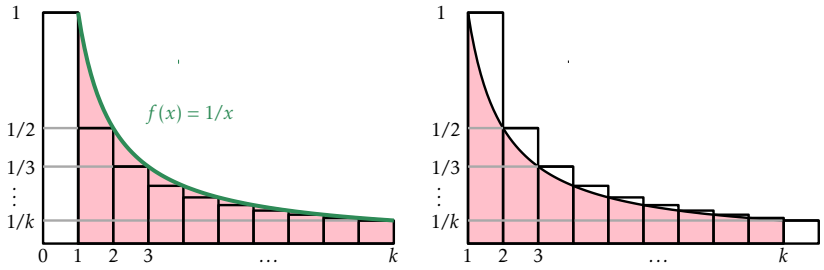
Dejstvo je, da obstaja $21,964,800$ dodatnih zaporedij, ki lahko ustvarijo drevo na desni strani in samo eno zaporedje, ki lahko ustvari drevo na levi strani.

Zgornji primer daje nekaj nezanesljivih dokazov, saj če izberemo naključno permutacijo od $0,\ldots,14$, in jo dodamo v binarno iskalno drevo, potem je bolj verjetno, da bi dobili zelo uravnoteženo drevo (na desni strani 7.1) tako lahko dobimo zelo neuravnoteženo drevo (na levi strani 7.1).

Formaliziramo to notacijo s preučevanjem naključnih binarnih iskalnih dreves. *Naključno binarno iskalno drevo* velikosti n dobimo na naslednji način: Vzamemo naključno permutacijo, $x_0,\ldots,x_{n-1}$, celih števil $0,\ldots,n-1$ in dodajamo njene elemente, enega za drugim v `BinarySearchTree`. Z *naključnimi permutacijami* mislimo, da vsaka izmed $n!$ permutacij (urejena) od $0,\ldots,n-1$ enako verjetna, tako da je verjetnost pridobitve posebne permutacije $1/n!$.

Upoštevajmo, da lahko vrednosti $0,\ldots,n-1$ nadomestimo s poljubnimi urejenim izborom n elementov brez spreminjanja nobene od lastnosti naključnega binarnega iskalnega drevesa. Element $x \in \{0,\ldots,n-1\}$ preprosto stoji za elementom ranga x v urejenem izboru velikosti n.

Preden bomo lahko predstavili naš glavni rezultat o naključnih binarnih iskalnih drevesih, si moramo vzeti nekaj časa za kratek odmik, da lahko razpravljamo o tipu števila, ki se pojavlja pogosteje pri preučevanju



Slika 7.2: k-iško harmonično število $H_k = \sum_{i=1}^{k} 1/i$ je zgoraj omejeno in spodaj omejeno z dvema integraloma. Vrednost teh integralov je podana s območjem, ki je zasenčeno, medtem, ko je vrednost H_k podana z območjem, kjer so pravokotniki.

naključnih struktur. Za nenegativno celo število, k, k-tiško *harmonično število*, označeno H_k, je definirano kot

$$H_k = 1 + 1/2 + 1/3 + \cdots + 1/k \ .$$

Harmonično število H_k nima preproste zaprte oblike, vendar je zelo tesno povezano z naravnim logaritmom od k. Zlasti,

$$\ln k < H_k \leq \ln k + 1 \ .$$

Bralci, ki so študirali računanje lahko opazijo, da je tako, ker integral $\int_1^k (1/x)\,\mathrm{d}x = \ln k$. Imejmo v mislih, da integral je lahko interpretiran kot območje med krivuljo in x-os, vrednost H_k je lahko nižje omejena z integralom $\int_1^k (1/x)\,\mathrm{d}x$ in višje omejena z $1 + \int_1^k (1/x)\,\mathrm{d}x$. (Glej 7.2 za grafično razlago.)

Lema 7.1. *V naključnem binarnem iskalnem drevesu velikosti* n*, držijo naslednje izjave:*

1. *Za vsak* $\mathsf{x} \in \{0, \ldots, \mathsf{n}-1\}$*, pričakovana dolžina iskane poti za* x *je* $H_{\mathsf{x}+1} + H_{\mathsf{n}-\mathsf{x}} - O(1)$.[1]
2. *Za vsak* $\mathsf{x} \in (-1, n) \setminus \{0, \ldots, \mathsf{n}-1\}$*, pričakovana dolžina iskane poti za* x *je* $H_{\lceil \mathsf{x} \rceil} + H_{\mathsf{n}-\lceil \mathsf{x} \rceil}$.

[1] Izraz $\mathsf{x}+1$ in $\mathsf{n}-\mathsf{x}$ si je mogoče razlagati, kot število elementov v drevesu, ki je manjše ali enako x in število elementov v drevesu, ki je večje ali enako x.

Dokazali bomo 7.1 v naslednjem poglavju. Za zdaj, upoštevajmo kaj nam povedo oba dela 7.1. Prvi del nam pove, da če iščemo element v drevesu velikosti n, potem je predvidena dolžina iskane poti največ $2\ln n + O(1)$. Drugi del nam pove, enako stvar pri iskanju za vrednsot, ki ni shranjena v drevesu. Če primerjamo oba dela Leme, vidimo, da je nekoliko hitrejše iskanje, če iščemo nekaj, kar je v drevesu v primerjavi z nečem, kar ni.

7.1.1 Dokaz 7.1

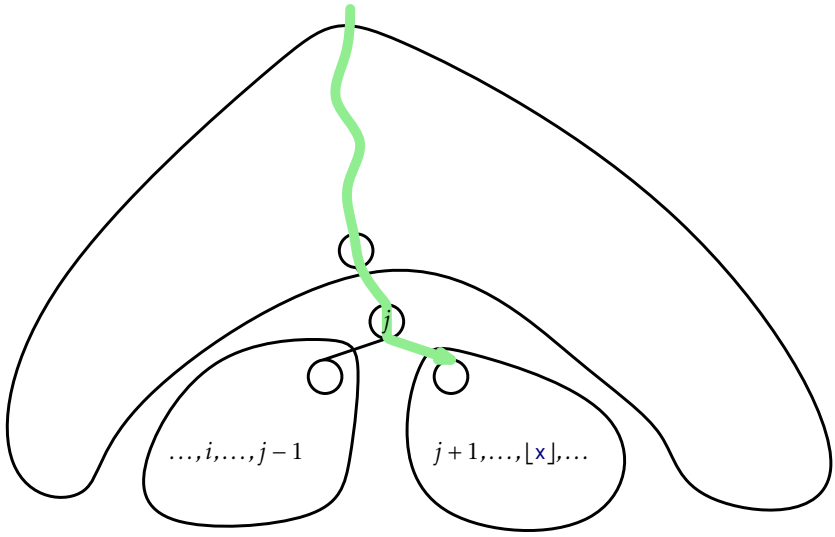
Ključna ugotovitev pri dokazovanju 7.1 je naslednja: Iskana pot za vrednost x v odprtem intervalu $(-1, n)$ v naključnem binarnem iskalnem drevesu, T, vsebuje vozlišče s ključem $i < x$ če, in samo če je naključna permutacija uporabljena za ustvarjanje T, i preden se pojavi katerakoli od $\{i+1, i+2, \ldots, \lfloor x \rfloor\}$.

Da bi to videli, se nanašamo 7.3 in lahko opazimo, da do nekaterih vrednosti v $\{i, i+1, \ldots, \lfloor x \rfloor\}$ je dodana iskana pot za vsako vrednost v oprtem intervalu $(i-1, \lfloor x \rfloor + 1)$ ter te sta enake. (Zapomnimo si to, za dve vrednosti, ki imata različne iskane poti, tu mora biti nek element v drevesu, ki je različen od obeh.) Naj bo j prvi element v $\{i, i+1, \ldots, \lfloor x \rfloor\}$, ki nastopa v naključni permutaciji. Opazimo, da j je zdaj in bo vedno v iskani poti za x. Če $j \neq i$ potem vozlišče u_j , ki vsebuje j je ustvarjeno pred vozliščem u_i , ki vsebuje i. Kasneje, ko je i dodan, bo bil dodan v korenu poddrevesa pri u_j.left, saj $i < j$. Po drugi strani iskana pot za x , ne bo nikoli obiskala poddrevo, ker bi se nadaljevala k u_j.right po obisku u_j.

Podobno za $i > x$, i se pojavi v iskalni poti za x če, in samo če i se pojavi pred katerikoli od $\{\lceil x \rceil, \lceil x \rceil + 1, \ldots, i-1\}$ v naključni permutaciji, ki uporablja za ustvarjanje T.

Opazimo, da če začnemo z naključno permutacijo od $\{0, \ldots, n\}$, potem pod-zaporedje vsebuje samo $\{i, i+1, \ldots, \lfloor x \rfloor\}$ in $\{\lceil x \rceil, \lceil x \rceil + 1, \ldots, i-1\}$ so tudi naključne permutacije njihovih pripadajočih elementov. Vsak element, potem v podmnožici $\{i, i+1, \ldots, \lfloor x \rfloor\}$ in $\{\lceil x \rceil, \lceil x \rceil + 1, \ldots, i-1\}$ je verjetno, da nastopi pred katerikoli drugim v svoji podmnožici v naključni permutaciji uporabljeni za ustvarjanje T. Torej imamo

$$\Pr\{i \text{ is on the search path for } x\} = \begin{cases} 1/(\lfloor x \rfloor - i + 1) & \text{if } i < x \\ 1/(i - \lceil x \rceil + 1) & \text{if } i > x \end{cases} .$$



Slika 7.3: Vrednost $i < x$ je na iskalni poti za x če, in samo če i je prvi element med $\{i, i+1, \ldots, \lfloor x \rfloor\}$ dodan drevesu.

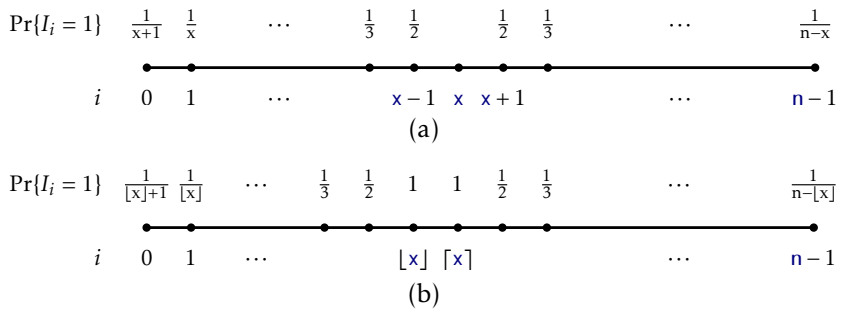
S tem opazovanjem, dokaz za 7.1 vključuje nekaj preprostih izračunov z harmonskimi števili:

Dokaz 7.1. Naj I_i bo pokazatelj naključne spremenljivke, ki je enaka ena, kadar se i pojavi na iskalni poti za x in nič sicer. Potem je dolžina iskalne poti podana z

$$\sum_{i \in \{0,\ldots,n-1\} \setminus \{x\}} I_i$$

tako da, če $x \in \{0, \ldots, n-1\}$, je pričakovana dolžina iskalne poti podana z (glej 7.4.a)

$$\begin{aligned} \mathrm{E}\left[\sum_{i=0}^{x-1} I_i + \sum_{i=x+1}^{n-1} I_i\right] &= \sum_{i=0}^{x-1} \mathrm{E}[I_i] + \sum_{i=x+1}^{n-1} \mathrm{E}[I_i] \\ &= \sum_{i=0}^{x-1} 1/(\lfloor x \rfloor - i + 1) + \sum_{i=x+1}^{n-1} 1/(i - \lceil x \rceil + 1) \\ &= \sum_{i=0}^{x-1} 1/(x - i + 1) + \sum_{i=x+1}^{n-1} 1/(i - x + 1) \end{aligned}$$



Slika 7.4: Verjetnost, da je element na iskalni poti za x kadar (a) x je celo število in (b) kadar x ni celo število.

$$
\begin{aligned}
&= \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{x+1} \\
&\quad + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-x} \\
&= H_{x+1} + H_{n-x} - 2 \ .
\end{aligned}
$$

Ustrezen izračun za iskalno vrednost $x \in (-1, n) \setminus \{0, \ldots, n-1\}$ so skoraj enake (glej 7.4.b). □

7.1.2 Povzetek

Spodnji teorem povzame učinkovitost naključnega binarnega iskalnega drevesa:

Izrek 7.1. *Naključno binarno iskalno drevo lahko ustvarimo v $O(n \log n)$ času. V naključnem binarne iskalnem drevesu,* `find`(x) *operacija potrebuje $O(\log n)$ predvidenega časa.*

Ponovno moramo poudariti, da pričakovanja v 7.1 je v zvezi z naključno permutacijo uporabljena za ustvarjanje naključnega binarnega iskalnega drevesa. Predvsem, pa ni odvisno od naključne izbire x; , saj je pravilna za vsako x vrednost x.

7.2 Treap: Naključno generirano binarno iskalno drevo

Problem naključnih binarno iskalnih dreves je seveda, da niso dinamična. Ta drevesa ne podpirajo add(x) ali remove(x) operacij, ki so potrebne za implementacijo SSet vmesnika. V tem poglavju bomo opisali podatkovno strukturo, imenovano Treap, ki uporablja 7.1 za implementacijo SSet vmesnika. [2]

Vozlišče v Treap je kot vozlišče v BinarySearchTree s tem, da ima podatkovno vrednost, x, toda vsebuje tudi edinstveno številčno *prioriteto*,p, ki je dodeljena naključno:

Treap

```
class TreapNode : public BSTNode<Node, T> {
  friend class Treap<Node,T>;
  int p;
};
```

Poleg tega, da je binarno iskalno drevo, vozlišča v Treap prav tako ubogajo *lastnostim kopice*:

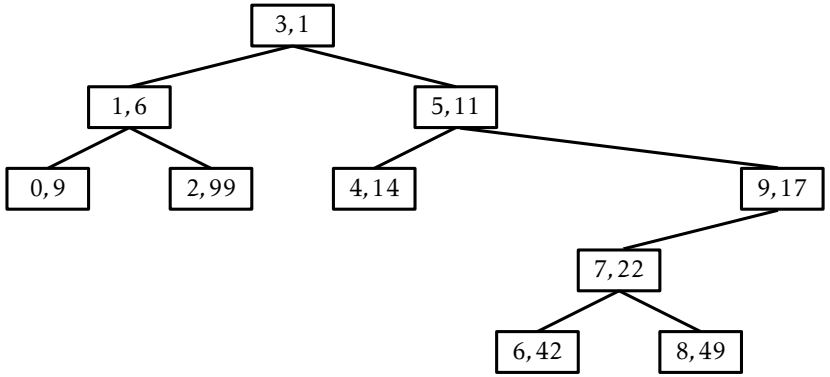
- (Lastnosti kopice) Pri vsakem vozlišču u, razen pri korenu, u.parent.p < u.p.

Z drugimi besedami, vsako vozlišče ima prioriteto manjšo od svojih dveh otrok. Primer je prikazan na 7.5.

Pogoji kopice in binarno iskalnega drevesa skupaj zagotavljajo, da enkrat ko so ključ (x) in prioriteta (p) definirane za vsako vozlišče, je oblika drevesa Treap popolnoma določena. Lastnost kopice nam pove, da vozlišče z najmanjšo prioriteto mora biti koren, r, drevesa Treap. Lastnost binarno iskalnega drevesa nam pove, da vsa vozlišča s ključem manjšim od r.x so shranjene v poddrevesu, ki je zasidran na r.left in vsa vozlišča s ključem večjim od r.x so shranjene v poddrevesu, ki je zasidran na r.right.

Pomembna točka o vrednosti prioritete v drevesu Treap je, da so edinstveni id dodeljeni naključno. Zaradi tega obstajajo dva enakovredna načina razmišljanja o drevesu Treap. Kot je definirano zgoraj, drevo Treap

[2] Ime Treap izhaja iz dejstva, da je podatkovna struktura , hkrati binarno iskalno drevo (**tr**ee) (6.2) in kopice(h**eap**)(**??**).



Slika 7.5: Primer drevesa Treap, ki vsebuje cela števila $0,\ldots,9$. Vsako vozlišče, u, je prikazano kot škatla, ki vsebuje u.x, u.p.

uboga lastnostim kopice in binarno iskalnega drevesa. Alternativno lahko razmišljamo o drevesu Treap kot o BinarySearchTree katerega vozlišča so bila dodana v naraščajočem vrstnem redu prioritete. Na primer drevo Treap na 7.5 ga lahko dobimo z dodajanjem zaporedja (x, p) vrednosti

$$\langle (3,1),(1,6),(0,9),(5,11),(4,14),(9,17),(7,22),(6,42),(8,49),(2,99)\rangle$$

v BinarySearchTree.

Ker so prioritete izbrane naključno, je to enako, če vzamemo naključno permutacijo ključev—v tem primeru permutacija je

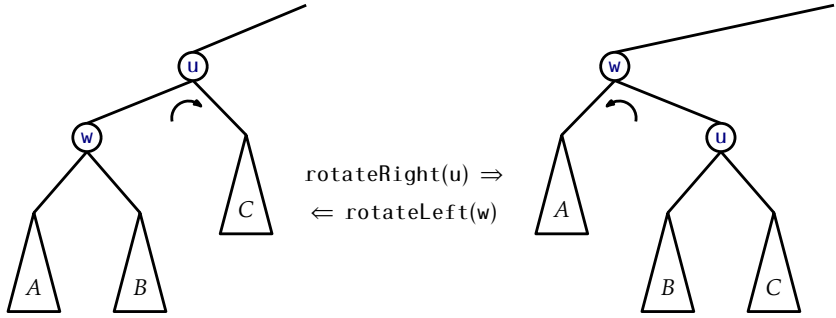
$$\langle 3,1,0,5,9,4,7,6,8,2\rangle$$

—in jo dodamo v BinarySearchTree. To pa pomeni, da je oblika treap drevesa identična obliki naključnega binarno iskalnega drevesa. Še posebej, če želimo zamenjati vsak ključ x z njegovim rangom[3], potem se aplicira 7.1. Preračunavanju lemref rbs glede na drevesa Treap, imamo:

Lema 7.2. *V drevesu Treap, ki shranjuje niz S z n ključi, naslednje izjave držijo:*

1. *Za vsak $x \in S$, pričakovana dolžina iskanja poti za x je $H_{r(x)+1}+H_{n-r(x)}-O(1)$.*

[3] Rang elementa x v nizu S elementov je število elementov v S, ki so manjši kot x.



Slika 7.6: Leva in desna rotacija v binarno iskalnem drevesu.

2. *Za vsak* $x \notin S$, *pričakovana dolžina iskanja poti za* x *je* $H_{r(x)} + H_{n-r(x)}$.

Tukaj, $r(x)$ *označuje rang* x *v nizu* $S \cup \{x\}$.

Ponovno poudarimo, da se pričakovanje pri 7.2 prevzemajo preko naključne izbire prioritet za vsako vozlišče. To ne potrebuje nobene predpostavke o naključju ključev.

7.2 nam pove, da lahko `Treap` drevesom učinkovito implementiramo `find(x)` operacijo. Vendar, resnična korist `Treap` dreves je, da lahko podpre operacije `add(x)` in `delete(x)`. Za narediti to, mora izvajati rotacije, tako da ohrani lastnosti kopice. Nanaša se na figrefrotations. *Rotacija* v binarno iskalnih drevesih je lokalna sprememba, ki vzame starša `u` vozlišča `w` in naredi, da je `w` starš od `u`, medtem ko ohranjuje lastnosti binarno iskalnega drevesa. Rotacije pridejo v dveh okusih: emph levo ali emph desno glede na to, ali je `w` desni ali levi otrok od `u`.

Koda, ki implementira to mora ravnati z tema dvema možnostma in mora biti pozorna na mejne primere (ko je `u` koren), tako da je dejanska koda malo daljša kot 7.6 bi vodila bralca, da verjame:

BinarySearchTree

```
void rotateLeft(Node *u) {
  Node *w = u->right;
  w->parent = u->parent;
  if (w->parent != nil) {
    if (w->parent->left == u) {
      w->parent->left = w;
    } else {
```

```
      w->parent->right = w;
    }
  }
  u->right = w->left;
  if (u->right != nil) {
    u->right->parent = u;
  }
  u->parent = w;
  w->left = u;
  if (u == r) { r = w; r->parent = nil; }
}
void rotateRight(Node *u) {
  Node *w = u->left;
  w->parent = u->parent;
  if (w->parent != nil) {
    if (w->parent->left == u) {
      w->parent->left = w;
    } else {
      w->parent->right = w;
    }
  }
  u->left = w->right;
  if (u->left != nil) {
    u->left->parent = u;
  }
  u->parent = w;
  w->right = u;
  if (u == r) { r = w; r->parent = nil; }
}
```

V zvezi s podatkovno strukturo Treap je najpomembnejša lastnost rotacije, da se globina od w zmanjša za ena, medtem ko se globina u poveča za ena.

Z uporabo rotacij, lahko implementiramo operacijo add(x), kakor sledi: ustvarimo novo vozlišče, u, dodelimo $u.x = x$, in izberemo naključno vrednost za $u.p$. Nato dodamo u z uporabo običajnega add(x) algoritma za BinarySearchTree , tako da je u zdaj list Treap drevesa. Na tej točki, naše Treap drevo izpolnjuje lastnosti binarno iskalnega drevesa, vendar pa ni nujno, da izpolnjuje lastnosti kopice. Zlasti se lahko zgodi, da $u.\text{parent}.p > u.p$. Če se to zgodi, moramo izvesti rotacijo na vozlišču

w=u.parent, tako da u postane starš w. Če u še naprej krši lastnosti kopice, bomo morali ponoviti to, zmanjšuje globino u-ja za ena vsakič, dokler u ne postane koren ali u.parent.p < u.p.

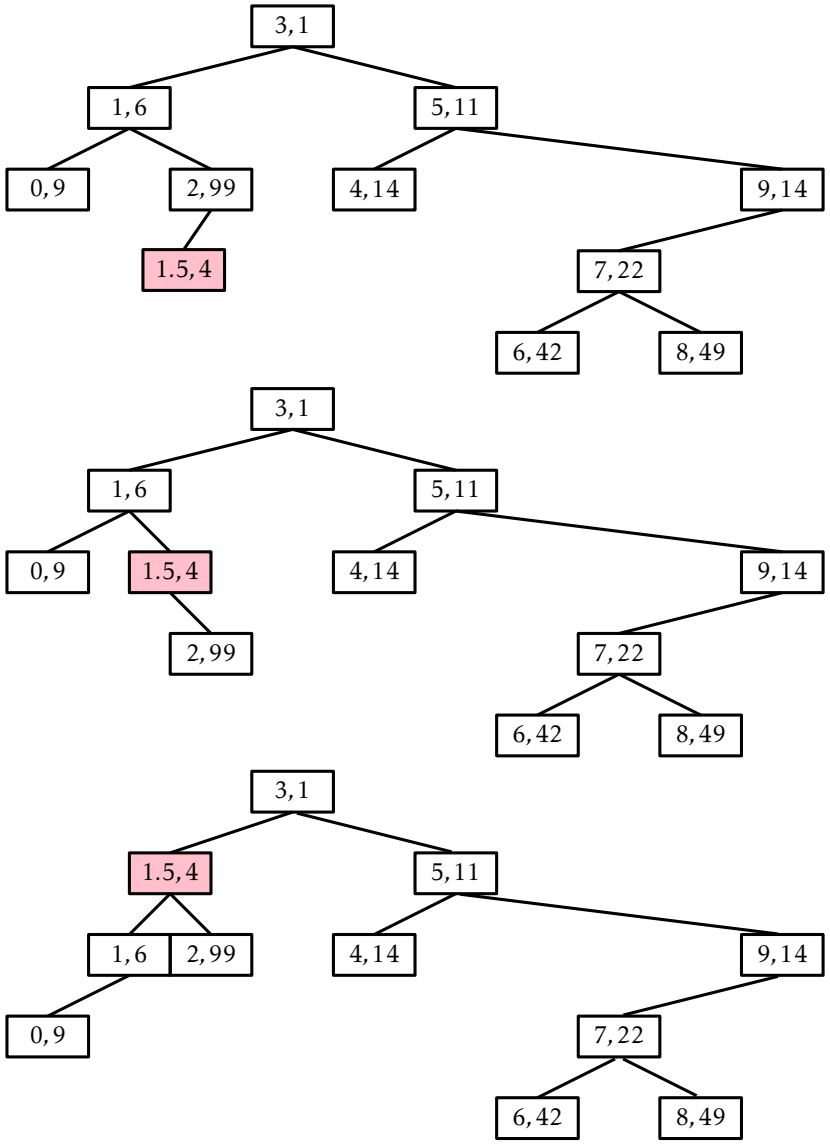
Treap

```
bool add(T x) {
  Node *u = new Node;
  u->x = x;
  u->p = rand();
  if (BinarySearchTree<Node,T>::add(u)) {
    bubbleUp(u);
    return true;
  }
  return false;
}
void bubbleUp(Node *u) {
  while (u->parent != nil && u->parent->p > u->p) {
    if (u->parent->right == u) {
      rotateLeft(u->parent);
    } else {
      rotateRight(u->parent);
    }
  }
  if (u->parent == nil) {
    r = u;
  }
}
```

Primer add(x) operacije je prikazana na 7.7.

Čas izvajanja operacije add(x) je podan s časom, ki je potreben, za slediti iskalni poti do x plus število vrtljajev, ki so bili opravljeni za premik novo dodanega vozlišča, u, do njegove prave lokacije v drevesu Treap. Z 7.2 je pričakovano trajanje iskalne poti maksimalno $2\ lnn + O(1)$. Poleg tega, vsaka rotacija zmanjša globino u. To se ustavi, če u postane koren, tako da pričakovano število rotacij ne sme preseči predvidene dolžine iskalne poti. Zato je pričakovani čas izvajanja operacije add(x) v drevesu Treap, $O(\log n)$. (**??** sprašuje po dokazu, da je pričakovano število opravljenih rotacij v času dodajanja samo $O(1)$.)

Operacija remove(x) v drevesu Treap je nasprotna operaciji add(x). Iščemo vozlišče, u, ki vsebuje x, nato izvedemo rotacije za premakniti



Slika 7.7: Dodajamo vrednost 1.5 v Treap drevo iz 7.5.

u navzdol, dokler ne postane list in potem spojimo u iz Treap drevesa. Opazite, da za premikanje u navzdol, lahko opravljamo bodisi levo bodisi desno rotacijo na u, ki bo nadomestila u z u.right ali u.left. Izbira je opravljena s prvim od naslednjih, ki velja:

1. Če u.left in u.right sta null, potem u je list in rotacija ni bila izvedena.

2. Če u.left (ali u.right) je null, potem izvedi desno (oz. levo) rotacijo na u.

3. Če u.left.p < u.right.p (ali u.left.p > u.right.p), potem izvedi desno rotacijo (oz. levo rotacijo) na u.

Ta tri pravila zagotavljajo, da drevo Treap ne postane nepovezano in da se lastnosti kopice obnovijo, ko je u odstranjen.

Treap

```
bool remove(T x) {
  Node *u = findLast(x);
  if (u != nil && compare(u->x, x) == 0) {
    trickleDown(u);
    splice(u);
    delete u;
    return true;
  }
  return false;
}
void trickleDown(Node *u) {
  while (u->left != nil || u->right != nil) {
    if (u->left == nil) {
      rotateLeft(u);
    } else if (u->right == nil) {
      rotateRight(u);
    } else if (u->left->p < u->right->p) {
      rotateRight(u);
    } else {
      rotateLeft(u);
    }
    if (r == u) {
      r = u->parent;
    }
```

```
    }
}
```

Primer operacije remove(x) je prikazan na 7.8.

Trik za analizirati čas izvajanja operacije remove(x) je opaziti, da operacija obrne operacijo add(x). Še posebej, če bi ponovno vstavili x z uporabo iste prioritete u.p, potem bi operacija add(x) naredila popolnoma enako število rotacij in bi obnovila drevo Treap kot je bilo pred potekom operacije remove(x). (Branje iz dna do vrha, 7.8 prikazuje dodajanje vrednosti 9 v drevo Treap.) To pomeni, da je pričakovan čas izvajanja remove(x) na drevesu Treap z velikostjo n je sorazmeren s pričakovanim časom izvajanja operacije add(x) na drevesu Treap, ki je velikosti $n-1$. Zaključujemo tako, da je pričakovani čas izvajanja remove(x) $O(\log n)$

7.2.1 Povzetek

Naslednji izrek povzema zmogljivosti podatkovne strukture Treap:

Izrek 7.2. *Treap implementira vmesnik SSet. Treap podpira operacije* add(x), remove(x) *in* find(x) *v pričakovanem času* $O(\log n)$ *za vsako operacijo.*

To je vredno primerjave podatkovne strukture Treap s podatkovno strukturo SkiplistSSet. Obe implementirata operacije SSet v predvidenem času $O(\log n)$ za vsako operacijo. V obeh podatkovnih strukturah, add(x) in remove(x) vključujeta iskanje in nato konstantno število sprememb kazalca (glej **??** spodaj). Tako je za obe strukturi, pričakovana dolžina iskalne poti je kritična vrednost pri ocenjevanju njihove uspešnosti. V SkiplistSSet, pričakovana dolžina iskalne poti je

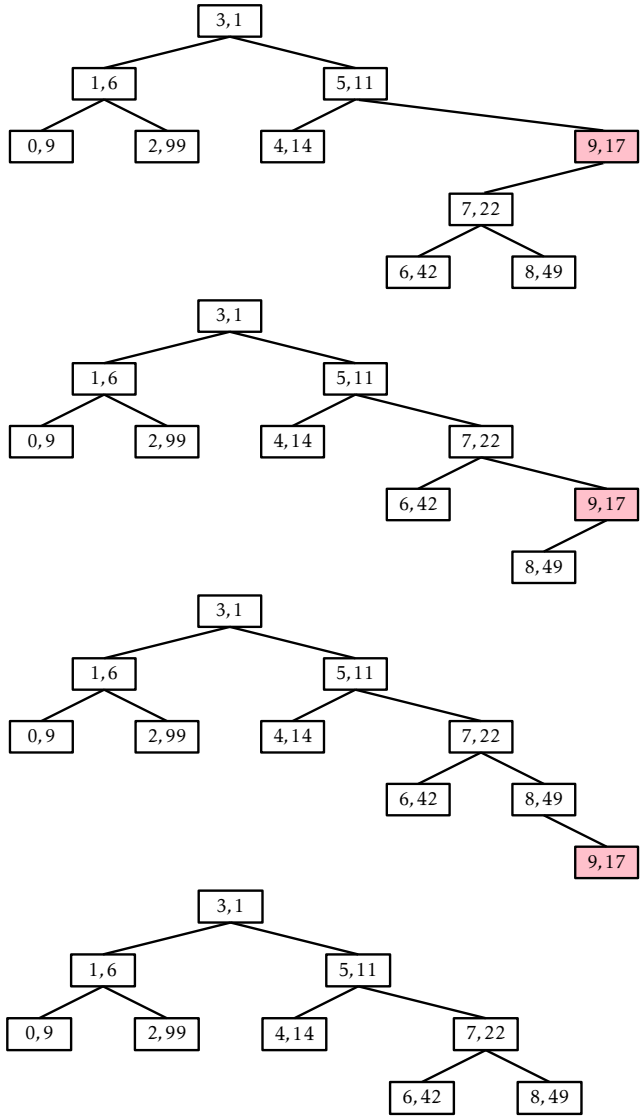
$$2\log n + O(1) \ ,$$

V Treap, pričakovana dolžina iskalne poti je

$$2\ln n + O(1) \approx 1.386\log n + O(1) \ .$$

Tako je iskanje poti v Treap precej krajše in to se prevede v občutno hitrejše operacije nad Treap drevesih kot nad Skiplist. 4.7 v 4 prikazuje, kako se lahko pričakovana dolžina iskalne poti v Skiplist zmanjša na

$$e\ln n + O(1) \approx 1.884\log n + O(1)$$



Slika 7.8: Brišemo vrednost 9 iz drevesa Treap na 7.5.

z uporabo pristranskega meta kovanca. Tudi s to optimizacijo, pričakovana trajanje iskanja poti v `SkiplistSSet` je občutno daljše kot v `Treap`.

Poglavje 8

Drevesa "grešnega kozla"

V tem poglavju bomo preučili podatkovno strukturo dvojiškega iskalnega drevesa, `ScapegoatTree`. Struktura temelji na znanem dejstvu, da, ko gre nekaj narobe, ljudje najprej nekoga okrivijo (*grešni kozel*). Ko najdemo grešnega kozla, lahko ves problem prepustimo njemu.

`ScapegoatTree` ohranja ravnotežje z *operacijami delne rekonstrukcije*. Med delno rekonstrukcijo se celotno poddrevo razstavi in zgradi nazaj v popolnoma uravnoteženo poddrevo. Obstaja mnogo načinov, kako spremeniti drevo s korenom v vozlišču `u` v popolnoma uravnoteženo drevo. Eden od najpreprostejših je, da se sprehodimo čez poddrevo `u` in zberemo vsa vozlišča v tabelo `a`, nato pa iz te tabele rekurzivno zgradimo uravnoteženo poddrevo. Če je $m = \texttt{a.length}/2$, potem je element novi `a`$[m]$ koren poddrevesa, elementi $\texttt{a}[0],\ldots,\texttt{a}[m-1]$ se shranijo rekurzivno v levo poddrevo in $\texttt{a}[m+1],\ldots,\texttt{a}[\texttt{a.length}-1]$ se shranijo rekurzivno v desno poddrevo.

ScapegoatTree

```
void rebuild(Node *u) {
  int ns = BinaryTree<Node>::size(u);
  Node *p = u->parent;
  Node **a = new Node*[ns];
  packIntoArray(u, a, 0);
  if (p == nil) {
    r = buildBalanced(a, 0, ns);
    r->parent = nil;
  } else if (p->right == u) {
    p->right = buildBalanced(a, 0, ns);
    p->right->parent = p;
```

```
  } else {
    p->left = buildBalanced(a, 0, ns);
    p->left->parent = p;
  }
  delete[] a;
}
int packIntoArray(Node *u, Node **a, int i) {
  if (u == nil) {
    return i;
  }
  i = packIntoArray(u->left, a, i);
  a[i++] = u;
  return packIntoArray(u->right, a, i);
}
```

En klic `rebuild(u)` traja $O(\texttt{size}(u))$. Popravljeno poddrevo je minimalne velikosti; ni možno izgraditi nižjega drevesa s `size(u)` vozlišči.

8.1 ScapegoatTree: Dvojiško iskalno drevo z delno rekonstrukcijo

`ScapegoatTree` je `BinarySearchTree`, ki poleg števca (`n`) vozlišč v drevesu hrani še števec (`q`), ki drži zgornjo mejo dovoljenega števila vozlišč.

ScapegoatTree

```
int q;
```

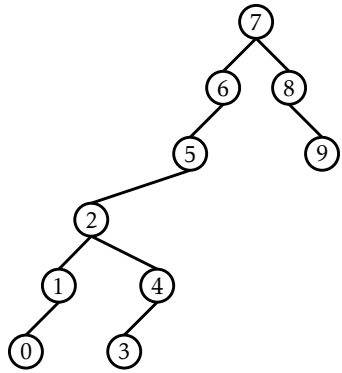
$$q/2 \le n \le q \ .$$

Poleg tega ima `ScapegoatTree` logaritmično višino; njegova višina nikoli ne prekorači

$$\log_{3/2} q \le \log_{3/2} 2n < \log_{3/2} n + 2 \ . \tag{8.1}$$

Tudi s to omejitvijo lahko `ScapegoatTree` izgleda presenetljivo neuravnoteženo. Drevo na sliki 8.1 ima $q = n = 10$ in višino $5 < \log_{3/2} 10 \approx 5.679$.

Implementacija `find(x)` je v `ScapegoatTree` narejena s standardnim algoritmom za iskanje v `BinarySearchTree` (glej 6.2). Njena časovna zahtevnost je sorazmerna z višino drevesa, ki je po (8.1) enaka $O(\log n)$.



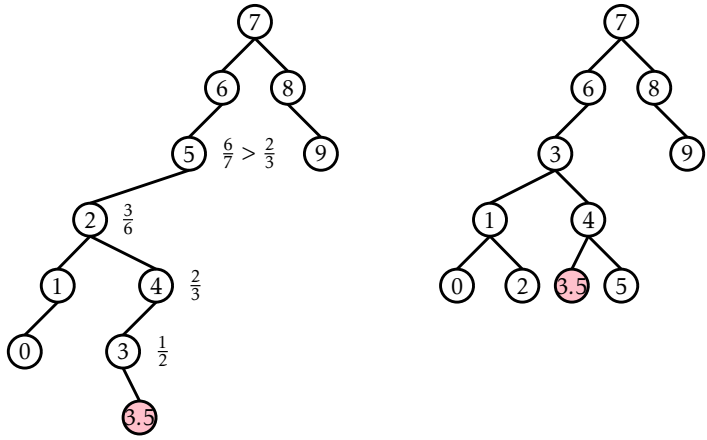
Slika 8.1: ScapegoatTree z 10 vozlišči in višino 5.

Ko implementiramo operacijo add(x), najprej povečamo n in q in potem uporabimo običajni algoritem za dodajanje x v dvojiško iskalno drevo; poiščemo x in nato dodamo nov list u z u.x = x. Tu se nam lahko posreči in globina u ne preseže $\log_{3/2} q$. V tem primeru smo zadovoljni z rezultatom in ne naredimo nič drugega.

Žal se včasih zgodi, da depth(u) > $\log_{3/2} q$. V tem primeru moramo višino zmanjšati. To pa ni velik zalogaj, saj imamo le eno vozlišče, u, katerega globina presega $\log_{3/2} q$. Da popravimo u, se sprehodimo nazaj proti korenu in iščemo *grešnega kozla*, w. Ta grešni kozel, w, je zelo neuravnoteženo vozlišče z lastnostjo

$$\frac{\texttt{size(w.child)}}{\texttt{size(w)}} > \frac{2}{3} \ , \tag{8.2}$$

kjer w.child predstavlja otroka w na poti od korena do u. Kmalu bomo dokazali, da grešni kozel obstaja, za zdaj pa to predpostavimo. Ko smo našli grešnega kozla w, popolnoma uničimo poddrevo s korenom v w in ga ponovno izgradimo kot popolnoma uravnoteženo dvojiško iskalno drevo. Iz (8.2) vemo, da že pred vstavljanjem u, poddrevo w ni bilo polno dvojiško drevo. Zato se ob ponovni izgradnji poddrevesa w njegova višina zniža za vsaj 1, tako da je višina celotnega drevesa spet kvečjemu $\log_{3/2} q$.



Slika 8.2: Vstavljanje 3.5 v ScapegoatTree poveča njegovo višino v 6, kar krši (8.1), saj je $6 > \log_{3/2} 11 \approx 5.914$. Grešni kozel je drevo z elementom 5.

ScapegoatTree

```
bool add(T x) {
  // first do basic insertion keeping track of depth
  Node *u = new Node;
  u->x = x;
  u->left = u->right = u->parent = nil;
  int d = addWithDepth(u);
  if (d > log32(q)) {
    // depth exceeded, find scapegoat
    Node *w = u->parent;
    int a = BinaryTree<Node>::size(w);
    int b = BinaryTree<Node>::size(w->parent);
    while (3*a <= 2*b) {
      w = w->parent;
      a = BinaryTree<Node>::size(w);
      b = BinaryTree<Node>::size(w->parent);
    }
    rebuild(w->parent);
  }
  return d >= 0;
}
```

Če ne upoštevamo cene iskanja grešnega kozla w in ponovne izgradnje poddrevesa s korenom v w, je čas za izvedbo add(x) odvisen od začetnega iskanja, ki traja $O(\log q) = O(\log n)$. Ceno iskanja grešnega kozla in rekonstrukcije bomo izračunali z amortizacijsko analizo v naslednji sekciji.

Implementacija remove(x) v ScapegoatTree je zelo preprosta. Poiščemo element x in ga odstranimo z običajnim algoritmom za odstranjevanje vozlišča iz BinarySearchTree. (To nikoli ne poveča višine drevesa.) V naslednjem koraku znižamo n, q pa pustimo nespremenjen. Na koncu preverimo, če je $q > 2n$ in, če je, *ponovno zgradimo celotno drevo* v popolnoma uravnoteženo dvojiško iskalno drevo in nastavimo $q = n$.

ScapegoatTree

```
bool remove(T x) {
  if (BinarySearchTree<Node,T>::remove(x)) {
    if (2*n < q) {
      rebuild(r);
      q = n;
    }
    return true;
  }
  return false;
}
```

Če zanemarimo ceno rekonstrukcije, je čas izvajanja remove(x) spet sorazmeren z višino drevesa, torej je enak $O(\log n)$.

8.1.1 Analiza pravilnosti in časovne kompleksnosti

V tej sekciji bomo analizirali pravilnost in amortiziran čas izvajanja operacij na ScapegoatTree. Najprej dokažimo pravilnost tako, da pokažemo, da ko operacija add(x) naredi vozlišče, ki krši pogoj (8.1), vedno lahko najdemo grešnega kozla:

Lema 8.1. *Naj bo u vozlišče višine $h > \log_{3/2} q$ v ScapegoatTree. Potem obstaja vozlišče w na poti od u do korena, za katerega drži*

$$\frac{\texttt{size}(w)}{\texttt{size}(\texttt{parent}(w))} > 2/3 \ .$$

Dokaz. Uporabili bomo dokaz s protislovjem. Predpostavimo, da lema ne

drži in je

$$\frac{\texttt{size}(\texttt{w})}{\texttt{size}(\texttt{parent}(\texttt{w}))} \leq 2/3 \ .$$

za vsa vozlišča w na poti od u do korena. Označimo pot od korena do u kot $\texttt{r} = \texttt{u}_0, \ldots, \texttt{u}_h = \texttt{u}$. Potem drži $\texttt{size}(\texttt{u}_0) = \texttt{n}$, $\texttt{size}(\texttt{u}_1) \leq \frac{2}{3}\texttt{n}$, $\texttt{size}(\texttt{u}_2) \leq \frac{4}{9}\texttt{n}$ in bolj v splošnem,

$$\texttt{size}(\texttt{u}_i) \leq \left(\frac{2}{3}\right)^i \texttt{n} \ .$$

A to nas pripelje to protislovja, saj je $\texttt{size}(\texttt{u}) \geq 1$, torej drži

$$1 \leq \texttt{size}(\texttt{u}) \leq \left(\frac{2}{3}\right)^h \texttt{n} < \left(\frac{2}{3}\right)^{\log_{3/2} \texttt{q}} \texttt{n} \leq \left(\frac{2}{3}\right)^{\log_{3/2} \texttt{n}} \texttt{n} = \left(\frac{1}{\texttt{n}}\right)\texttt{n} = 1 \ . \qquad \square$$

Sedaj analiziramo še dele algoritma, ki jih prej nismo upoštevali. Ostala sta dva dela: cena klicev size(u), ko iščemo grešne kozle in cena rebuild(w), ko najdemo grešnega kozla w. Cena klicev size(u) je povezana s ceno klicev rebuild(w) na sledeč način:

Lema 8.2. *Med klicem* add(x) *v* ScapegoatTree *je cena iskanja grešnega kozla* w *in rekonstrukcije poddrevesa s korenom v* w *enaka* $O(\texttt{size}(\texttt{w}))$.

Dokaz. Cena rekonstrukcije vozlišča w, ko ga najdemo, je $O(\texttt{size}(\texttt{w}))$. Ko iščemo grešnega kozla, kličemo size(u) na zaporedju vozlišč $\texttt{u}_0, \ldots, \texttt{u}_k$, dokler ne najdemo grešnega kozla $\texttt{u}_k = \texttt{w}$. A ker je $\texttt{u}_k$ prvo vozlišče v tem zaporedju, ki je grešni kozel, vemo, da

$$\texttt{size}(\texttt{u}_i) < \frac{2}{3}\texttt{size}(\texttt{u}_{i+1})$$

za vse $i \in \{0, \ldots, k-2\}$. Torej je cena vseh klicev $\texttt{size}(\texttt{u})$ enaka

$$\begin{aligned}
O\left(\sum_{i=0}^{k} \texttt{size}(\texttt{u}_{k-i})\right) &= O\left(\texttt{size}(\texttt{u}_k) + \sum_{i=0}^{k-1} \texttt{size}(\texttt{u}_{k-i-1})\right) \\
&= O\left(\texttt{size}(\texttt{u}_k) + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i \texttt{size}(\texttt{u}_k)\right) \\
&= O\left(\texttt{size}(\texttt{u}_k)\left(1 + \sum_{i=0}^{k-1} \left(\frac{2}{3}\right)^i\right)\right) \\
&= O(\texttt{size}(\texttt{u}_k)) = O(\texttt{size}(\texttt{w})) \ ,
\end{aligned}$$

kjer zadnja vrstica sledi iz dejstva, da je vsota geometrično padajoča vrsta. □

Ostane nam le še, da dokažemo zgornjo mejo cene vseh klicov `rebuild(u)` med zaporedjem m operacij:

Lema 8.3. *Če začnemo s praznim* `ScapegoatTree`*, vsako zaporedje m operacij* `add(x)` *in* `remove(x)` *zahteva kvečjemu $O(m \log m)$ časa za* `rebuild(u)` *operacije.*

Dokaz. Da to dokažemo, bomo uporabili *kreditno shemo*. Predstavljajmo si, da ima vozlišče neko količino kreditov. Z vsakim kreditom lahko za rekonstrukcijo plačamo neko konstantno število, `c`, enot časa. Ta shema nam skupaj da $O(m \log m)$ kreditov in vsak klic `rebuild(u)` plačamo s krediti, ki jih ima `u`. Med vstavljanjem ali izbrisom damo en kredit vsakemu vozlišču na poti do vstavljenega ali izbrisanega vozlišča `u`. Na ta način podelimo največ $\log_{3/2} q \le \log_{3/2} m$ kreditov na operacijo. Za vsako operacijo izbrisa damo še en dodaten kredit "na stran." Skupaj torej podelimo kvečjemu $O(m \log m)$ kreditov. Dokazati moramo le še, da jih imamo dovolj, da plačamo vse klice `rebuild(u)`.

Če kličemo `rebuild(u)` med vstavljanjem, je to zato, ker je `u` grešni kozel. Zamislimo si, da drži

$$\frac{\texttt{size(u.left)}}{\texttt{size(u)}} > \frac{2}{3} \ .$$

Če uporabimo dejstvo, da

$$\texttt{size(u)} = 1 + \texttt{size(u.left)} + \texttt{size(u.right)}$$

pridemo do sklepa

$$\frac{1}{2}\texttt{size(u.left)} > \texttt{size(u.right)}$$

in torej

$$\texttt{size(u.left)} - \texttt{size(u.right)} > \frac{1}{2}\texttt{size(u.left)} > \frac{1}{3}\texttt{size(u)} \ .$$

Zadnjič, ko je bilo neko poddrevo, ki vsebuje `u`, ponovno zgrajeno (če se to nikoli ni zgodilo, pa takrat, ko je bil `u` vstavljen), je držalo

$$\texttt{size(u.left)} - \texttt{size(u.right)} \le 1 \ .$$

Zato je število add(x) in remove(x) operacij, ki so vplivale na u.left ali u.right od takrat enako ali večje

$$\frac{1}{3}\texttt{size}(\texttt{u}) - 1 \ .$$

Zato je v u vsaj toliko kreditov in z njimi lahko plačamo ceno $O(\texttt{size}(\texttt{u}))$, ki jo zahteva rebuild(u).

Če kličemo rebuild(u) med izbrisom, je to zato, ker $\texttt{q} > 2\texttt{n}$. V tem primeru smo med izbrisi že dali $\texttt{q} - \texttt{n} > \texttt{n}$ kreditov "na stran" in z njimi lahko plačamo $O(\texttt{n})$ ceno, potrebno za rekonstrukcijo korena. S tem je dokaz zaključen. □

8.1.2 Povzetek

Sledeči izrek povzame učinkovitost podatkovne strukture ScapegoatTree:

Izrek 8.1. *ScapegoatTree implementira vmesnik SSet. Če zanemarimo ceno* rebuild(u) *operacij, ScapegoatTree podpira operacije* add(x), remove(x) *in* find(x) *v času* $O(\log \texttt{n})$ *na operacijo.*

Poleg tega, če začnemo s praznim ScapegoatTree, poljubno zaporedje m add(x) *in* remove(x) *operacij zahteva kvečjemu* $O(m \log m)$ *časa za klice* rebuild(u).

Poglavje 9

Rdeče-Črna Drevesa

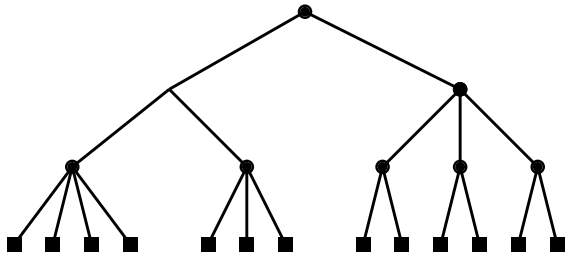
V tem poglavju so predstavljena rdeče-črna drevesa. Le ta so zasnovana kot uravnotežena iskalna dvojiška drevesa z logaritemsko višino. So ena najbolj razširjenih podatkovnih struktur in se pojavljajo kot primarne iskalne strukture v mnogih knjižnicah, kot je Java Collections Framework, številnih implementacijah C++ Standard Template Library ter tudi znotraj jedra operacijskega sistema Linux. Nekaj glavnih razlogov zakaj so rdeče-črna drevesa tako priljubljena:

1. Največja možna višina rdeče-črnega drevesa z n vozlišči je enaka $2\log n$.

2. Časovna zahtevnost operacij `add(x)` in `remove(x)` je enaka $O(\log n)$ *v najslabšem primeru*.

3. Amortizirano število rotacij, ki nastopijo med izvajanjem operacij `add(x)` ali `remove(x)` je konstantno.

Že prvi dve lastnosti postavljajo rdeče-črna drevesa pred preskočne sezname, naključna iskalna binarna drevesa in samouravnotežena binarna drevesa. Preskočni seznami in naključna iskalna binarna drevesa se zanašajo na naključje, njihova pričakovana časovna zahtevnost je $O(\log n)$. Samouravnotežena binarna drevesa imajo zagotovljeno omejitev višine, vendar se `add(x)` in `remove(x)` izvršita v $O(\log n)$ amortiziranem času. Tretja lastnost je le pika na i. Pove nam, da je čas potreben za vstavitev ali izločitev elementa x manjši od časa, ki ga porabimo za iskanje elementa



Slika 9.1: 2-4 drevo višine 3.

x.[1]

Vendar pa imajo dobre lastnosti rdeče-črnih dreves določeno ceno: kompleksnost implementacije. Ohranjati mejo višine $2 \log n$ ni preprosto. Zahteva pazljivo in podrobno analizo številnih primerov. Zagotoviti moramo, da implementacija naredi natančno določeno stvar za določen primer. Že samo ena napačna rotacija ali zamenjava barve povzroči napako, ki jo je težko najti in razumeti.

Preden se bomo lotili implementacije rdeče-črnih dreves, bomo spoznali ozadje sorodne podatkovne strukture: 2-4 drevesa. S tem bomo pridobili informacije na podlagi česa so bila rdeče-črna drevesa ustvarjena in kako jih je možno tako učinkovito ohranjati.

9.1 2-4 Trees

2-4 Drevo je korensko drevo, ki ima naslednje lastnosti:

Lastnost 9.1 (height)**.** Vsi listi imajo enako globino.

Lastnost 9.2 (degree)**.** Vsako notranje vozlišče ima 2, 3 ali 4 otroke.

Primer 2-4 drevesa je prikazan v 9.1. Lastnost 2-4 dreves je logaritemska višina v številu listov:

Lema 9.1. *Najvišja višina 2-4 drevesa z* n *listi je* $\log n$.

[1] Naključna iskalna binarna drevesa in samouravnotežena binarna drevesa imajo enako lastnost. Glej vaje 4.6 in **??**.

Dokaz. Omejenost vsakega notranjega vozlišča na najmanj 2 otroka dokakzuje, da imamo v primeru višine h v 2-4 drevesu vsaj 2^h listov. Z drugimi besedami,

$$n \geq 2^h \ .$$

Če obe strani logaritmiramo dobimo neenačbo $h \leq \log n$. □

9.1.1 Dodajanje lista

Dodajanje lista v 2-4 drevo je preprosto (glej 9.2). Če želimo dodati list u kot otroka nekemu vozlišču w na predzadnjem nivoju, potem preprosto postavimo u za otroka vozlišča w. V tem primeru vsekakor ohranja višino, ampak lahko krši pravilo; če bi imel w štiri otroke pred dodajanjem u, potem ima w sedaj pet otrok. V tem primeru moramo *razdeliti* w v dve vozlišči, w in w', ki imata sedaj 2 in 3 otroke. A ker w' sedaj nima staršev, w rekurzivno nastavimo kot otroka starša w. V tem primeru ima lahko starš vozlišča w' preveč otrok, zato ga moramo razdeliti. Ta postopek se nadaljuje, dokler ne pridemo do vozlišča, ki ima manj kot štiri otroke ali dokler ne razdelimo korena r, v dva vozlišča r in r'. V slednjem primeru naredimo nov koren ki ima otroka r in r'. To hkrati povečuje globino vseh listov in tako ohranja višino the height property.

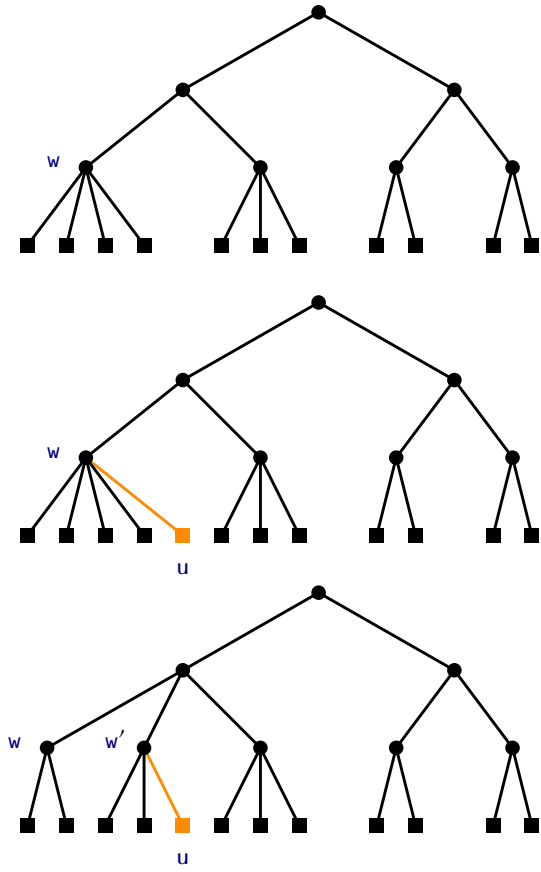
Ker višina 2-4 drevesa ni nikoli več kot $\log n$, se proces dodajanja listov konča po največ $\log n$ korakih.

9.1.2 Odstranjevanje lista

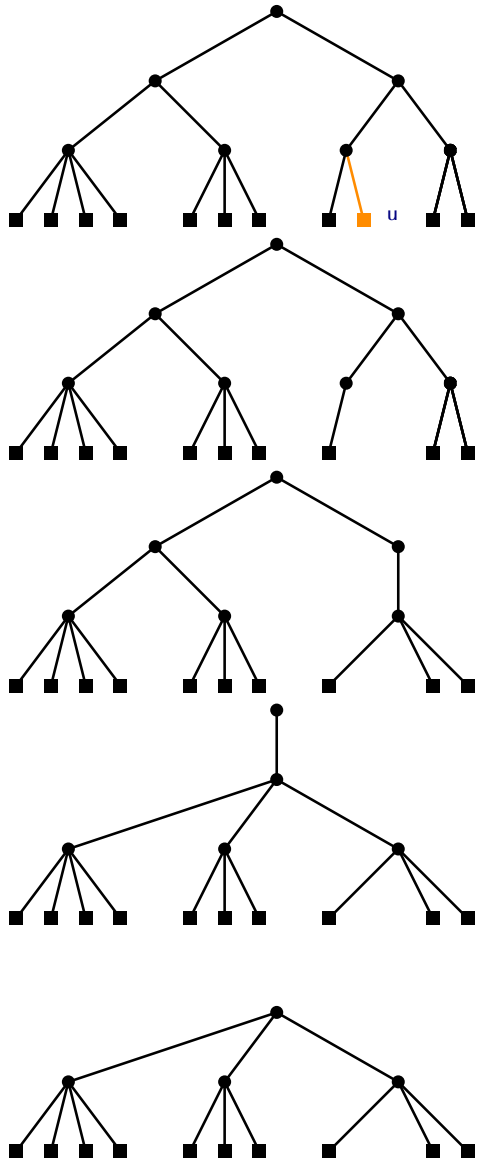
Odstranjevanje lista 2-4 drevesa je lahko rahlo bolj komplicirano kot dodajanje(Glej 9.3). Da ločimo list u od njegovega starša w, ga samo odstranimo. Če ima w samo dva lista in mu mi enega izmed njih odstranimo, moramo drevo ustrezno popraviti, saj krši pravilo.

Da popravimo napako, poiščemo brata w ki je w'. Vozlišče w' definitivno obstaja, ker ima starš w vsaj dva otroka. Če ima w' tri ali štiri otroke, potem vzamemo enega izmed otrok in ga dodamo w. Sedaj ima w dva otroka in w' ima dva ali tri, nato končamo s popravljanjem.

Če ima w' samo dva otroka, potem ju *združimo* v skupno vozlišče, ki ima tri otroke. Potem moramo rekurzivno izbrisati w'. dokler ne dosežemo vozlišča u ali njegovega brata, ki ima več kot dva otroka ali ne dosežemo



Slika 9.2: Dodajanje lista v 2-4 drevo. Ta proces se konča po enemu razdeljevanju, ker ima `w.parent` stopnjo manj kot 4 pred dodajanjem.



Slika 9.3: Odstranjevanje lista z 2-4 drevesa. Ta proces sega vse do korena, saj ima vsak prednik in bratje vozlišča u samo dva otroka.

korena. Če je koren levi z enim samim otrokom, nato pobrišemo koren in otroka dodamo v koren. Tudi to istočasno zmanjšuje višino vsakega lista in tako ohranimo višino drevesa.

Ker višina 2-4 drevesa ni nikoli več kot $\log n$, se proces odstranjevanja listov konča po največ $\log n$ korakih.

9.2 RedBlackTree: Simulirano 2-4 drevo

Rdeče-črno drevo je binarno iskalno drevo, katerega vsako vozlišče, u, je *rdeče* ali *črno*. Rdeče predstavlja vrednost 0, črno pa vrednost 1.

RedBlackTree

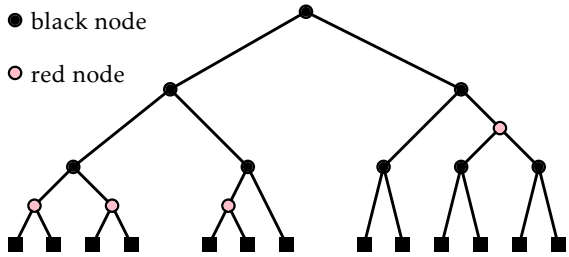
```
class RedBlackNode : public BSTNode<Node, T> {
  friend class RedBlackTree<Node, T>;
  char colour;
};
int red = 0;
int black = 1;
```

Pred in po spreminjanju rdeče-črnega drevesa, morata veljati naslednji dve lastnosti. Vsaka lastnost je definirana v obeh izrazih, v rdeči in črni barvi in številskih vrednostih 0 in 1.

Lastnost 9.3 (višina-črnih)**.** Enako število črnih vozlišč v poti od korena do katerega koli lista. (Vsota barv na poti od korena do poljubnega lista je enaka.)

Lastnost 9.4 (list-ni-rdeč)**.** Dve rdeči vozlišči nista med seboj nikoli sosednji. (Velja za vsako vozlišče u, razen korena, $\texttt{u.barva} + \texttt{u.stars.barva} \geq 1$.)

Opazili smo, da lahko vedno pobarvamo koren, r, rdeče-črnega drevesa črno, ne da bi kršili katero od lastnosti, zato bomo predvidevali, da je koren črne barve in algoritmi za posodabljanje rdeče-črnih dreves bodo to upoštevali. Druga stvar, ki poenostavlja rdeče-črna drevesa je, da so zunanja vozlišča (predstavljena z nil) črna vozlišča. Na ta način ima vsako vozlišče, u, rdeče-črnega drevesa natanko dva otroka, vsak z opredeljeno barvo. Primer rdeče-črnega drevesa je predstavljen v sliki 9.4.



Slika 9.4: Primer rdeče-črnega drevesa, kjer je višina črnih 3. Zunanja (nil) vozlišča so v obliki kvadrata.

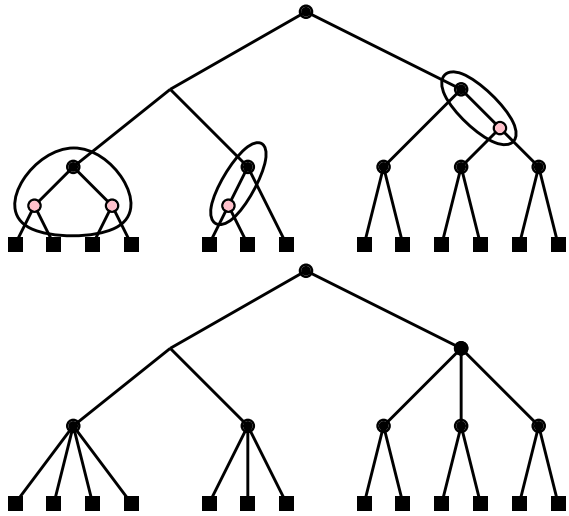
9.2.1 Rdeče-Črna drevesa in 2-4 Drevesa

Sprva se morda zdi presenetljivo, da lahko rdeče-črno drevo učinkovito posodabljamo tako, da ohranjamo višine črnih vozlišč in ne ohranjamo lastnosti rdečih vozlišč. Zdi se tudi nenavadno, da nekateri menijo, da so to koristne lastnosti. Kakorkoli, rdeče-črna drevesa so bila zasnovana za učinkovito simulirati 2-4 drevesa kot binarna drevesa.

Nanašanje na 9.5. Vzemimo, da ima katerokoli rdeče-črno drevo, T, n vozlišč in izvaja naslednje operacije: Zbriše vsako rdeče vozlišče n in poveže otroka vozlišča u direktno na (črnega) starša vozlišča u. Po spremembi imamo drevo T' s samo črnimi vozlišči.

Vsako notranje vozlišče v T' ima dva, tri ali štiri otroke: Črno vozlišče, ki je imelo dva črna otroka bo še vedno imelo črna otroka po spremembi. Črno vozlišče, ki je imelo enega rdečega in enega črnega otroka bo imelo tri otroke po tej spremembi. Črno vozlišče, ki je imelo dva rdeča otroka bo imelo štiri otroke po teji spremembi. Poleg tega, lastnost črnih vozlišč nam zagotavlja, da je vsaka pot od korena do lista v T' enake dolžine. Z drugimi besedami, T' je 2-4 drevo!

2-4 drevo T' ima $n+1$ listov, ki ustrezajo $n+1$ zunanjim vozliščim rdeče-črnega drevesa. Torej, to drevo ima višino največ $\log(n+1)$. Vsaka pot od korena do lista v 2-4 drevesu ustreza poti od korena rdeče-črnega drevesa T do zunanjega vozlišča. Prvo in zadnje vozlišče na poti sta črni in največ eno na vsaki dve notranji vozlišči je rdeče, tako, da ima ta pot največ $\log(n+1)$ črnih in največ $\log(n+1)-1$ rdečih vozlišč. Torej, najdaljša



Slika 9.5: Vsako rdeče-črno drevo ima ustrezno 2-4 drevo.

pot od korena do kateregakoli *notranjega* vozlišča v T je največ

$$2\log(n+1) - 2 \leq 2\log n \ ,$$

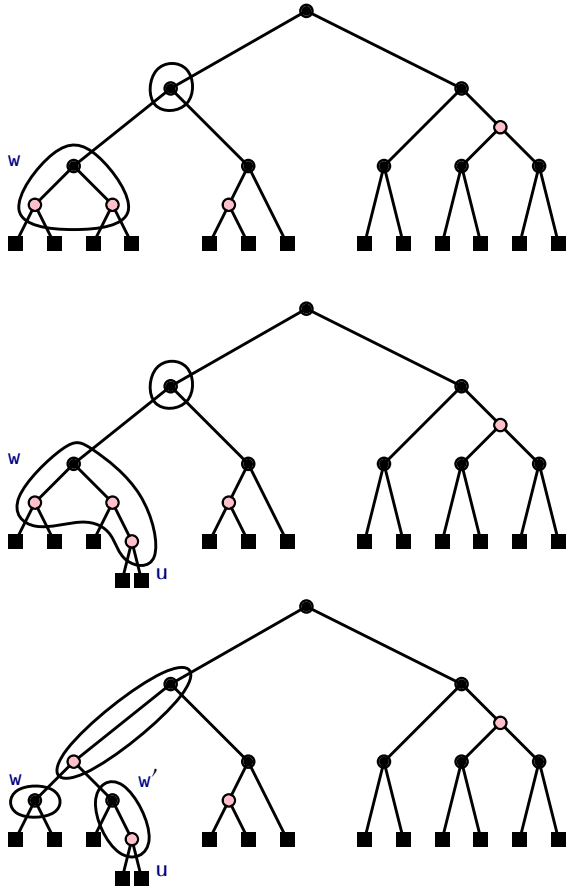
za vsak $n \geq 1$. S tem dokažemo najpomembnejšo lastnost rdeče-črnih dreves:

Lema 9.2. *Višina rdeče-črnega drevesa z* n *vozlišči je največ* $2\log n$.

Sedaj, ko smo videli relacijo med 2-4 drevesi in rdeče-črnimi drevesi, ni tako težko za verjeti, da lahko učinkovito ohranjamo rdeče-črno drevo med dodajanjem in brisanjem elementov.

Videli smo že, da dodajanje elementa v `BinarySearchTree` izvedemo z dodajanjem novega lista. Torej, za implementacijo `add(x)` v rdeče-črno drevo moramo imeti metodo za simulacijo razdelitve vozlišča s petimi otroki v 2-4 drevesu. Vozlišče v 2-4 drevesu s petimi otroki je predstavljeno s črnim vozliščem, ki ima dva rdeča otroka, eden od teh ima tudi rdečega otroka. Lahko "razdelimo" to vozlišče s tem, da ga pobarvamo v rdeče in pobarvamo njegova dva otroka v črno. Primer prikazuje 9.6.

Podobno, implementacija `remove(x)` zahteva metodo za združevanje dveh vozlišč in izposojo sorodnikovega otroka. Združitev dveh vozlišč



Slika 9.6: Simuliranje operacije deljenja 2-4 drevesa med dodajanjem v rdeče-črno drevo. (To simulira dodajanje v 2-4 drevo prikazano na 9.2.)

je inverz deljenja vozlišč (prikazano na 9.6) in vključuje barvanje dveh (črnih) sorodnikov v rdeče in barvanje njegovega (rdečega) starša v črno. Izposoja od sorodnika je najboj zakompliciran postopek in vključuje obe rotacije in barvanje vozlišč.

Vsekakor, med vsem tem moramo še vedno ohranjati lastnost list-ni-rdeč in lastnost višina-črnih. Medtem ko ni več presenetljivo, da lahko naredimo direktno simulacijo 2-4 drevesa z rdeče-črnim drevesom, je vseeno veliko primerov, na katere moramo paziti. V določenem trenutku postane lažje, če ne upoštevamo 2-4 drevesa in samo ohranjamo lastnosti rdeče-črnega drevesa.

9.2.2 Levo-poravnana rdece-crna drevesa

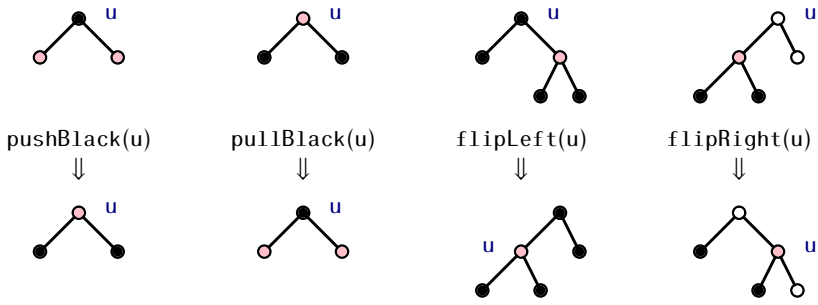
Definicija rdeče-črnega drevesa ne obstaja. Namesto tega imamo družino struktur, ki znajo ohranjati lastnosti višina-črnih in list-ni-rdeč med uporabo operacij `add(x)` in `remove(x)`. Različne strukture to delajo na različne načine. V našem primeru implementiramo podatkovno strukturo, ki ji rečemo `RedBlackTree`. Ta struktura implementira posebno obliko rdeče-črnega drevesa, ki zadovoljuje dodatno lastnost:

Lastnost 9.5 (levo-poravnano)**.** Na kateremkoli vozlišču `u`, če je `u.left` črno, potem je `u.right` črno.

Opomnimo, da rdeče-črno drevo prikazano na 9.4 ne zadošča lastnosti levo-poravnano. Krši jo starš rdečega vozlišča na najbolj desni poti od korena proti listu.

Razlog za ohranjanje lastnosti levo-poravnano je, da zmanjšuje število soočenih primerov pri posodabljanju drevesa med operacijama `add(x)` in `remove(x)`. V smislu 2-4 dreves, to pomeni, da ima vsako 2-4 drevo edinstveno zastopanje: Vozlišče stopnje dva postane črno vozlišče z dvema črnima otrokoma. Vozlišče stopnje tri postane črno vozlišče, katerega levi otrok je rdeč in desni otrok je črn. Vozlišče stopnje štiri postane črno vozlišče z dvema rdečima otrokoma.

Preden podrobno opišemo implementacijo operacij `add(x)` in `remove(x)`, predstavimo nekaj osnovnih podoperacij, uporabljenih v metodah prikazanih v 9.7. Prvi dve podoperaciji stao za manipulacijo barv med ohranjanjem lastnosti višina-črnih. Operacija `pushBlack(u)` vzame za vhod črno



Slika 9.7: Rotacije, potegi in potiski

vozlišče u, katero ima dva rdeča otroka in pobarva u rdeče in njegova dva otroka črno. Operacija pullBlack(x) obrne to opisano operacijo:

RedBlackTree
```
void pushBlack(Node *u) {
  u->colour--;
  u->left->colour++;
  u->right->colour++;
}
void pullBlack(Node *u) {
  u->colour++;
  u->left->colour--;
  u->right->colour--;
}
```

Metoda flipLeft(u) zamenja barve vozlišča u in u.right ter izvede levo rotacijo nad vozliščem u. Ta metoda obrne barve teh dveh vozlišč tako kot tudi njuno relacijo starš-otrok:

RedBlackTree
```
void flipLeft(Node *u) {
  swapcolours(u, u->right);
  rotateLeft(u);
}
```

Operacija flipLeft(u) je posebej uporabna pri povrnitvi lastnosti levo-poravnano na vozlišču u, katero krši to lastnost (ker je u.left črno in u.right rdeče). V tem posebnem primeru, smo lahko prepričani, da ta

operacija ohranja obe lastnosti višina-črnih in list-ni-rdeč. Relacija `flipRight(u)` je simetrična s `fliPLeft(u)`, ko so vloge levega in desnega obrnjene.

RedBlackTree
```
void flipRight(Node *u) {
  swapcolours(u, u->left);
  rotateRight(u);
}
```

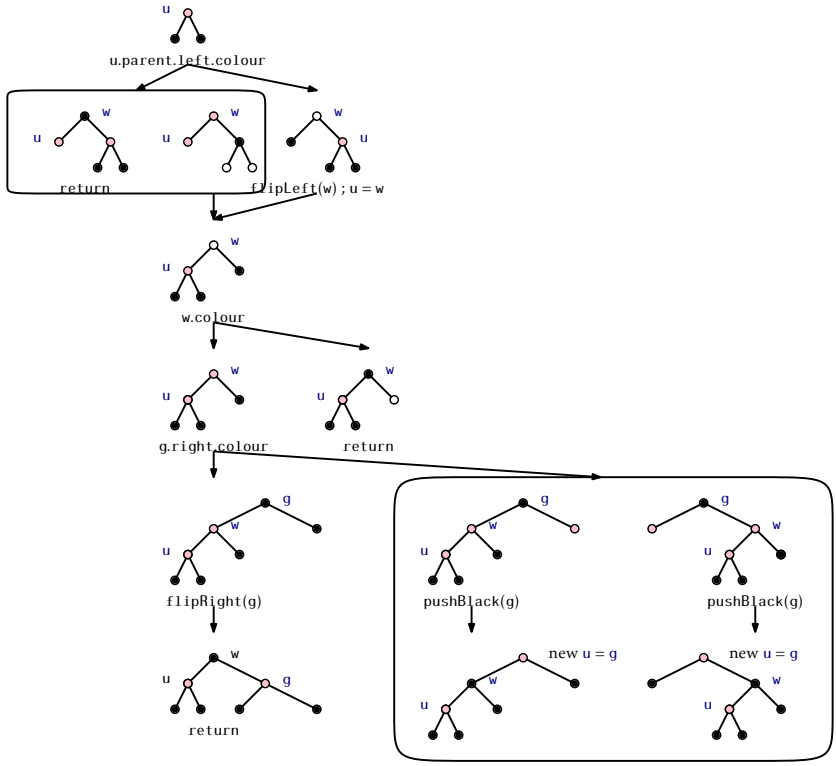
9.2.3 Dodajanje

Za implementacijo `add(x)` v `RedBlackTree`, izvedemo standardno `BinarySearchTree` vstavljanje za dodajanje novega lista, `u`, z `u.x = x` in nastavimo `u.colour = red`. Opomnimo, da to ne spremeni črne višine katerekoli vozlišču, torej ne krši lastnosti višina-črnih. To pa lahko krši lastnost levo-poravnano (če je `u` desni otrok svojega starša) in lahko krši lastnost list-ni-rdeč (če je ujev starš `red`). Za povrnitev teh lastnosti, moramo klicati metodo `addFixup(u)`.

RedBlackTree
```
bool add(T x) {
  Node *u = new Node();
  u->left = u->right = u->parent = nil;
  u->x = x;
  u->colour = red;
  bool added = BinarySearchTree<Node,T>::add(u);
  if (added)
    addFixup(u);
  return added;
}
```

Ilustrirano na 9.8, metoda `addFixup(u)` vzame na vhod vozlišče `u`, katerega barva je rdeča in katero bi lahko kršilo lastnost list-ni-rdeč in/ali lastnost levo-poravnano. Slednja razprava je verjetno nemogoča za sledenje brez sklicevanja na 9.8 ali ponovnega ustvarjanja na kosu papirja. Preden bralec nadaljuje, bi moral preučiti to sliko.

Če je `u` koren drevesa, potem lahko pobarvamo `u` črno za povrnitev obeh lastnosti. Če je tudi ujev sorodnik rdeč, potem mora biti ujev starš črn, torej obe lastnosti levo-poravnano in list-ni-rdeč že držita.



Slika 9.8: Prikaz enega koraka pri popravljanju Lastnost 2 po vstavljanju.

Sicer, najprej preverimo, če je ujev starš, w, kršil lastnost levo-poravnano in, če je da, potem izvedemo operacijo `flipLeft(w)` in nastavimo u = w. Tako pristanemo v lepo definiranem stanju: u je levi otrok starša, w, torej w sedaj zadošča lastnosti levo-poravnano. Vse kar nam ostane je, da zagotovimo lastnost list-ni-rdeč na u. Moramo samo še skrbeti za primer, v katerem je w rdeč, sicer že zadošča lastnosti list-ni-rdeč.

Če sta u in w rdeča, še nismo končali. Lastnost list-ni-rdeč (katero krši u in ne w) implicira, da ujev stari starš g obstaja in je črn. Če je gjev desni otrok rdeč, potem lastnost levo-poravnano zagotavlja, da oba gjev otrok je rdeč in klic na `pushBlack(g)` naredita g rdečega in w črnega. To povrne lastnost list-ni-rdeč na u, ampak lahko povzroči, da jo krši na vozlišču g tako, da celoten proces začne z u = g.

Če je gjev otrok črn, potem klic na `flipRight(g)` postane w črni starš od g in naredi wju dva rdeča otroka, u in g. To zagotovi, da u zadošča lastnosti list-ni-rdeč in g zadošča lastnosti levo-poravnano. Sedaj lahko zaključimo.

RedBlackTree

```
void addFixup(Node *u) {
  while (u->colour == red) {
    if (u == r) { // u is the root - done
      u->colour = black;
      return;
    }
    Node *w = u->parent;
    if (w->left->colour == black) { // ensure left-leaning
      flipLeft(w);
      u = w;
      w = u->parent;
    }
    if (w->colour == black)
      return; // no red-red edge = done
    Node *g = w->parent; // grandparent of u
    if (g->right->colour == black) {
      flipRight(g);
      return;
    } else {
      pushBlack(g);
      u = g;
    }
```

```
  }
}
```

Metoda `insertFixup(u)` ima konstantni čas za iteracijo in vsaka iteracija, ali konča ali premakne `u` bližje korenu. Zato, metoda `insertFixup(u)` konča po $O(\log n)$ iteracijah in po $O(\log n)$ času.

9.2.4 Odstranitev

Operacija `remove(x)` v `RedBlackTree` je najbolj zahtevna za implementacijo in to velja za vse različice rdeče-črnega drevesa. Tako kot operacija `remove(x)` v `BinarySearchTree`, ta operacija išče vozlišče `w` z enim otrokom, `u`, in spoji `w` iz drevesa tako, da `w.parent` posvoji `u`.

Težava lahko nastane takrat, ko je `w` črn, saj s tem kršimo lastnost višina-črnih v `w.parent`. Temu se lahko začasno izognemo z dodajanem `w.colour` do `u.colour`. To predstavlja dve težavi: (1) če se `u` in `w` obe začneta s črno, potem `u.colour` + `w.colour` = 2 (dvojna-črna), ki pa ni veljavna. Če je bil `w` rdeč, se ga nadomesti s črnim vozliščem `u`, kateri lahko krši lastnost levo-poravnano pri `u.parent`. Obe težavi lahko rešimo tako, da pokličemo metodo `removeFixup(u)`.

Metoda `removeFixup(u)` prejme kot vhodni parameter vozlišče `u`, ki je črne (1) ali dvojno-črne barve (2). Če je `u` dvojno-črn, potem `removeFixup(u)` opravi vrsto vrtenj in prebarvanj tako, da dvojno-črno vozlišče premika navzgor po drevesu, dokler ni odpravljeno. Skozi ta postopek se vozlišče `u` spreminja, dokler ne pride do konca, `u` pa pripada korenu podrevesa, ki se je spremenil. Koren tega drevesa je lahko sedaj druge barve. Če je prešel iz rdeče na črno barvo, metoda `removeFixup(u)` na koncu preverja, če ujev starš krši lastnost levo-poravnano in če jo, to popravi.

RedBlackTree

```
void removeFixup(Node *u) {
  while (u->colour > black) {
    if (u == r) {
      u->colour = black;
    } else if (u->parent->left->colour == red) {
      u = removeFixupCase1(u);
    } else if (u == u->parent->left) {
      u = removeFixupCase2(u);
    } else {
```

```
      u = removeFixupCase3(u);
    }
  }
  if (u != r) { // restore left-leaning property, if needed
    Node *w = u->parent;
    if (w->right->colour == red && w->left->colour == black) {
      flipLeft(w);
    }
  }
}
```

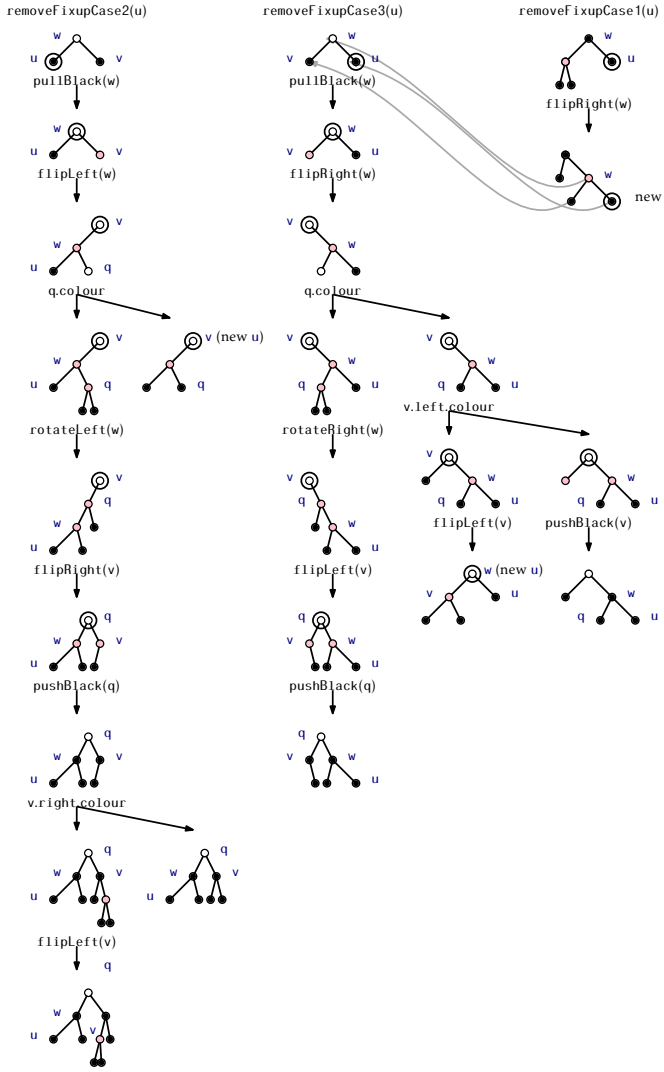
Metoda `removeFixup(u)` je predstavljena na 9.9. Naslednjemu besedilu bo težko, če ne kar nemogoče slediti, brez sklicevanja na 9.9. Vsaka ponovitev zanke v postopku `removeFixup(u)` dvojno-črnega vozlišča `u`, temelji na enemu od štirih primerov:
Primer 0: `u` je koren. To je najpreprostejši primer. Prebarvali smo `u` v črno (s tem ne kršimo nobene lastnosti rdeče-črnega drevesa).
Primer 1: ujev sorodnik, `v`, je rdeč. V tem primeru, je ujev sorodnik levi otrok njegovega starša, `w` (z lastnostjo levo-poravnano). Opravimo desno rotacijo na `w` in nadaljujemo z naslednjo ponovitvijo. Upoštevamo, da ta ukrep povzroči, da wjev starš krši lasnost levo-poravnano in globina `u` naraste. To pomeni tudi, da bo naslednja ponovitev v Primer 3, z `w` obarvanim rdeče. Pri preučevanju Primer 3 spodaj, bomo videli, da se postopek ustavi med naslednjo ponovitvijo.

RedBlackTree

```
Node* removeFixupCase1(Node *u) {
  flipRight(u->parent);
  return u;
}
```

Primer 2: ujev sorodnik, `v`, je črn, `u` je levi otrok njegovega starša, `w`. V tem primeru pokličemo funkcijo `pullBlack(w)`, ki obarva `u` črno, `v` rdeče in spremeni barvo `w` v črno ali dvojno-črno. V tem primeru `w` ne izpolnjuje lastnost levo-porovnano, zato to uredimo tako, da pokličemo `flipLeft(w)`.

V tem trenutku je `w` rdeč, `v` pa je koren poddrevesa, v katerem smo začeli. Preveriti moramo še, če `w` ne povzroča kršitve lastnosti list-ni-rdeč. To naredimo tako, da preverimo wjevega desnega otroka `q`. Če je



Slika 9.9: Iteracija v procesu odpravljanje dvojno-črnega vozlišča po odstranitvi.

q črn, potem w izpolnjuje lastnost list-ni-rdeč in nadaljujemo z naslednjo ponovitvijo z u = v.

Sicer (q je rdeč) sta obe lastnosti, list-ni-rdeč – rdeče pravilo in levo-poravnano, kršeni pri q in w. Levo-poravnano popravimo s klicem `rotateLeft(w)`, sedaj nam ostane le še lastnost list-ni-rdeč, ki jo še vedno kršimo. V tem trenutku je q levi sin od v, w je levi sin od q, q in w sta rdeča, v je črn ali dvojno-črn. `flipRight(v)` popravi drevo tako, da je q sedaj starš tako od v kot od w. Takoj zatem pokličemo `pushBlack(q)`, tako dobimo sledečo situacijo: v in w postaneta črna, q pa dobi originalno barvo od w.

Tako smo se znebili dvojno-črnega vozlišča ter ponovno vzpostavili lastnosti list-ni-rdeč in višina-črnih. Ostane nam samo še ena težava: če ima v desnega sina, ki je rdeč, kršimo lastnost levo-poravnano. To še preverimo ter pokličemo `flipLeft(v)`, ki nam to težavo odpravi, če je potrebno.

RedBlackTree

```
Node* removeFixupCase2(Node *u) {
  Node *w = u->parent;
  Node *v = w->right;
  pullBlack(w); // w->left
  flipLeft(w); // w is now red
  Node *q = w->right;
  if (q->colour == red) { // q-w is red-red
    rotateLeft(w);
    flipRight(v);
    pushBlack(q);
    if (v->right->colour == red)
      flipLeft(v);
    return q;
  } else {
    return v;
  }
}
```

Primer 3: ujev sorodnik je črn in u je desni otrok w. Primer je simetričen Primeru 2 in ga rešujemo precej podobno. Razlikuje se v tem, da je lastnost levo-poravnano asimetrična, in zato ga obravnavamo drugače.

Kot pri prejšnjem, začnemo s klicem `pullBack(w)`, kar naredi v rdeče vozlišče in u črno. S klicem `flipRight(w)` postane v koren našega pod-

drevesa. Tako je w rdeč, zato sedaj ločimo dva primera glede na q, ki je levi sorodnik w.

Če je q rdeč, nam da isto situacijo kot pri Primer 2, vendar z olajševalno okoliščino, namreč, v nam ne more pokvariti lastnosti levo-poravnano.

Bolj zapleteno pa je v primeru, ko je q črne barve. Tu moramo preveriti barvo levega otroka vozlišča v. Če je ta rdeč, potem ima v dva rdeča sinova, zato pokličemo pushBlack(v). Sedaj je w črn, v je prejšnje barve w in smo končali z urejanjem.

Če je vjev levi otrok črn, kršimo lastnost levo-poravnano. Vzpostavimo jo nazaj s klicem flipLeft(v). Nato vrnemo vozlišče v, zato da se naslednja iteracija removeFixup(u) nadaljuje z u = v.

RedBlackTree

```
Node* removeFixupCase3(Node *u) {
  Node *w = u->parent;
  Node *v = w->left;
  pullBlack(w);
  flipRight(w);             // w is now red
  Node *q = w->left;
  if (q->colour == red) { // q-w is red-red
    rotateRight(w);
    flipLeft(v);
    pushBlack(q);
    return q;
  } else {
    if (v->left->colour == red) {
      pushBlack(v); // both v's children are red
      return v;
    } else { // ensure left-leaning
      flipLeft(v);
      return w;
    }
  }
}
```

Vsaka iteracija removeFixup(u) se izvrši v konstantnem času. Primer 2 in 3 lahko proceduro končata, ali pa premakneta u bližje korenu drevesa. Primer 0 (kjer je u koren) se vedno konča, Primer 1 pelje v Primer 3, ki se prav tako konča. Ker vemo, da je višina drevesa največ $2 \log n$, zaključimo, da imamo največ $O(\log n)$ iteracij procedure removeFixup(u),

torej se `removeFixup(u)` izvrši v $O(\log n)$ času.

9.3 Povzetek

Naslednji izrek povzema učinkovitost podatkovne strukture `RedBlackTree`:

Izrek 9.1. *`RedBlackTree` uporablja vmestnik `SSet` in omogoča, da se operacije* `add(x)`, `remove(x)` *in* `find(x)` *izvedejo v najslabšem času* $O(\log n)$ *na operacijo.*

Kar ni vključeno v zgornji teoriji, ima dodatni bonus:

Izrek 9.2. *Med vsemi klici metod* `addFixup(u)` *in* `removeFixup(u)` *se vsako zaporedje operacij* `dodaj(x)` *in* `odstrani(x)` *izvede v času* $O(m)$, *na zacetku ko je `RedBlackTree` prazen.*

Naredili smo samo skico dokaza za 9.2. S primerjanjem metod `addFixup(u)` in `removeFixup(u)`, z algoritmi za dodajanje ali odstranjevanje listov v 2-4 drevesu se lahko prepričamo, da se ta lastnost deduje z 2-4 drevesa. Običajno, če lahko dokažemo, da je skupni čas porabljen za delitev, združevanje in zadolževanje v 2-4 drevesu $O(m)$, potem ta dokaz namiguje na 9.2.

Dokaz tega izreka za 2-4 drevo uporablja potencial odplačne analize.[2] Definiraj potencial za notranje vozlišče `u` v 2-4 drevesu kot

$$\Phi(u) = \begin{cases} 1 & \text{če ima } u \text{ 2 otroka} \\ 0 & \text{če ima } u \text{ 3 otroke} \\ 3 & \text{če ima } u \text{ 4 otroke} \end{cases}$$

in potencial za 2-4 drevo kot vsoto potencialov za njegova vozlišča. Delitev se pojavi, ko se vozlišča s štirimi otroci razdelijo na dve vozlišči z dvemi in tremi octroci. To pomeni, da se skupni potencial zmanjša za $3 - 1 - 0 = 2$. Ko pride do združevanja, se dve vozlišči z dvemi otroki zamenjata z vozliščem, ki ima tri otroke. Rezultat tega je zmanjšanje potenciala za $2 - 0 = 2$. Torej se za vsako delitev ali združitev potencial zmanjša za dva.

[2] Oglej si 2.2 in 3.1 dokaze za potencialno metodo v ostalih aplikacijah.

Nato bodite pozorni, da če zanemarimo delitev in združevanje vozlišč, temu sledi konstantno število vozlišč katerih število otrok je bilo s tem ali odstranitvijo lista spremenjeno. Ob dodajanju vozlišča se nekemu vozlišču število otrok poveča za ena, s tem pa povečamo potencial za največ tri. Med odstranitvijo lista, se vozlišču zmanjša število otrok za ena, potencial pa se mu poveča največ za ena. Ob tem sta lahko v odstranjevanje vključeni dve vozlišči s čimer se njun potencial poveča za največ ena.

Kot povzetek torej sledi, da lahko vsaka združitev ali delitev povzroči zmanjšanje potenciala za vsaj dva. V primeru, da ne upoštevamo združitev ter delitev pri dodajanju oziroma odstranjevanju, pa lahko povzroči povečanje potenciala za največ tri. Potencial je vedno ne-negativno število. Zatorej je število združitev ter delitev, povzročenih s strani m dodajanj oziroma odstranjevanj, na prvotno praznem drevesu največ $3m/2$. 9.2 izhaja iz te analize in povezav med 2-4 drevesi in rdeče-črnimi drevesi.

9.4 Razprava in naloge

Rdeče-črna drevesa sta prvič predstavila Guibas in Sedgewick [**?**]. Kljub njihovi visoki zapletenosti izvedbe so najdeni v nekaterih najbolj pogosto uporabljenih knjižnjicah in aplikacijah. Večina algoritmov in učbenikov o podatkovnih strukturah razpravlja o nekaj različicah rdeče-črnih dreves.

Andersson [**?**] je predstavil levo-visečo različico uravnananega drevesa, ki je podobna rdeče-črnim drevesom, vendar z omejitvijo, da ima vsako vozlišče lahko največ enega rdečega otroka. Zaradi omenjene omejitve je izvedba 2-3 dreves veliko pogostejša od 2-4 dreves. Ta so veliko preprostejša kot podatkovna struktura `RedBlackTree` predstavljenih v tem poglavju.

Sedgewick [**?**] opisuje dve verziji levo-visečih rdeče-črnih dreves. Te uporabljajo rekurzijo, skupaj s simulacijo delitvije od zgoraj navzdol in združevanje v 2-4 drevesih. Kombinacija obeh tehnik nam omogoča zelo kratek in eleganten zapis kode.

Povezana, a starejša, podatkovna struktura je *AVL tree* [**?**]. AVL drevesa so *height-balanced*: V vsakem vozlišču u se višina levega poddrevesa `u.left` ter desnega poddrevesa `u.right` razlikuje za največ ena. Iz tega

sledi: če je $F(h)$ najmanjše število listov drevesa višine h, potem se $F(h)$ uvršča v okvir Fibonaccijevega zaporedja

$$F(h) = F(h-1) + F(h-2)$$

z osnovnima primeroma $F(0) = 1$ in $F(1) = 1$. $F(h)$ je tako približno $\varphi^h/\sqrt{5}$, kjer je $\varphi = (1+\sqrt{5})/2 \approx 1.61803399$ is the *golden ratio*. (Bolj natančno $|\varphi^h/\sqrt{5} - F(h)| \leq 1/2$.) S pomočjo utemeljitvije v 9.1, to pomeni

$$h \leq \log_\varphi \mathsf{n} \approx 1.440420088 \log \mathsf{n} \ ,$$

torej imajo AVL drevesa manjšo višino kot rdeče-črna drevesa. Višina je lahko vzdrževana med izvajanjem `dodaj(x)` in `odstrani(x)` operacij z sprehodom navzgor do korena drevesa, med katerim se izvede uravnoteženje vsakega vozlišču `u`, katerega višina levega in desnega poddrevesa se razlikuje za dva. Glej 9.10.

Uporaba Anderssonove in Sadgewickove različice rdeče-črnih dreves in uporaba AVL dreves je enostavnejša kot uporaba strukture `RedBlackTree`. Žal pa ne more nobena od njih zagotavljati, da bi bil amortizacijski čas $O(1)$, za vsako posodobitev uravnovešen. Zlasti zato, ker te strukture nemoremo primerjati z 9.2.

Naloga 9.1. Nariši 2-4 drevo, ki ustreza `RedBlackTree` iz 9.11.

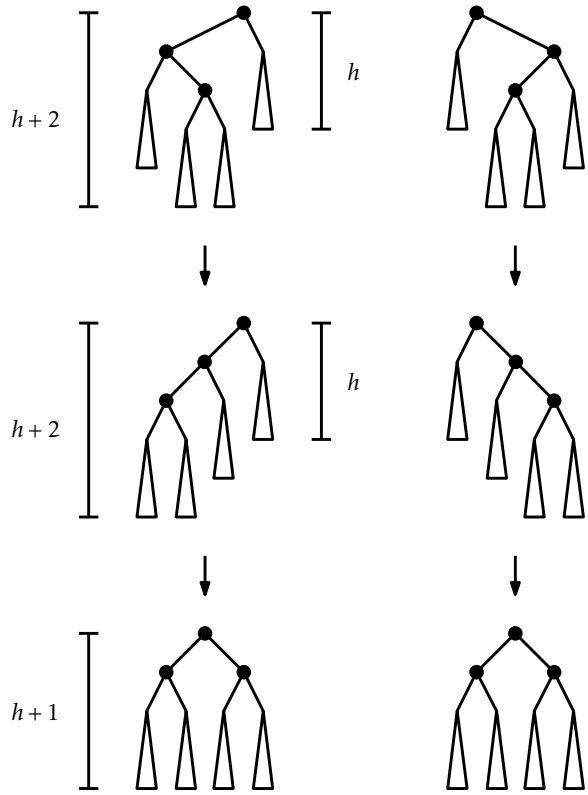
Naloga 9.2. Nariši dodajanje elementov 13, 3.5 in 3.3 na `RedBlackTree` iz 9.11.

Naloga 9.3. Nariši odstranjevanje elementov 11, 9, ter 5 na `RedBlackTree` iz 9.11.

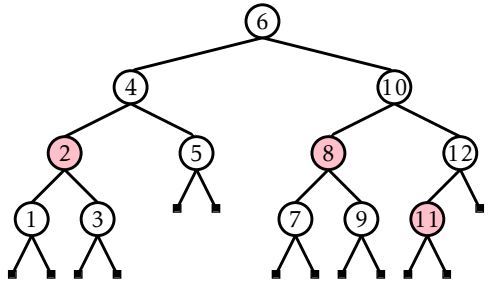
Naloga 9.4. Pokaži, da za poljubno velike vrednosti `n`, obstaja rdeče-črno drevo z `n` vozlišči, ki imajo višino $2\log \mathsf{n} - O(1)$.

Naloga 9.5. Preuči operaciji `pushBlack(u)` and `pullBlack(u)`. Kaj naredijo ti dve operaciji na 2-4 drevesu, ki temelji na simulaciji z rdeče-črnim drevesom.

Naloga 9.6. Pokaži, da za poljubno velike vrednosti `n`, obstaja zaporedje ukazov `dodaj(x)` in `odstrani(x)`, ki vodi do rdeče-črnega drevesa z `n` vozlišči, ki imajo višino $2\log \mathsf{n} - O(1)$.



Slika 9.10: Uravnoteženje v AVL drevesih. Največ dve rotaciji sta potrebni, da vozlišče s poddrevesoma višine h in $h+2$ spremenimo v vozlišče s poddrevesoma višine $h+1$.



Slika 9.11: A red-black tree on which to practice.

Naloga 9.7. Zakaj metoda `odstrani(x)` v `RedBlackTree` izvede operacijo `u.parent = w.parent`? Naj nebi bilo to storjeno že z klicem metode `splice(w)`?

Naloga 9.8. Predvidevaj, da ima 2-4 drevo T, n_ℓ listov in n_i notranjih vozlišč.

1. Kakšna je najmanjša vrednost n_i, kot funkcija n_ℓ?
2. Kakšna je največja vrednost n_i, kot funkcija n_ℓ?
3. Če je T' rdeče-črno drevo, ki predstavlja T, koliko ima potem T' rdečih vozlišč?

Naloga 9.9. Predpostavimo, da imamo binarno iskalno drevo z n vozlišči in višini največ $2\log n - 2$. je možno, da vedno pobarvamo vozlišča tako, da drevo zadošča pogoju črne višine in pogoju da rob ni rdeč? Če da, ali potem zadošča tudi lastnostim levo-visečih dreves?

Naloga 9.10. Predpostavimo, da imamo dva rdeče-črna drevesa T_1 in T_2, ki imata enako višino črnih vozlišč h in, da je največji ključ v T_1 manjši od najmanjšega ključa v T_2. Prikaži kako se združita drevesi T_1 in T_2 v eno rdeče-črno drevo v času $O(h)$.

Naloga 9.11. Nadgradi rešitev iz 9.10, da bo veljala tudi za drevesi T_1 in T_2, ki imata različni višini črnih vozlišč , $h_1 \neq h_2$. Čas izvajanja naj bo $O(\max\{h_1, h_2\})$.

Naloga 9.12. Dokaži, da mora AVL drevo pri izvajanju `add(x)` metode, izvesti največ eno operacijo uravnoteženja (vključuje največ dve rotaciji; glej 9.10). Podaj primer AVL drevesa in klica metode `remove(x)` na tem drevesu, ki zahteva $\log n$ operacij uravnoteženja.

Naloga 9.13. Napiši razred `AVLTree`, ki uporablja AVL drevo kot je opisano zgoraj. Primerjaj hitrost izvajanja s hitrostjo `RedBlackTree`. Katera izvedba ima hitrejšo operacijo `find(x)`?

Naloga 9.14. Oblikuj in izvedi vrsto poskusov, da primerjamo relativno uspešnost metod `find(x)`, `add(x)`, in `remove(x)` for the `SSet` implemeentations `SkiplistSSet`, `ScapegoatTree`, `Treap`, and `RedBlackTree`. Bodite

prepričani, da vključite več testnih primerov, vključno s primeri, ko so podatki naključno razporejeni, že razporejeni, jih odstranite, ko so urejeni in tako naprej.

Poglavje 10

Kopice

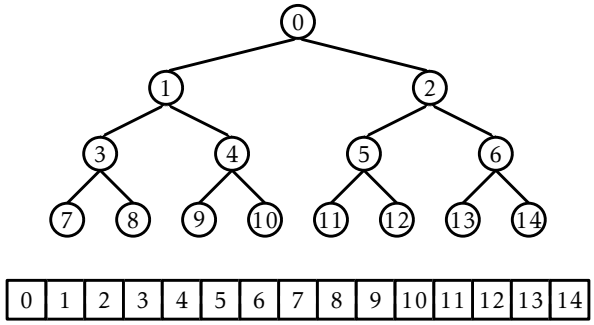
V tem poglavju si bomo pogledali 2 implementacije zelo uporabne podatkovne strukture `Polje` s prednostjo. Obe od teh dveh struktur sta posebne oblike Dvojiškega drevesa imenovani *Kopica*, kar pomeni "neorganizirana kopica". To je v nasprotju z dvojiškimi iskalnimi drevesi pri katerih pomislimo na zelo urejeno kopico.

Prva izvedba kopic uporablja polje, da simuliramo popolno dvojiško drevo. Ta zelo hitra implementacija je osnova za enega izmed najhitrejših znanih sortirnih algoritmov, in sicer kopično urejanje (glej 11.1.3). Druga implementacija je bazirana na bolj fleksiblinih dvojiških drevesih, ki podpirajo `meld`(h) operacijo, ki omogoča vrsti s prednostjo, da obsorbira elemente druge vrste s prednostjo h.

10.1 `BinaryHeap`: implicitno dvojiško drevo

Naša prva implementacija `Queue` (s prednostjo) temelji na tehniki, ki je stara preko 400 let. *Eytzingerjeva metoda* nam omogoča, da predstavimo popolno dvojiško drevo kot polje, v katerem imamo vozlišča postavljena v vrsto iz leve proti desni (glej 6.1.2). Na ta način je koren drevesa shranjen na poziciji 0, njegov levi otrok je shranjen na poziciji 0, njegov desni otrok na pozciji 1, levi otrok na 2, levi otrok otroka na poziciji 3 in tako naprej. Glej 10.1.

Če uporabimo Eytzingerjevo metodo na dovolj velikih drevesih se začnejo pojavljati vzorci. Levi otrok vozlišča pri indexu i je na indexu



Slika 10.1: Eytzingerjeva metoda predstavlja popolno dvojiško drevo kot polje.

`left(i)` = 2`i`+1 in desni otrok vozlišča pri indexu `i` je na indexu `right(i)` = 2`i` + 2. Starš vozlišča pri indexu `i` pa je na `parent(i)` = (`i` − 1)/2.

BinaryHeap
```
int left(int i) {
  return 2*i + 1;
}
int right(int i) {
  return 2*i + 2;
}
int parent(int i) {
  return (i-1)/2;
}
```

`BinaryHeap` uporablja to tehniko, da implicitno predstavi popolno dvojiško drevo v katerem so elementi *kopično urejeni*: Vrednost shranjena na katerem koli indexu `i` ni manjša kot vrednost shranjena na katerem koli indexu `parent(i)`, razen izjeme vrednosti korena `i` = 0. To nam omogoča, da je najmanjša vrednost `Queue` s prednostjo tako shranjena na poziciji 0 (koren).

V `BinaryHeap`, je `n` elementov shranjenih v tabeli `a`:

BinaryHeap
```
array<T> a;
int n;
```

Implementacija operacije `add(x)` je preprosta. Kot vse strukture bazirane na polju najprej pogledamo, če je `a` poln (preverimo `a.length` = `n`)

in če je, povečamo a. Nato x zapišemo na mesto a[n] in povečamo n. Na tej točki je potrebno storiti samo še to, da zagotovimo lastnost kopice. To storimo tako, da zamenjujemo x z njegovim staršem, dokler ni x manjši od svojega starša. Glej 10.2.

BinaryHeap

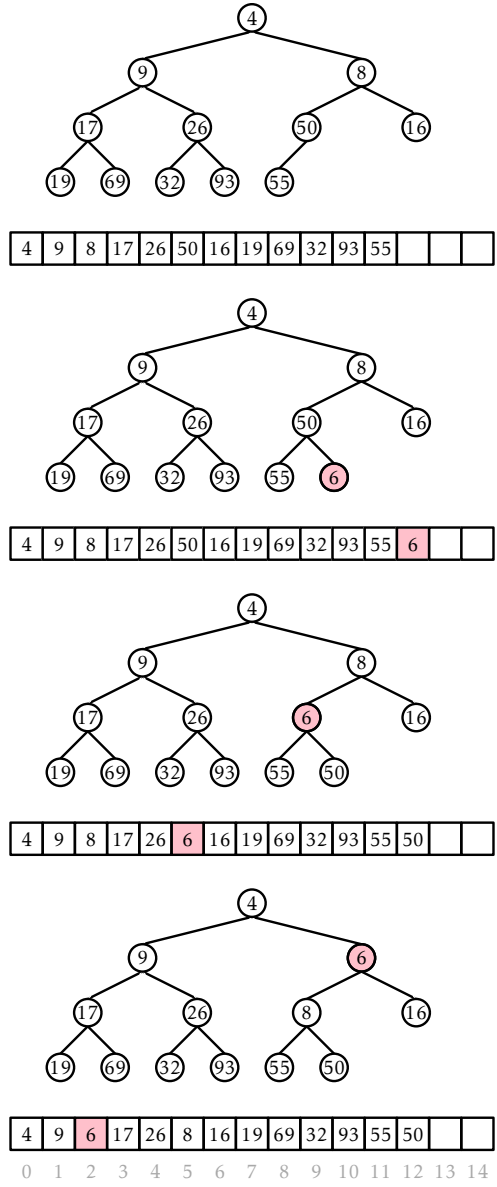
```
bool add(T x) {
  if (n + 1 > a.length) resize();
  a[n++] = x;
  bubbleUp(n-1);
  return true;
}
void bubbleUp(int i) {
  int p = parent(i);
  while (i > 0 && compare(a[i], a[p]) < 0) {
    a.swap(i,p);
    i = p;
    p = parent(i);
  }
}
```

Implementacija remove() operacije, katera odstrani najmanjšo vrednost v kopici, je nekoliko težja. Vemo, kje je najmanjši element (v korenu), vendar ga moramo po odstranitvi nadomestiti in zagotoviti, da ohranjamo lastnosti kopice.

Najlažji način, da to naredimo je, da koren nadomestimo z vrednostjo $a[n-1]$, zbrišemo vrednost in zmanjšamo n. Na žalost novi koren najverjetneje ni najmanjši element, zato ga moramo prestaviti po kopici navzdol. To naredimo tako, da rekurzivno primerjamo element z njegovimi otroki. V primeru, da je element v kopici najmanjši smo končali, v nasprotnem primeru ga zamenjamo z najmanjšim izmed otrok in nadaljujemo ta postopek rekurzivno.

BinaryHeap

```
T remove() {
  T x = a[0];
  a[0] = a[--n];
  trickleDown(0);
  if (3*n < a.length) resize();
  return x;
}
```



Slika 10.2: Dodajanje elementa 6 v `BinaryHeap`.

```
void trickleDown(int i) {
  do {
    int j = -1;
    int r = right(i);
    if (r < n && compare(a[r], a[i]) < 0) {
      int l = left(i);
      if (compare(a[l], a[r]) < 0) {
        j = l;
      } else {
        j = r;
      }
    } else {
      int l = left(i);
      if (l < n && compare(a[l], a[i]) < 0) {
        j = l;
      }
    }
    if (j >= 0)  a.swap(i, j);
    i = j;
  } while (i >= 0);
}
```

Kot ostale implementirane strukture polja, bomo mi ignorirali porabljen čas v celicah za funkcijo `resize()`, ker se to lahko obračunava na amortizacijskem argumentu iz Lemma 2.1. Pretečeni čas za `add(x)` in `remove()` je odvisen od višine (implicitnega) dvojiškega drevesa. Na srečo je to *polno* Dvojiško drevo; vsak nivo, razen zadnjega ima največje možno število vozlišč. Tako, je višina drevesa enaka h in ima najmanj 2^h vozlišč. Začnimo na ta način

$$n \geq 2^h \ .$$

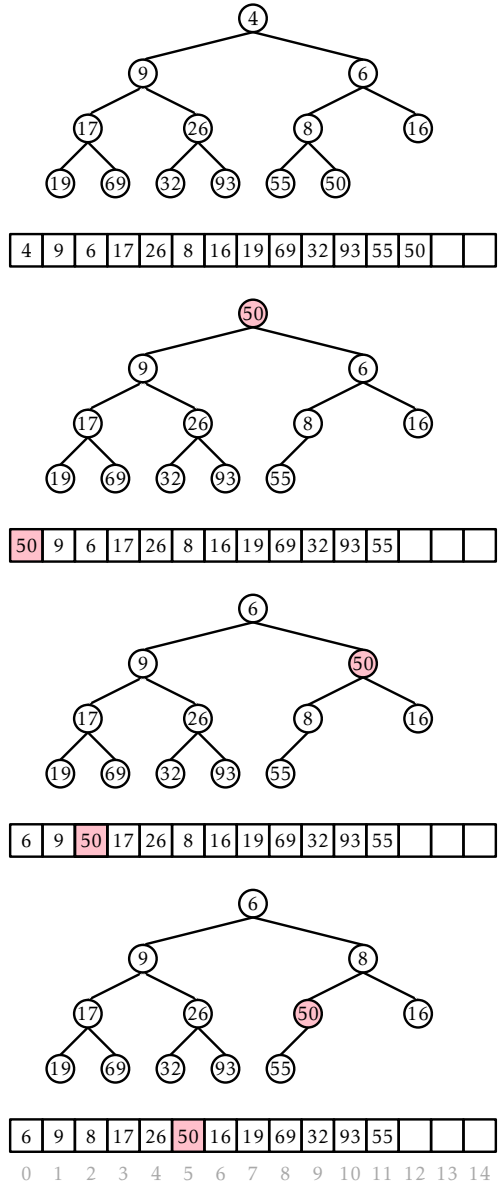
Če logaritmiramo, dobimo na obeh straneh enačbe

$$h \leq \log n \ .$$

Tako obe, `add(x)` in `remove()` operaciji tečeta v $O(\log n)$ času.

10.1.1 Povzetek

Naslednji teorem povzame uspešnost `BinaryHeap`.



Slika 10.3: Odstranjevanje najmanjšega elementa, 4, iz `BinaryHeap`.

Izrek 10.1. *`BinaryHeap` implementira `Queue` (s prednostjo). Če ignoriramo ceno* `resize()` *za povečanje polja,* `BinaryHeap` *izvede operaciji* `add(x)` *in* `remove()` *v času* $O(\log n)$ *na operacijo.*

Poleg tega, začenši s prazno `BinaryHeap`*, katero koli zaporedje* m *operacij* `add(x)` *in* `remove()` *potrebuje skupno* $O(m)$ *časa za vse klice funkcije* `resize()`.

10.2 MeldableHeap: Naključna zlivalna kopica

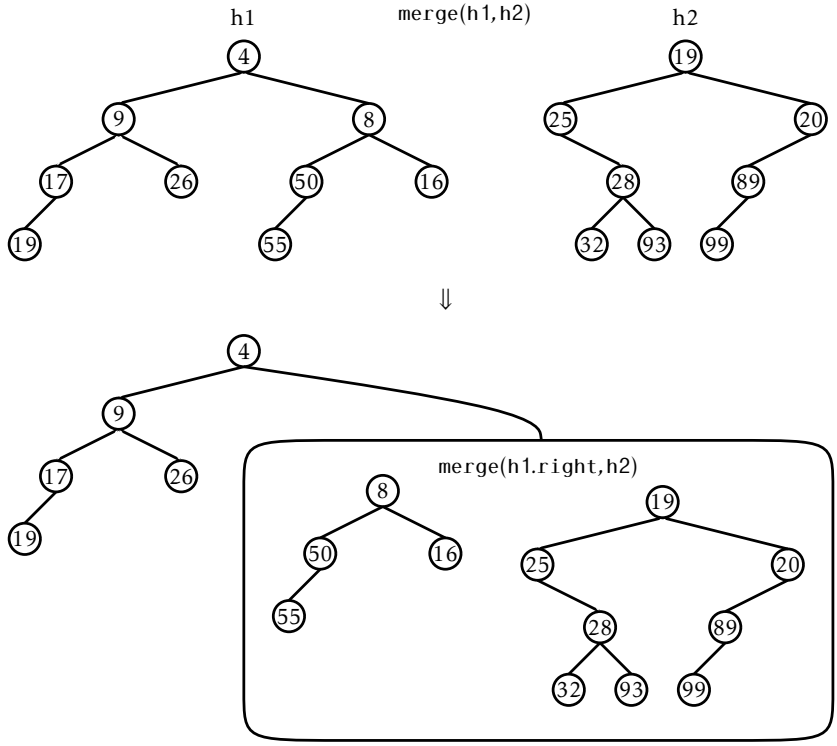
V poglavju bomo opisali `MeldableHeap`, implementacijo prioritetne vrste `Queue`, shranjeno v kopičasto urejenem dvojiškem drevesu. Za razliko od `BinaryHeap`, pri katerem dvojiško drevo definira število elementov, dvojiško drevo `MeldableHeap` nima omejitev glede oblike.

Operaciji `add(x)` in `remove()` v `MeldableHeap` sta implementirani z uporabo operacije `merge(h1,h2)`. Operacija `merge(h1,h2)` zlije skupaj kopici`h1` in `h2` tako, da združi vozlišči kopice `h1` in `h2` in vrne korensko vozlišče nove kopice, ki vsebuje vse elemente poddreves vozlišč `h1` in `h2`.

Operacijo `merge(h1,h2)` lahko implementiramo rekurzivno. Glej 10.4. Če je vozlišče `h1` oz. `h2` `nil`, zlivamo s prazno množico in vrnemo vozlišče `h1` ali `h2`, ki ni `nil`. V nasprotnem primeru zamenjamo vlogi `h1` in `h2` glede na velikost vrednosti vozlišča tako, da večje od obeh vozlišč postane koren nove kopice. V primeru, da je v korenu vrednost `h1.x`, potem lahko `h2` lahko rekurzivno zlijemo z `h1.left` ali `h1.right`, odvisno od naključne vrednosti meta kovanca.

MeldableHeap

```
Node* merge(Node *h1, Node *h2) {
  if (h1 == nil) return h2;
  if (h2 == nil) return h1;
  if (compare(h1->x, h2->x) > 0) return merge(h2, h1);
        // now we know h1->x <= h2->x
  if (rand() % 2) {
    h1->left = merge(h1->left, h2);
    if (h1->left != nil) h1->left->parent = h1;
  } else {
    h1->right = merge(h1->right, h2);
    if (h1->right != nil) h1->right->parent = h1;
  }
```



Slika 10.4: Zlivanje h1 in h2 opravimo z združitvijo h2 in h1.left oz. h1.right.

```
  return h1;
}
```

V naslednjem delu poglavja pokažemo, da ima operacija merge(h1,h2) pričakovano časovno zahtevnost $O(\log n)$, kjer je n končno število elementov v h1 in h2.

S pomočjo operacije merge(h1,h2) je vstavljanje add(x) enostavno. Ustvarimo novo vozlišče u z vrednostjo x in zlijemo vozlišče s korenom kopice:

MeldableHeap

```
bool add(T x) {
  Node *u = new Node();
  u->left = u->right = u->parent = nil;
  u->x = x;
```

```
  r = merge(u, r);
  r->parent = nil;
  n++;
  return true;
}
```

Operacija ima pričakovano časovno zahtevnost $O(\log(n+1)) = O(\log n)$.

Podobno je z operacijo `remove()`. Odstranjujemo korensko vozlišče, ki ga zamenja rezultat zlivanja njegovih otrok:

MeldableHeap

```
T remove() {
  T x = r->x;
  Node *tmp = r;
  r = merge(r->left, r->right);
  delete tmp;
  if (r != nil) r->parent = nil;
  n--;
  return x;
}
```

Tudi `remove()` ima pričakovano časovno zahtevnost $O(\log n)$.

`MeldableHeap` lahko implementira tudi mnogo ostalih operacij s časovno zahtevnostjo $O(\log n)$, npr.:

- `remove(u)`: iz kopice odstranimo vozlišče `u` (in pripadajoč ključ `u.x`).
- `absorb(h)`: vse elemente `MeldableHeap` `h` dodamo kopici, kjer v postopku praznimo `h`.

Vsaka operacija lahko vsebuje konstantno število `merge(h1,h2)` operacij s časovno zahtevnostjo $O(\log n)$.

10.2.1 Analiza `merge(h1,h2)`

Analiza operacije `merge(h1,h2)` je osnovana na analizi naključnega sprehoda v dvojiškem drevesu. V dvojiškem drevesu se *naključni sprehod* začne v korenu drevesa. V vsakem koraku naključnega sprehoda vržemo kovanec, ki določa smer sprehoda (levi ali desni otrok trenutnega vozlišča). Ko trenutno vozlišče postane `nil` se sprehod konča.

Sledeča lema je zanimiva, ker ni odvisna od oblike dvojiškega drevesa:

Lema 10.1. *Pričakovana dolžina naključnega sprehoda v dvojiškem drevesu z* n *vozlišči je največ* $\log(n+1)$.

Dokaz. Trditev lahko dokažemo z indukcijo. Za osnovo izberimo $n = 0$ in dolžino sprehoda $0 = \log(n+1)$. Trditev drži za vsa pozitivna števila $n' < n$.

Dolžino korenskega levega poddrevesa označimo z n_1, da bo $n_2 = n - n_1 - 1$ velikost korenskega desnega poddrevesa. Sprehod se začne v korenu, zavzame en korak in nato nadaljuje v poddrevesu velikosti n_1 ali n_2. Po naši induktivni predpostavki je pričakovana dolžina sprehoda naključnega sprehoda

$$\mathrm{E}[W] = 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \ ,$$

saj je vsako od n_1 ali n_2 manjše od n. Ker je log funkcija konkavne oblike $\mathrm{E}[W]$ doseže maksimum, ko je $n_1 = n_2 = (n-1)/2$. Potemtakem je pričakovano število korakov

$$\begin{aligned}\mathrm{E}[W] &= 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \\ &\le 1 + \log((n-1)/2+1) \\ &= 1 + \log((n+1)/2) \\ &= \log(n+1) \ .\end{aligned}$$

□

Za bralce s pomanjkljivim poznavanjem informacijske teorije lahko dokaz za 10.1 izrazimo s pomočjo entropije.

Informaciski teoretični dokaz za 10.1. Naj d_i označuje globino i-tega zunanjega vozlišča. Spomnimo se, da ima dvojiško drevo z n vozlišči $n+1$ zunanjih vozlišč. Verjetnost, da bo naključni sprehod dosegel i-to zunanje vozlišče je natančno $p_i = 1/2^{d_i}$. Tako je pričakovana dolžina naključnega sprehoda

$$H = \sum_{i=0}^{n} p_i d_i = \sum_{i=0}^{n} p_i \log\left(2^{d_i}\right) = \sum_{i=0}^{n} p_i \log(1/p_i)$$

Desna stran enačbe je prepoznavna kot entropija verjetnostne distribucije na $n+1$ elementih, katera nikoli ne preseže $\log(n+1)$, kar dokazuje lemo.

□

S tem tudi enostavno dokažemo, da je čas izvajanja operacije `merge(h1,h2)` $O(\log n)$.

Lema 10.2. *Če sta* `h1` *in* `h2` *korena dveh kopic z vozliščema* n_1 *in* n_2 *je pričakovan čas izvajanja operacije* `merge(h1,h2)` *največ* $O(\log n)$, *kjer je* $n = n_1 + n_2$.

Dokaz. Vsak korak algoritma za zlivanje zavzame en korak v naključnem sprehodu, bodisi v kopici s korenom `h1` bodisi v kopici s korenom `h2`. Algoritem se zaključi ko katerikoli izmed dveh naključnih sprehodov doseže konec drevesa. Pričakovano število korakov zlivalnega algoritma je največ

$$\log(n_1 + 1) + \log(n_2 + 1) \leq 2 \log n \ . \qquad \square$$

10.2.2 Povzetek

Sledeči teorem povzame zmogljivost `MeldableHeap`:

Izrek 10.2. *`MeldableHeap` implementira (prioritetni) `Queue` vmesnik. `MeldableHeap` podpira operaciji* `add(x)` *in* `remove()`. $O(\log n)$ *je pričakovan čas izvajanja posamezne operacije.*

10.3 Diskusije in vaje

Izgleda, da je implicitno predstavitev polnega dvojiškega drevesa s tabelo ali seznamom prvič predlagal Eytzinger [**?**]. Implicitno predstavitev je v svojih knjigah uporabil na primeru družinskih drevesih plemiških družin Podatkovno strukturo `BinaryHeap` opisano v tej knjigi je prvič predstavil Williams [**?**].

Naključno podatkovno strukturo `MeldableHeap` sta prvič predlagala Gambin in Malinowski [**?**]. Obstajajo tudi druge implementacije zlivalnih kopic vključno z levo poravnane kopice [**?**, **?**, Section 5.3.2], binomske kopice [**?**], Fibonaccijeve kopice [**?**], parne kopice [**?**], in samoprilagoditvene kopice [**?**], Čeprav niso tako enostavne kot je struktura `MeldableHeap`.

Nekaj zgoraj navedenih struktur podpira tudi operacijo `decreaseKey(u,v)` v kateri se vrednost vozlišča `u` zniža na vrednost vozlišča `y` (ob predpogoju $y \leq u.x$). V večini strukturah lahko opracijo `decreaseKey(u,v)` iz-

vajamo s časovno zahtevnostjo $O(\log n)$ z odstranjanjem vozlišča u in dodajanjem vozlišča t. Nekatere strukture lahko implementirajo operacijo bolj učinkovito. V Fibonaccijevih kopicah ima amortizirano časovno zahtevnost $O(1)$ in amortizirano $O(\log \log n)$ v posebni različici parnih kopic [**?**]. Omenjena učinkovitejša različica operacije `decreaseKey(u,y)` se uporablja pri pohitritvi grafov, vključno z algoritmom Dijkstre za iskanje najkrajše poti [**?**].

Naloga 10.1. Narišite dodajanje elementov vrednosti 7 in vrednosti 3 na `BinaryHeap` prikazano na koncu slike 10.2.

Naloga 10.2. Narišite odstranjevanje naslednjih dveh elementov (6 in 8) na `BinaryHeap` prikazano na koncu slike 10.3.

Naloga 10.3. Implementirajte metodo `remove(i)`, ki odstrani shranjene vrednosti v `a[i]` v `BinaryHeap`. Metoda mora teči v časovni zahtevnosti $O(\log n)$. Razložite, zakaj se ta metoda verjetno ne bo uporabljala.

Naloga 10.4. A d-ary drevo je posplošitev dvojiškega drevesa, v katerem ima vsako notranje vozlišče d otrok. Uporabite Eytzingerjevo metodo, ki je lahko predstavljena kot popolno d-tiško drevo z uporabo tabele. Ugotovite enačbe, v katerih je podan indeks i, določite indeks staršev od i in vsakega d otroka od indeksa i'.

Naloga 10.5. Uporabite kar ste spoznali v 10.4, oblikujte in implementirajte *DaryHeap*, d-aryeva posplošitev `BinaryHeap`. Analizirajte čas poteka za operacije na `DaryHeap` in testirajte vaše delovanje `DaryHeap` katera se izvaja na `BinaryHeap` katere implementacija je podana.

Naloga 10.6. Narišite dodajanje elementov vrednosti 17 in 82 v `MeldableHeap` h1 prikazan na sliki 10.4. Uporabite kovanec za simulacijo naključnega bita, če je potrebno.

Naloga 10.7. Narišite odstranjevanje naslednjih dveh elementov (4 in 8) iz `MeldableHeap` h1 prikazano v 10.4. Uporabite kovanec za simulacijo naključnega bita, če je potrebno.

Naloga 10.8. Implementirajte metodo `remove(u)`, ki odstrani vozlišče u iz a `MeldableHeap`. Metoda mora teči v časovni zahtevnosti $O(\log n)$.

Naloga 10.9. Pokažite, kako poiskati drugo najmanjšo vrednost v `BinaryHeap` ali v `MeldableHeap` v konstantnem času.

Naloga 10.10. Poiščite k-to najmanjšo vrednost v `BinaryHeap` ali v `MeldableHeap` v časovni zahtevnosti $O(k \log k)$. (Namig: Mogoče pomaga uporaba drugačne kopice.)

Naloga 10.11. Predpostavimo da imamo podanih `k` razporejenih seznamov, dolžine `n`. Z uporabo kopice, pokažite kako združiti urejene sezname v času $O(n \log k)$e. (Namig: Pomaga, če začnete s primerom $k = 2$".)

Poglavje 11

Algoritmi za urejanje

V tem poglavju se bomo pogovarjali o algoritmih, ki uredijo zbirko n elementov. To se lahko sliši kot čudna tema v knjigi podatkovnih struktur, ampak za to obstaja nekaj dobrih razlogov. Najočitnejši je ta, da sta dva od urejevalnih algoritmov (hitro urejanje in urejanje s kopico) tesno povezana s podatkovnima strukturama, ki smo ju že obdelali (naključno dvojiško drevo in kopice).

V prvem delu tega poglavja bo govora o algoritmih, ki uporabljajo zgolj primerjanje in sicer tri take algoritme s časovno zahtevnostjo $O(n \log n)$. Kot se izkaže, so vsi trije algoritmi asimptotično optimalni. Se pravi vsak algoritem, ki uporablja zgolj primerjanje izvede približno $n \log n$ primerjav v najslabšem kot tudi v povprečnem primeru.

Preden nadaljujemo velja izpostaviti, da lahko uporabimo katerikoli implementacijo urejene množice ali prioritetne vrste, ki smo jih predstavili v prejšnjih poglavjih, da dobimo algoritem za urejanje s časovno zahtevnostjo $O(n \log n)$. Naprimer, lahko uredimo n elemente tako, da izvedemo najprej n `add(x)` operacij, nato pa n `remove()` operacij na dvojiški ali zdržljivi kopici. Alternativno lahko uporabimo tudi n `add(x)` operacij na katerimkoli dvojiškim iskalnim drevesom, kjer nato izvedemo vmesno prečkanje (**??**), da dobimo elemente v urejenem vrstnem redu. Vendar si v obeh primerih naredimo veliko preglavic, da zgradimo strukturo, ki je nikoli ne uporabljamo v celoti. Urejanje je tako pomembna težava, da je vredno razviti direktne metode, ki so kot se le da hitre, preproste in prostorsko učnkovite.

Drugi del tega poglavja kaže, da ne obstaja časovnih zagotovil, če upo-

rabimo katerekoli druge operacije razen primerjave. Je pa res, da lahko s tabelnim indeksiranjem uredimo množico n števil v območju $\{0,\ldots,n^c-1\}$ s časovno zahtevnostjo $O(cn)$.

11.1 Urenanje s primerjanjem

V tem delu bomo predstavili tri algoritme za urejanje: urejanje z zlivanjem, hitro urejanje in urejanje s kopico. Vsak izmed teh treh algoritmov sprejme kot prvi argument tabelo a, ki jo uredi v naraščujočem vrstnem redu v (pričakovanem) času $O(n \log n)$. Vsi ti algoritmi delujejo *na osnovi primerjav*. Njihov drugi argument c je `Comparator` ki implementira metodo `compare(a,b)`. Ti algoritmi za urejanje nimajo privzetega tipa podatkov, ker izvajajo zgolj operacijo compare(a,b). Spomnimo se poglavja 1.2.4, kjer smo se naučili, da `compare(a,b)` vrača negativno vrednost če je $a < b$, pozitivno, če je vrednost $a > b$ in nič, če je $a = b$.

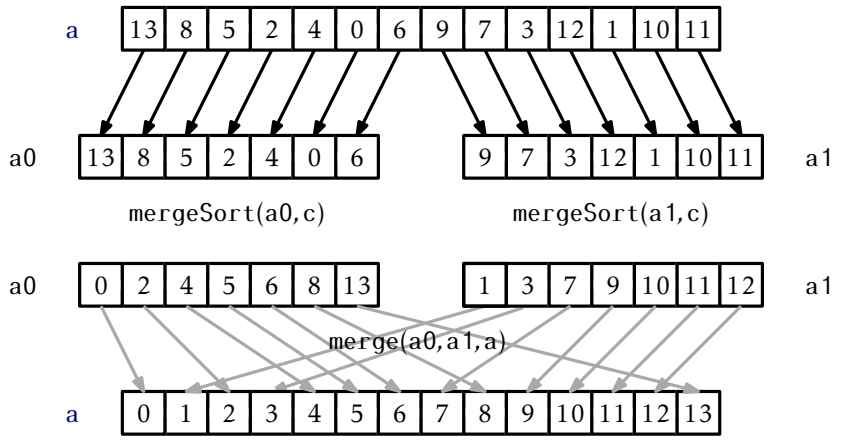
11.1.1 Urejanje z zlivanjem (merge-sort)

Algoritem *urejanja z zlivanjem* je klasičen primer rekurzivnega algoritma deli in vladaj. Če je dolžina od a največ 1, potem je a že urejen in zato ne naredimo ničesar. V nasprotnem primeru pa a razdelimo na dva dela, $a0 = a[0],\ldots,a[n/2-1]$ in $a1 = a[n/2],\ldots,a[n-1]$. Nato rekurzivno uredimo a0 in a1 ter ju nato združimo s čimer dobimo popolno urejeno tabelo a.

Algorithms

```
 mergeSort(array<T> &a) {
if (a.length <= 1) return;
array<T> a0(0);
array<T>::copyOfRange(a0, a, 0, a.length/2);
array<T> a1(0);
array<T>::copyOfRange(a1, a, a.length/2, a.length);
mergeSort(a0);
mergeSort(a1);
merge(a0, a1, a);
```

Primer 11.1.



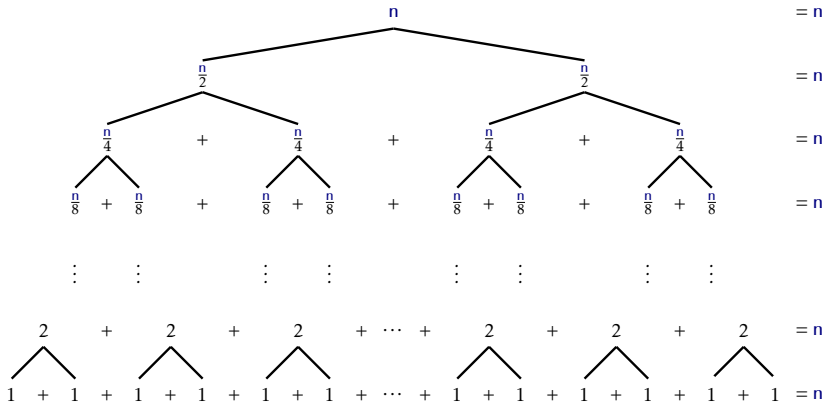
Slika 11.1: Izvedba `mergeSort(a,c)`

V primerjavi z urejanjem je zlivanje urejenih tabel `a0` in `a1` dokaj enostavno. Elemente dodajamo enega za drugim. Če je `a0` ali `a1` prazna, potem dodajamo naslednje elementi iz druge (ne prazne) tabele. Sicer vzamemo manjšega od nasledjih elementov iz obeh tabel in ga dodamo v `a`:

Algorithms

```
 merge(array<T> &a0, array<T> &a1, array<T> &a) {
int i0 = 0, i1 = 0;
for (int i = 0; i < a.length; i++) {
  if (i0 == a0.length)
    a[i] = a1[i1++];
  else if (i1 == a1.length)
    a[i] = a0[i0++];
  else if (compare(a0[i0], a1[i1]) < 0)
    a[i] = a0[i0++];
  else
    a[i] = a1[i1++];
}
```

Opazimo, da algoritem `merge(a0,a1,a,c)` v najslabšem primeru izvede $n-1$ primerjav, preden izprazne `a0` ali `a1`.



Slika 11.2: The merge-sort recursion tree.

Da bi lažje razumeli čas izvajanja urejanja z zlivanjem, si ga je najbolje predstavljati kot njegovo rekurzivno drevo. Zaenkrat predpostavimo, da je n potenca števila dve, tako da je $n = 2^{\log n}$, $\log n$ pa je celo število. Glej sliko 11.2. Urejanje z zlivanjem spremeni problem urejanja n elementov v dva problema urejanja $n/2$ elementov. Ta dva podproblema potem spremeni vsakega v dva nova podproblema, torej skupno v štiri probleme velikosti $n/4$. Te štiri nato razdelimo v osem podproblemov velikosti $n/8$ in tako dalje. Na koncu tega procesa $n/2$ podproblemov, vsakega velikosti dve, razdelimo v n problemov velikosti ena. Za vsak podproblem velikosti $n/2^i$ je čas zlivanja in kopiranja podatkov razreda $O(n/2^i)$. Ker imamo 2^i podproblemov velikosti $n/2^i$, je skupen čas reševanja problemov velikosti 2^i, če ne štejemo rekurzivnih klicev:

$$2^i \times O(n/2^i) = O(n) \ .$$

Iz tega sledi, da je skupen čas, ki ga porabi urejanje z zlivanjem:

$$\sum_{i=0}^{\log n} O(n) = O(n \log n) \ .$$

Dokaz za naslednji izrek je osnovan na prejšnji analizi, vendar pa moramo biti pazljivi zaradi primera, kadar n ni potenca števila 2.

Izrek 11.1. *Metoda* `mergeSort(a,c)` *teče v času* $O(n \log n)$ *in izvede največ* $n \log n$ *primerjav.*

Dokaz. Dokažemo z indukcijo po n. Osnovni primer, kadar je $n = 1$, je trivialen; če algoritem v obdelavo dobi tabelo dolžine 0 ali 1, to tabelo enostavno vrne, brez da bi izvedel kakršne koli primerjave.

Zlivanje dveh urejenih seznamov skupne dolžine n zahteva največ $n - 1$ primerjav. Naj $C(n)$ označuje največje možno število primerjav, ki jih izvede `mergeSort(a,c)` na tabeli `a` dolžine n. Če je n sodo število, potem za podproblema uporabimo indukcijsko hipotezo in s tem dobimo:

$$\begin{aligned} C(n) &\le n - 1 + 2C(n/2) \\ &\le n - 1 + 2((n/2)\log(n/2)) \\ &= n - 1 + n\log(n/2) \\ &= n - 1 + n\log n - n \\ &< n \log n \ . \end{aligned}$$

Če je n liho število pa je dokaz nekoliko bolj zapleten. V tem primeru uporabimo dve neenačbi, ki jih lahko enostavno dokažemo:

$$\log(x+1) \le \log(x) + 1 \ , \tag{11.1}$$

za vse $x \ge 1$ in:

$$\log(x + 1/2) + \log(x - 1/2) \le 2\log(x) \ , \tag{11.2}$$

za vse $x \ge 1/2$. Neenačbo (11.1) izpeljemo iz dejstva, da je $\log(x) + 1 = \log(2x)$, (11.2) pa izpeljemo iz dejstva, da je log konkavna funkcija. Ko vemo vse to, za lihi n velja:

$$\begin{aligned} C(n) &\le n - 1 + C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) \\ &\le n - 1 + \lceil n/2 \rceil \log \lceil n/2 \rceil + \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor \\ &= n - 1 + (n/2 + 1/2)\log(n/2 + 1/2) + (n/2 - 1/2)\log(n/2 - 1/2) \\ &\le n - 1 + n\log(n/2) + (1/2)(\log(n/2 + 1/2) - \log(n/2 - 1/2)) \\ &\le n - 1 + n\log(n/2) + 1/2 \\ &< n + n\log(n/2) \\ &= n + n(\log n - 1) \\ &= n \log n \ . \end{aligned}$$

□

11.1.2 Hitro urejanje (quicksort)

Hitro urejanje ali *quicksort* algoritem je še en klasični algoritem deli in vladaj . V nasprotju z algoritmom zlivanja (mergesort), ki združuje rešitvi dveh podproblemov, algoritem hitrega urejanja počne vse svoje delo vnaprej.

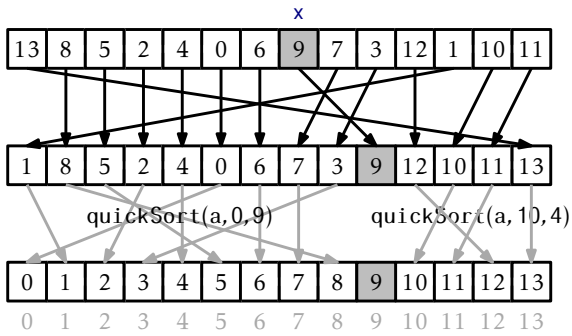
Algoritem lahko preprosto opišemo tako: Izberemo naključni delilni element, ki ga imenujemo *pivot*, `x`, ki ga dobimo iz `a`; Razdelek `a` je sestavljena iz sklopa elementov manjših od `x`, sklopa elementov enakih kot `x` in niz elementov večjih od `x`; na koncu rekurzivno razvrstimo prvi in tretji sklop števil v tem razdelku. Primer je prikazan na sliki 11.3.

Algorithms

```
 quickSort(array<T> &a) {
quickSort(a, 0, a.length);

 quickSort(array<T> &a, int i, int n) {
if (n <= 1) return;
T x = a[i + rand()%n];
int p = i-1, j = i, q = i+n;
// a[i..p]<x,  a[p+1..q-1]??x, a[q..i+n-1]>x
while (j < q) {
  int comp = compare(a[j], x);
  if (comp < 0) {       // move to beginning of array
    a.swap(j++, ++p);
  } else if (comp > 0) {
    a.swap(j, --q);  // move to end of array
  } else {
    j++;              // keep in the middle
  }
}
// a[i..p]<x,  a[p+1..q-1]=x, a[q..i+n-1]>x
quickSort(a, i, p-i+1);
quickSort(a, q, n-(q-i));
```

Vse to je narejeno na mestu, tako da namesto ustvarjanja kopij urejenih podseznamov, `quickSort(a,i,n,c)` metoda razvršča samo podseznam $a[i],\ldots,a[i+n-1]$. Prvotno kličemo to metodo kot



Slika 11.3: Primer izvajanja `quickSort(a,0,14)`

`quickSort(a,0,a.length,c)`.

V središču algoritma quicksort je algoritem delitve na mestu. Ta algoritem, brez uporabe dodatnega prostora, zamenja elemente v `a` in izračuna indekse `p` in `q` tako da:

$$a[i]\begin{cases} < x & \text{če } 0 \le i \le p \\ = x & \text{če } p < i < q \\ > x & \text{če } q \le i \le n-1 \end{cases}$$

Ta delitev, ki se opravi z `while` zanko v sami kodi, deluje s ponavljajočim povečanjem `p`-ja in zmanjševanjem `q`-ja ob ohranjanju prvega in zadnjega od teh pogojev (`p` in `q`). Ob vsakem koraku element na položaju `j` premaknemo na prvo mesto, ali pa na zadnje mesto. V prvih dveh primerih, je `j` povečan, v zadnjem primeru pa ne, ker nov element na položaju `j` še ni bil obdelan.

Quicksort algoritem je zelo tesno povezan z naključnim dvojiškim iskalnim drevesom, opisanem v poglavju 7.1. Pravzaprav, če poženemo quicksort algoritem nad `n` različnimi elementi, potem je quicksort-ovo rekurzivno drevo enako naključnemu iskalnemu drevesu. Da bi to videli, se moramo spomniti, kako gradimo naključno dvojiška iskalna drevesa. Najprej naključno izberemo element `x` in ga postavimo za koren drevesa. Nato vsak naslednji element primerjamo z `x`-om. Manjše elemente posta-

vljamo v levo poddrevo, večje pa v desno poddrevo.

S tem algoritmom izberemo nakjučni element x in takoj za tem primerjamo vse elemente s tem x-om. Najmanjše elemente postavimo na začetek polja, večje pa postavimo na konec. Quicksort algoritem nato rekurzivno uredi začetek in konec polja, medtem ko naključno dvojiško iskalno drevo rekurzivno vstavi manjše elemente v levo poddrevo korena in večje elemente v desno poddrevo korena.

Zgornje ujemanje med naključnim dvojiškim iskalnim drevesom in algoritmom hitrega urejanja lahko uporabimo za lemo 7.1

Lema 11.1. *Ko kličemo algoritem quicksort za urejanje polja, ki vsebuje cela števila* $0,\ldots,n-1$*, je pričakovano število primerjav elementa s pivot-om* $H_{i+1} + H_{n-i}$.

Malo seštevanja harmoničnih števil nam da naslednji izrek o času delovanja, katerega porabi algoritem:

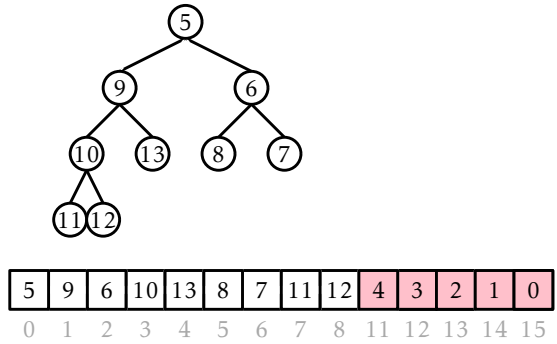
Izrek 11.2. *Ko quicksort algoritem uporabimo za urejanje polja z* n *različnimi elementi, pričakujemo največje število opravljenih primerjav* $2n\ln n + O(n)$.

Dokaz. Naj bo T število primerjav opravljenih z algoritmom quicksort, ko razvršča n različnih elementov. Z uporabo Leme 11.1, imamo:

$$\begin{aligned} \mathrm{E}[T] &= \sum_{i=0}^{n-1} (H_{i+1} + H_{n-i}) \\ &= 2\sum_{i=1}^{n} H_i \\ &\leq 2\sum_{i=1}^{n} H_n \\ &\leq 2n\ln n + 2n = 2n\ln n + O(n) \end{aligned}$$

□

Izrek 11.3 opisuje primer, kjer so razvrščeni elementi vsi različni. Ko vhodni seznam a, vsebuje podvojene elemente, pričakovani čas delovanja za hitro urejanje ni nič slabši, in je lahko celo boljši. Vedno ko je podvojeni element x izbran kot pivot a, vse pojavitve x-a združimo in jih kasneje ne vključimo v enega od dveh podproblemov.



Slika 11.4: Primer izvedbe heapSort(a, c).

Izrek 11.3. *Časovna zahtevnost metode* Quicksort(a, c) *je* $O(n \log n)$, *pričakovano število primerjav, ki jih opravi, je večinoma* $2n \ln n + O(n)$.

11.1.3 Urejanje s kopico (heap-sort)

Algoritem *Heap-sort* je še eden izmed algoritmov urejanja na mestu. Heap-sort uporablja dvojiško kopico, ki smo jo obravnavali v poglavju 10.1. Spomnimo se, da podatkovna struktura BinaryHeap"predstavlja kopico, ki je realizirana z enim seznamom. Urejanje s kopico pretvori vhodni seznam a v kopico in nato ponavljajoče izloča minimalno vrednost.

Bolj natančno povedano, kopica hrani n elementov v seznamu a na lokacijah $a[0], \ldots, a[n-1]$ z najmanjšo vrednostjo v korenu oz. $a[0]$. Po transformaciji v BinaryHeap urejanje s kopico ponavljajoče izmenjuje $a[0]$ in $a[n-1]$, ter kliče trickleDown(0), tako da so $a[0], \ldots, a[n-2]$ ponovno v kopičasti ureditvi. Ko se ta proces konča (ker je $n = 0$), so elementi a shranjeni v padajočem zaporedju. Sedaj a obrnemo, da dobimo rezultat. [1] Na sliki 11.4 je primer izvajanja heapSort(a, c).

BinaryHeap

```
void sort(array<T> &b) {
  BinaryHeap<T> h(b);
```

[1] Algoritem bi lahko prav tako redefiniral compare(x, y), tako da algoritem že na začetku vstavi elemente v naraščajočem vrstnem redu.

```
  while (h.n > 1) {
    h.a.swap(--h.n, 0);
    h.trickleDown(0);
  }
  b = h.a;
  b.reverse();
}
```

Ključna podrutina v heap-sort je konstruktor za pretvorbo urejenega seznama a v kopico. To bi z lahkoto storili v času $O(n \log n)$ s ponavljajočim klicem metode add(x) dvojiške kopice, a znamo to operacijo izvesti hitreje z uporabo bottom-up algoritma. Spomnimo se, da so v dvojiški kopici otroci a[i] shranjeni na položajih a[2i + 1] in a[2i + 2]. To namiguje, da elementi $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ nimajo otrok. Z drugimi besedami je vsak element $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ podkopica velikosti 1. Sedaj ko delamo od zadaj naprej, lahko kličemo metodo trickleDown(i) za vsak $i \in \{\lfloor n/2 \rfloor - 1, \ldots, 0\}$. To deluje, ker je do trenutka ko kličemo trickleDown(i) vsak od otrok a[i] koren podkopice. S tem ko kličemo trickleDown(i), nastavimo a[i] kot koren svoje podkopice.

BinaryHeap

```
BinaryHeap(array<T> &b) : a(0) {
  a = b;
  n = a.length;
  for (int i = n/2-1; i >= 0; i--) {
    trickleDown(i);
  }
}
```

Zanimivost te bottom-up strategije je, da je bolj učinkovita kot klicanje metode add(x) n-krat. Opazimo, da za n/2 elementov sploh ne delamo, za n/4 elementov kličemo trickleDown(i) nad podkopico, katere koren je $a[i]$ in je njena višina enaka 1. Za n/8 elementov kličemo metodo trickleDown(i) nad podkopici katere višina je enaka 2 in tako dalje. Ker je delo, ki ga izvaja trickleDown(i) sorazmerno višini podkopice a[i], je celotnega dela največ

$$\sum_{i=1}^{\log n} O((i-1)n/2^i) \leq \sum_{i=1}^{\infty} O(in/2^i) = O(n) \sum_{i=1}^{\infty} i/2^i = O(2n) = O(n) \ .$$

Predzadnja enakost sledi, ker je seštevek $\sum_{i=1}^{\infty} i/2^i$ po definiciji enak pričakovanemu številu glav ob metu kovanc ob uporabi leme 4.2.

Naslednji izrek opisuje zmogljivost metode `heapSort(a,c)`.

Izrek 11.4. *Metoda* `heapSort(a,c)` *se izvede v času* $O(n \log n)$ *in izvede največ* $2n \log n + O(n)$ *primerjav.*

Dokaz. Algoritem deluje v treh korakih: (1) Pretvorba a v kopico, (2) ponavljajoče izločanje najmanjšega elementa iz a in (3) obrne elemente v a. Ravno smo zatrdili da korak 1 potrebuje $O(n)$ časa za izvedbo in $O(n)$ primerjav. Korak 3 potrebuje $O(n)$ čaza za izvedbo in nič primerjav. Korak 2 izvede n klicev metode `trickleDown(0)`. i-ti klic se izvaja na kopici velikosti $n - i$ in izvede največ $2\log(n-i)$ primerjav. Če seštejemo preko i dobimo

$$\sum_{i=0}^{n-i} 2\log(n-i) \leq \sum_{i=0}^{n-i} 2\log n = 2n\log n$$

S tem ko dodamo število izvedenih primerjav v vsakem od treh korakov dokončamo dokaz. □

11.1.4 Spodnja meja algoritmov za urejanje, temelječih na primerjavah

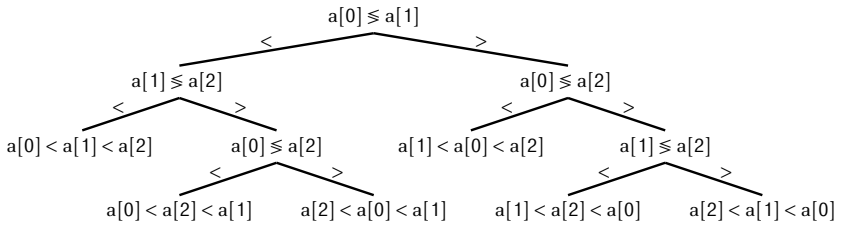
Sedaj smo videli tri primerjalne algoritme za urejanje, ki imajo časovno zahtevnost $O(n \log n)$. Čas je da se vprašamo, če obstaja hitrejši algoritem. Kratek odgovor, je ne. Če je edina dovoljena operacija primerjava dveh elementov a, potem ni algoritma, ki se lahko izogne približno $n \log n$ primerjavam. To ni težko dokazati in izhaja iz

$$\log(n!) = \log n + \log(n-1) + \cdots + \log(1) = n\log n - O(n) \ .$$

(Dokaz te formule je 11.10.)

Najprej bomo pozornost namenili determinističnim algoritmom, kot sta razvrščanje z zlivanjem in urejanje s kopico. Predstavljajte si, da tak algoritem uporabimo za urejanje n različnih elementov.

Pri dokazovanju spodnje meje je ključno opažanje, da je za deterministične algoritme z enako vrednostjo n, prva primerjava vedno enaka. Na primer pri `heapSort(a,c)`, ko je n liho, je prvi klic `trickleDown(i)` s vrednostjo $i = n/2 - 1$, in prva primerjava med elementoma $a[n/2-1]$ in $a[n-1]$.



Slika 11.5: Primerjalno drevo za urejanje polja $a[0], a[1], a[2]$ dolžine $n = 3$.

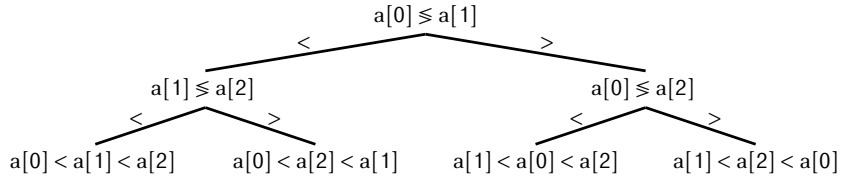
Ker se elementi ne ponavljajo, ima prva primerjava samo dva možna izida. Druga primerjava pa je lahko odvisna od prve. Tretja je pa lahko odvisna od prve in druge in tako naprej. Na ta način, si lahko kateri koli deterministični primerjalni algoritem za urejanje predstavljamo kot *comparison tree* s korenom. Vsako notranje vozlišče u, tega drevesa, je označeno s parom indeksov $u.i$ in $u.j$. Če je $a[u.i] < a[u.j]$, algoritem nadaljuje pot po levem pod drevesu, drugače pa po desnem pod drevesu. Vsak list w, je označen s permutacijo $w.p[0], \ldots, w.p[n-1]$ pri $0, \ldots, n-1$. To permutacijo potrebujemo za urejanje polja a, če primerjalno drevo obišče ta list. To je,

$$a[w.p[0]] < a[w.p[1]] < \cdots < a[w.p[n-1]] \ .$$

Primer primerjalnega drevesa za polje dolžine $n = 3$, je prikazano na sliki 11.5.

Primerjalno drevo za algoritem za urejanje, nam pove vse o njem. Pove nam natančno zaporedje primerjav, ki jih bo algoritem izvedel na nekem polju a, ki ima n različnih elementov, in nam pove, kako bo algoritem spremenil vrstni red polja a, da ga bo uredil. Posledično, mora primerjalno drevo vsebovati vsaj $n!$ listov. Če nima, pomeni da obstajata 2 različni permutaciji, ki vodita do istega lista, zato, algoritem ne uredi pravilno vsaj ene od permutacij.

Na primer, primerjalno drevo prikazano na 11.6 ima samo $4 < 3! = 6$ listov. Če pregledamo drevo, opazimo, da za polji z elementi $3, 1, 2$ in $3, 2, 1$ oba vodita do istega lista. Za polje $3, 1, 2$ dobimo pravilen izhod $a[1] = 1, a[2] = 2, a[0] = 3$. Ampak če imamo na vhodu $3, 2, 1$, nas napačno vodi do $a[1] = 2, a[2] = 1, a[0] = 3$. To vodi do osnovne spodnje meje za



Slika 11.6: Primerjalno drevo, ki ne uredi pravilno vseh možnih vhodnih podatkov.

urejevalne algoritme, ki temeljijo na primerjavah.

Izrek 11.5. *Za kateri koli deterministični algoritem za urejanje $\mathcal{A}$, ki temelji na primerjavah, in za katero koli celo število* $n \geq 1$*, obstaja tako vhodno polje* a *dolžine* n*, da se izvede vsaj* $\log(n!) = n \log n - O(n)$ *primerjav, ko urejamo* a*.*

Dokaz. Primerjalno drevo definirano kot $\mathcal{A}$, mora imeti vsaj n! listov. Preprost dokaz z indukcijo nam pokaže, da ima vsako dvojiško drevo s k listi, višino vsaj $\log k$. Posledično mora imeti primerjalno drevo $\mathcal{A}$ list w, ki ima globino vsaj log(n!) in obstaja tako vhodno polje a, da vodi do tega lista. Polje a je tako, da $\mathcal{A}$ izvede vsaj log(n!) primerjav. □

Izrek 11.5 govori o determinističnih algoritmih, kot sta razvrščanje z zlivanjem in urejanje s kopico. Kaj pa če imamo naključen algoritem kot je hitro urejanje? Ali bi lahko naključen algoritem bil boljši od spodnje meje log(n!) primerjav? Odgovor, je ponovno, ne. To lahko dokažemo, če na to, kaj je naključen algoritem, pomislimo malo drugače.

Predvidevali bomo, da je naše odločitveno drevo "očiščeno": Vsako vozlišče, ki ga ne moremo obiskati z nekim vhodnim poljem a, odrežemo. To pomeni, da bo imelo drevo natanko n! listov. Ima vsaj n! listov, ker drugače ne bi uredilo polje pravilno. Ima največ n! listov, ker za vsako od n! permutacij n elementov, obstaja natanko en koren, ki vodi do tega lista.

Na algoritem za urejanje $\mathcal{R}$, ki ima naključnost, lahko gledamo kot deterministični algoritem, ki sprejme 2 vhoda: Polje a, ki ga bomo uredili, in dolgo zaporedje $b = b_1, b_2, b_3, \ldots, b_m$ naključnih realnih števil v obsegu $[0, 1]$. Naključna števila potrebujemo za naključnost v algoritmu. Ko želi met kovanca ali naključno odločitev, uporabi eno od vrednosti iz b. Na

primer, če želimo izračunati indeks prvega pivota pri hitrem urejanju, lahko algoritem uporabi formulo $\lfloor nb_1 \rfloor$.

Če za b uporabimo neko določeno zaporedje $\hat{b}$, potem $\mathcal{R}$ postane deterministični algoritem za urejanje, $\mathcal{R}(\hat{b})$, ki ima primerjalno drevo $\mathcal{T}(\hat{b})$. Če za a izberemo naključno permutacijo iz $\{1,\ldots,n\}$, potem je to ekvivalentno izbiri naključnega lista w, od n! listov od $\mathcal{T}(\hat{b})$.

Naloga 11.12 zahteva dokaz, da če izberemo naključen list iz dvojiškega drevesa, ki ima k listov, potem je pričakovana globina tega lista vsaj $\log k$. Zaradi tega je pričakovana vrednost primerjav (determinističnega) algoritma $\mathcal{R}(\hat{b})$, ki sprejme za vhod naključno permutacijo iz $\{1,\ldots,n\}$, vsaj $\log(n!)$. To velja za vsako izbiro $\hat{b}$, zato to velja tudi za $\mathcal{R}$. To zaključi dokaz o spodnji meji za naključni algoritem.

Izrek 11.6. *Za vsako celo število $n \geq 1$ in kateri koli (deterministični ali naključni) primerjalni algoritem za urejanje $\mathcal{A}$, je pričakovana vrednost primerjav, ki jih stori algoritem pri naključni permutaciji $\{1,\ldots,n\}$, vsaj* $\log(n!) = n\log n - O(n)$.

11.2 Counting Sort and Radix Sort

V tem delu preučujemo dva urejevalna algoritma ki nista bazirana na primerjanju. Algoritma sta specializirana za ločevanje manjših celih števil, ter se izogneta spodnji meji izreka 11.5 z uporabo (nekaterih delov) elementov v a kot indeksov v polju. Razmislite o izrazu

$$c[a[i]] = 1 \quad .$$

Ta izraz se izvrši v konstantnem času , ampak ima c.length možnih različnih rezultatov, odvisno od vrednosti a[i]. To pomeni da izvršitev algoritma ki poda tako izjavo ni mogoče modelirati kot dvojiško drevo. To je glavni razlog da so algoritmi v tem delu zmožni urejati hitreje kot algoritmi bazirani na primerjavah.

11.2.1 Counting Sort

Recimo da imamo polje a sestavljeno iz n celih števil, vse v obsegu $0,\ldots,k-1$. Algoritm *urejanje s štetjem* urejanja a z uporabo pomožnega polja števcev

c. Ven dobimo urejeno verzijo polja a kot pomožno polje b.

Ideja pri urejanju s štetjem je preprosta: Za vsak i ∈ {0,...,k − 1}, prešteje število pojavitev i v a in to shrani v c[i]. Po urejanju izhodni produkt zgleda c[0] ponovitev 0, sledi c[1] pojavitev 1, sledi še c[2] pojavitev 2,..., sledi c[k − 1] pojavitev k − 1. Koda ki to izvrši je zelo spretna, njeno delovanje je ilustrirano na sliki 11.7:

Algorithms

```
countingSort(array<int> &a, int k) {
array<int> c(k, 0);
for (int i = 0; i < a.length; i++)
  c[a[i]]++;
for (int i = 1; i < k; i++)
  c[i] += c[i-1];
array<int> b(a.length);
for (int i = a.length-1; i >= 0; i--)
  b[--c[a[i]]] = a[i];
a = b;
```

Prva for zanka v tej kodi nastavi vsak števec c[i] tako, da šteje število ponovitev i v a. Z uporabo vrednosti a kot indeks se ti števci lahko vsi izračunajo v času $O(n)$ z eno samo for zanko. Na tej točki bi lahko uporabili c da direktno zapolnimo izhodni polje b. Vendar pa to ne bi delovalo če bi imeli elementi polja a povezane podatke. Zato porabimo nekaj več napora da prekopiramo elemente polja a v b.

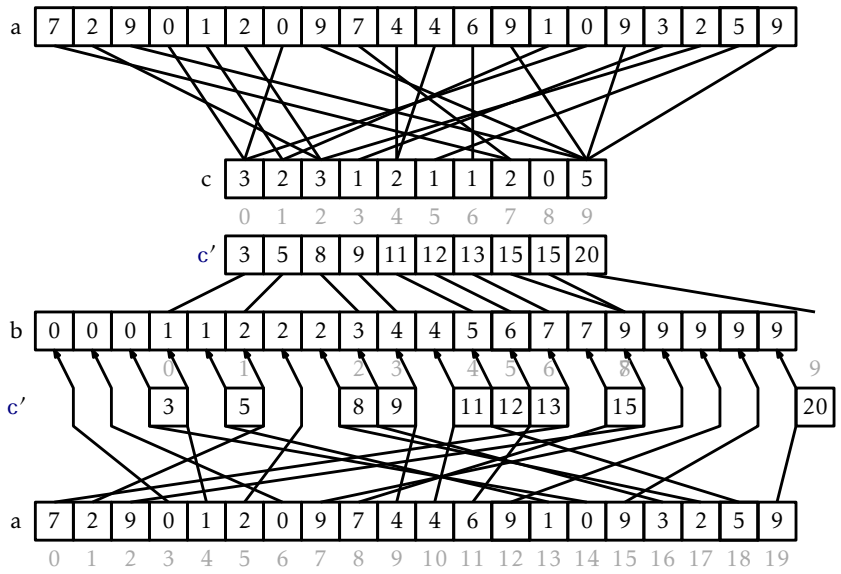
Naslednja for zanka, ki potrebuje $O(k)$ časa, izračuna tekoče vsote števcev tako da c[i] postane število elementov v a, ki so manjši ali enaki i. Še posebej za vsak i ∈ {0,...,k − 1}, bo izhodno polje b imelo

$$b[c[i-1]] = b[c[i-1]+1] = \cdots = b[c[i]-1] = i \ .$$

Na koncu algoritem pregleda a vzvratno tako, da postavi svoje elemente v pravem vrstnem redu v izhodno polje b. Ko pregleduje, postavi element a[i] = j na pozicijo b[c[j] − 1] in vrednost c[j] se zmanjša.

Izrek 11.7. *Metoda* countingSort(a,k) *lahko uredi polje* a, *ki vsebuje* n *števil v množici* {0,...,k − 1} *v času* $O(n+k)$.

Algoritem urejanje s štetjem ima prijetno lastnost in sicer da je *stabilen* relativni vrstni red enakih elementov. Če imata dva elementa a[i]in a[j]



Slika 11.7: Operacija urejanja s štetjem na polju velikosti $n = 20$, ki shrani $0, \ldots, k-1 = 9$ števil.

isto vrednost, in i < j, potem se bo a[i] pojavil pred a[j] v b. To bo uporabno v naslednjem poglavju.

11.2.2 Radix-Sort

Urejanje s štetjem je zelo efektivna metoda za urejanje polja števil , ko je dolžina polja n ni veliko manjša kot maksimalna vrednost k − 1, ki se pojavi v polju. Algoritem *korensko urejanje* , ki ga sedaj opisujemo uporablja več prehodov algoritma urejanja s štetjem, kar dopušča večji razpon maksimalnih vrednosti.

Korensko urejanje ureja w-bitna števila z uporabo w/d prehodov urejanja s štetjem, da ta števila uredi po d bitih naenkrat. [2] Natančneje, korensko urejanje prvo uredi števila po najmanj pomembnih d bitih nato po naslednjih d pomembnejših bitih in tako naprej, v zadnjem prehodu so števila urejena po najpomembnejših d bitih.

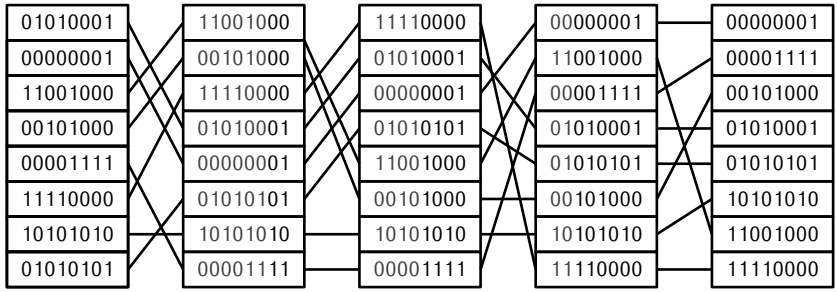
Algorithms

```
 radixSort(array<int> &a) {
int d = 8, w = 32;
for (int p = 0; p < w/d; p++) {
  array<int> c(1<<d, 0);
  // the next three for loops implement counting-sort
  array<int> b(a.length);
  for (int i = 0; i < a.length; i++)
    c[(a[i] >> d*p)&((1<<d)-1)]++;
  for (int i = 1; i < 1<<d; i++)
    c[i] += c[i-1];
  for (int i = a.length-1; i >= 0; i--)
    b[--c[(a[i] >> d*p)&((1<<d)-1)]] = a[i];
  a = b;
}
```

(V tej kodi, izraz (a[i] >> d * p)&((1 << d) − 1) izloči število katerega dvojiška predstavitev je dana z biti (p + 1)d − 1,...,pd od a[i].) Primer korakov tega algoritma je prikazan na skiki 11.8.

Ta neverjetni algoritem ureja pravilno, ker je urejanje s štetjem stabilen algoritem za urejanje. Če sta x < y dva elementa polja a in če ima

[2] Privzamemo da d deli w, v nasprotnem primeru lahko w povečamo na d⌈w/d⌉.



Slika 11.8: Uporaba korenskega urejanja za urejanje $w = 8$-bitnega števila z uporabo štirih prehodov z uporabo urejanja s štetjem na $d = 2$-bitnih številih

najpomembnejši bit pri katerem se x razlikuje od y index r, potem bo x postavljen pred y med prehodom $\lfloor r/d \rfloor$ in vsi naslednji prehodi ne bodo spremenili relativnega vrstnega reda x in y.

Korensko urejene opravi w/d prehodov urejanja s štetjem. Vsak prehod porabi $O(n+2^d)$ časa. Torej, zahtevnost korenskega urejanja je izražena v naslednjem izreku.

Izrek 11.8. *Za katerokoli število* $d > 0$, `radixSort(a,k)` *metoda lahko uredi polje* `a`, *ki vsebuje* n w*-bitnih števil v času* $O((w/d)(n + 2^d))$.

Če namesto elementov polja v razponu $\{0,\ldots,n^c - 1\}$ vzamemo $d = \lceil \log n \rceil$ dobimo nasledno različico izreka 11.8.

Posledica 11.1. *Metoda* `radixSort(a,k)` *lahko uredi polje* `a`, *ki vsebuje* n *številskih vrednosti v razponu* $\{0,\ldots,n^c - 1\}$ *v času* $O(cn)$.

11.3 Diskusija in naloge

Sortiranje je osnovni algoritemski problem v računalništvu in ima dolgo zgodovino. Knuth [**?**] pripisuje alogritem sortiranja z zlivanjem von Neumann(1945). Hitro urejanje je predstavil Hoare [**?**]. Originalno urejanje s kopico je last Williams [**?**], različico predstavljeno tu (v kateri se kopica gradi iz spodaj nazvgor v $O(n)$ času) pa je zasnoval Floyd [**?**]. Spodnje meje za urejanja, temelječa na primerjavah, se zdijo zastarele. Naslednja tabela povzame časovne zahtevnosti tovrstnih algoritmov:

	primerjave	na mestu
Urejanje z zlivanjem	$n \log n$ najslabši primer	Ne
Hitrjo urejanje	$1.38n \log n + O(n)$ pričakovano	Da
Urejanje s kopico	$2n \log n + O(n)$ najslabši primer	Da

Vsi algoritmi urejanje s primerjanjem imajo svoje prednosti in slabosti. Urejanje z zlivanjem naredi najmanj primerjav in se ne zanaša na naključnost. Na žalost, uporablja pomožno tabelo med fazo zlivanja. Dodeljevanje pomnilnika tej tabeli je lahko drago in potencialno usodno za algoritem, če je količina pomnilnika omejena. Hitro urejanje ureja elemente *na mestu* in je blizu na drugem mestu v številu primerjav, ampak vsebuje naključnost, zato čas izvajanja ni vedno zagotovljen. Urejanje s kopico naredi največ primerjav, ampak deluje na mestu in je deterministično.

Obstaja primer, v katerem je urejanje s kopico očiten zmagovalec; to se zgodi pri urejanju povezanega seznama. V tem primeru, ne potrebujemo pomožne tabele; dva urejena povezana seznama, se zelo lahko zljieta v en urejen povezan seznam z uporabo manipulacije kazalcev (glej 11.2).

Algoritma urejanje s štetjem in korensko urejanje, opisana tu, temeljita na Sewardovem principu [**?**, Section 2.4.6]. Ampak različice korenskega urejanja so v uporabi že od dvajsetih let 20. stoletja za razvrščanje luknjanih kartic z uporabo strojev. Te stroji lahko razvrstijo kup kartic v dva kupa, glede na obstoj (ali neobstoj) luknjice na specifični lokaciji na kartici. Ponovitev tega procesa, za drugo luknjico, nam da implementacijo korenskega urejanja.

Na koncu, opazimo, da urejanje s štetjem in korenskega urejanja lahko uporabimo, tudi za urejanje drugih števil poleg pozitivnih celih števil. Enostavne spremembe urejanja s štetjem lahko uredijo cela števila v poljubnem intervalu $\{a, \ldots, b\}$, v $O(n + b - a)$ času. Podobno, korensko urejanje, lahko ureja cela števila na enakem intervalu v $O(n(\log_n(b-a))$ času. Na koncu, lahko oba algoritma uporabimo za urejanje števil s plavajočo vejico v zapisu IEEE 754 plavjoče vejice. To lahko naredimo zato, ker je zapis IEEE zasnovan tako, da dovoljuje primerjavo dveh števil s plavajočo vejico glede na njuni vrednosti, kot če bi bili celi števili v predznačeni dvojiški predstavitvi [**?**].

Naloga 11.1. Ilustriraje izvedbo urejanje z zlivanjem in urejanja s kopico

nad vhodno tabelo s števili $1, 7, 4, 6, 2, 8, 3, 5$. Naredite vzorčno ilustracijo ene možnosti izvedbe hitrega urejanja nad isto tabelo.

Naloga 11.2. Implementirajte verzijo algoritma urejanja z zlivanjem, ki uredi dvojno povezan seznam brez uporabe pomožne tabele. (Glej nalogo 3.13.)

Naloga 11.3. Nekatere implementacije `quickSort(a, i, n, c)` vedno uporabljajo `a[i]` kot pivot. Podajte primer vhodne tabele dolžine `n` nad katero bi taka implementacija izvedla $\binom{n}{2}$ primerjav.

Naloga 11.4. Nekatere implementacije `quickSort(a, i, n, c)` vedno uporabljajo `a[i + n/2]` kot pivot. Podajte primer vhodne tabele dolžine `n` nad katero bi taka implementacija izvedla $\binom{n}{2}$ primerjav.

Naloga 11.5. Pokažite, da za katerokoli implementacijo `quickSort(a, i, n, c)`, ki izbere pivot deterministično, ne da pogleda katerokoli vrednost v `a[i]`,..., `a[i + n − 1]` obstaja vhodna tabela dolžine `n`, ki povzroči, da ta izvede $\binom{n}{2}$ primerjav.

Naloga 11.6. Načrtujte `Comparator`, `c`, ki lahko podate kot argument funkciji `quickSort(a, i, n, c)`, in bi povzročil $\binom{n}{2}$ primerjav. (Namig: Vašemu Comparator-ju ni potrebno gledati vrednosti, ki se primerjajo.)

Naloga 11.7. Natančneje od dokaza 11.3 analizirajte pričakovano število primerjav, ki jih naredi Quicksort. Dokažite, da je pričakovano število primerjav $2nH_n - n + H_n$.

Naloga 11.8. Opišite vhodno tabelo, ki povzroči, da urejanje s kopico, naredi največ $2n \log n - O(n)$ primerjav. Utemeljite vaš odgovor.

Naloga 11.9. Najdite nek drug par premutacij $1, 2, 3$, ki niso pravilno urejene z drevesom primerjav v 11.6.

Naloga 11.10. Dokažite, da $\log n! = n \log n - O(n)$.

Naloga 11.11. Dokažite, da je dvojiško drevo s k listi visoko vsaj $\log k$.

Naloga 11.12. Dokažite, da je pri izbiri naključnega lista v dvojiškem drevesu s k listi pričakovana višina lista najmanj $\log k$.

Naloga 11.13. Implementacija `radixSort(a, k)` podana tukaj, deluje ko vhodna tabela, `a` vsebuje samo cela števila. Napišite verzijo, ki deluje za predznačena cela števila.

Poglavje 12

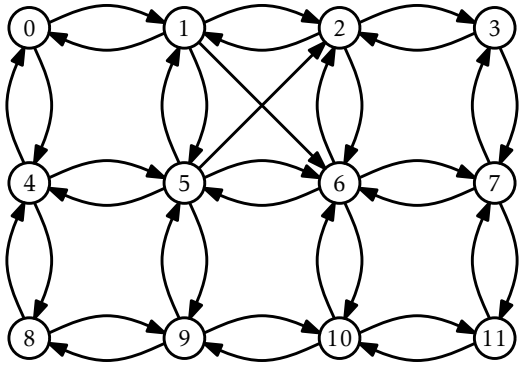
Grafi

V tem poglavju bomo preučili dva načina predstavitve grafov in temeljne algoritme, ki uporabljajo omenjeni predstavitvi.

Matematično je *(usmerjen) graf* par $G = (V, E)$ kjer je V množica *vozlišč* in E množica urejenih parov vozlišč imenovanih *povezave*. Povezava (i, j) je *usmerjena* od i do j; i se imenuje *izvor* povezave , j pa *ponor*. *Pot* v G je zaporedje vozlišč $v_0, \ldots, v_k$ tako, da so za vsak $i \in \{1, \ldots, k\}$ vse povezave (v_{i-1}, v_i) prisotne v E. Pot $v_0, \ldots, v_k$ je *cikel* če je tudi pot (v_k, v_0) prisotna v E. Pot (ali cikel) je *enostaven* če so tudi njegova vozlišča unikatna. Če je pot iz neke točke v_i do neke točke v_j, potem pravimo, da je v_j *dosegljiva* iz v_i. Primer grafa je prikazan na sliki 12.1.

Zaradi svoje zmogljivosti pri izdelavi modela raznih pojavov, imajo grafi ogromno število aplikacij. Obstajajo številni primeri. Računalniška omrežja lahko modeliramo v nek graf, kjer vozlišča (točke) predstavljajo računalnike in povezave predstavljajo (direktno) komunikacijsko pot med dvema računalnikoma. Tudi ceste v nekem mestu lahko predstavimo kot neki graf, kjer vozlišča predstavljajo križišča ter povezave predstavljajo ulice.

Primeri, ki so malo manj očitni, se pojavijo ko spoznamo, da grafe lahko modeliramo v pare kjer nimamo nobenih skupnih odnosov med sabo. Na primer v univerzi imamo lahko *konfliktni graf* urnika kjer vozlišča predstavljajo predavanja na univerzi in povezava (i, j) obstaja samo v primeru, če je prisoten vsaj en študent, ki hodi na predmet i in na predmet j. Tako ena povezava prikaže, da izpit za predmet i ne more na noben način biti načrtovan ob istem času tudi za predmet j.



Slika 12.1: Graf z dvanajstimi vozlišči. Vozlišča so narisana kot oštevilčeni krogci ter povezave so narisani kot usmerjene krivulje od izvora do ponora.

V tem poglavju nam `n` predstavlja število vozlišč v množici G in `m` število povezav v množici G. To pomeni, da `n` $= |V|$ in `m` $= |E|$. Poleg vsega tega pa predpostavimo, da je $V = \{0, \ldots, \texttt{n} - 1\}$. Katerekoli druge podatke, ki bi jih radi povezali z elementi množice V, lahko torej hranimo v tabeli dolžine `n`.

Tipične operacije nad grafi so:

- `addEdge(i, j)`: Doda povezavo (`i`, `j`) v E.
- `removeEdge(i, j)`: Zbriši povezavo (`i`, `j`) iz E.
- `hasEdge(i, j)`: Poišče povezavo (`i`, `j`) $\in E$
- `outEdges(i)`: Vrne `List` (seznam) celih števil `j` od (`i`, `j`) $\in E$
- `inEdges(i)`: Vrne `List` (seznam) celih števil `j` od (`j`, `i`) $\in E$

Vedeti je treba, da takšne operacije ni težko implementirati na učinkovit način. Na primer, prve tri operacije so lahko uporabljene direktno z uporabo `USet`, na tak način, da se lahko izvajajo v konstantnem pričakovanem času z uporabo razpršenih tabel (predstavljene v poglavju 5). Zadnji dve operaciji pa so lahko implementirane v konstantnem času s shranjevanjem, tako da za vsako vozlišče shranimo še seznam sosednjih vozlišč.

Različne aplikacije nad grafi imajo različne minimalne zahteve po hitrosti izvajanja operacij in potrebnega prostora. Idealno bi bilo, če bi uporabili najenostavnejšo izvedbo grafov, ki bi zadovoljila vse vrste aplikacij. V nadaljevanju si bomo pogledali dve najpogostejši izvedbi grafov.

12.1 AdjacencyMatrix: Predstavitev grafov z uporabo matrik

Matrika sosednosti je način za predstavitev n vozlišč grafa $G = (V, E)$ z matriko $n \times n$, a, kjer so notranji elementi tipa "boolean".

AdjacencyMatrix
```
int n;
bool **a;
```

Vnos elementa matrike a[i][j] je definiran kot

$$\texttt{a[i][j]} = \begin{cases} \texttt{true} & \text{if } (\texttt{i}, \texttt{j}) \in E \\ \texttt{false} & \text{otherwise} \end{cases}$$

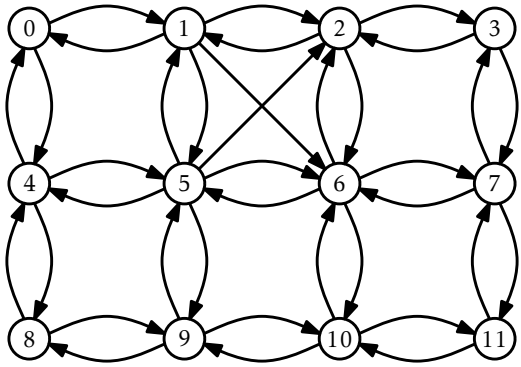
Matrika sosednosti za graf iz slike 12.1, je prikazana na sliki 12.2.

Tu je prikazana operacija addEdge(i, j), removeEdge(i, j) in hasEdge(i, j), ki vrne vrednost elementa a[i][j] matrike:

AdjacencyMatrix
```
void addEdge(int i, int j) {
  a[i][j] = true;
}
void removeEdge(int i, int j) {
  a[i][j] = false;
}
bool hasEdge(int i, int j) {
  return a[i][j];
}
```

Vse zgornje operacije trajajo $O(1)$ časa.

Izvajanje matrike sosednosti je slabše za operaciji outEdges(i) in inEdges(i). Da bi jo lahko implementirali, je potrebno preveriti vseh n vnosov v ustrezni vrstici oz. stolpcu iz a, in zbrati vse indekse j, kjer je a[i][j] oziroma a[j][i] resnična.



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	0	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	1	0	0	0	0
4	1	0	0	0	0	1	0	0	1	0	0	0
5	0	1	1	0	1	0	1	0	0	1	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0
7	0	0	0	1	0	0	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0
10	0	0	0	0	0	0	1	0	0	1	0	1
11	0	0	0	0	0	0	0	1	0	0	1	0

Slika 12.2: Graf in njegova matrika sosednosti.

AdjacencyMatrix
```
void outEdges(int i, List &edges) {
  for (int j = 0; j < n; j++)
    if (a[i][j]) edges.add(j);
}
void inEdges(int i, List &edges) {
  for (int j = 0; j < n; j++)
    if (a[j][i]) edges.add(j);
}
```

Časovna zahtevnost teh operacij je $O(n)$.

Druga slaba lastnost sosednostnjih matrik je, da so velike. V matriki je shranjeno $n \times n$ boolean vrednosti, kar pomeni, da potrebujemo vsaj n^2 bitov prostora v pomnilniku. Implementacija tu uporablja dejansko eno matriko z vrednostmi tipa boolean, tako da dejansko zavzame n^2 bajtov pomnilnika. Bolj previdna izvedba shrani w vrednosti boolean v vsako pomnilniško besedo in tako zmanjša porabo na $O(n^2/w)$ besed pomnilnika..

Izrek 12.1. *Podatkovna struktura* `AdjacencyMatrix`*, implementira vmesnik za grafe (v angleščini:* `Graph` *interface).* `AdjacencyMatrix` *podpira naslednje operacije*

- `addEdge(i, j)`, `removeEdge(i, j)`, *and* `hasEdge(i, j)` *v konstantnem času na operacijo; in*
- `inEdges(i)`, *in* `outEdges(i)` *v času* $O(n)$ *na operacijo.*

`AdjacencyMatrix` *zasede* $O(n^2)$ *prostora.*

Kljub visoki zahtevi po prostoru in neučinkovitega delovanja operacij `inEdges(i)` in `outEdges(i)`, je `AdjacencyMatrix` lahko še vedno uporabna za nekatere aplikacije. Še posebaj, ko je graf G gost (*dense*), kar pomeni, da ima približno n^2 povezav, potem mora zavzeti n^2 prostora kar je še vedno sprejemljivo.

Podatkovna struktura `AdjacencyMatrix` se pogosto uporablja, saj se operacije nad matriko `a` lahko uporabljajo za definiranje lastnosti grafa G. To je argument, ki se predela na tečaju za algoritme, ampak oglejmo si vsaj eno lastnost: če obravnavamo vhod kot neko celo število `a` (1 za true

in 0 za false) in pomnožimo matriko a samo s seboj z uporabo operacije množenja matrik, potem kot rezultat dobimo matriko a^2. Po definiciji je za množenje matrik

$$a^2[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot a[k][j] \ .$$

Interpretacija zgornje vsote glede na graf G bi bila naslednja: formula prešteje vsa vozlišča k, za katere G vsebuje obe povezavi (i,k) in (k,j). Prešteje vse poti iz i do j (skozi vmesno vozlišče k) katerih dolžina je 2. Omenjena značilnost je temelj algoritma za izračun najkrajših poti med vsemi vozlišči v G, ki namesto $O(n)$ potrebuje le $O(\log n)$ matričnih množenj.

12.2 AdjacencyLists: Predstavitev grafov s seznamom sosednosti

Seznam sosednosti - ponazoritev grafov vzame pristop bolj usmerjen v vozlišča. Obstaja veliko možnih izvedb seznamov sosednosti. V tem poglavju predstavljamo preprosto izvedbo. Na koncu razdelka razpravljamo o različnih možnostih. V seznamu sosednosti je graf $G = (V,E)$ predstavljen kot polje, adj, seznamov. Seznam adj[i] vsebuje vse sosede vozlišča i. To je vsak j, kjer velja $(i,j) \in E$.

AdjacencyLists

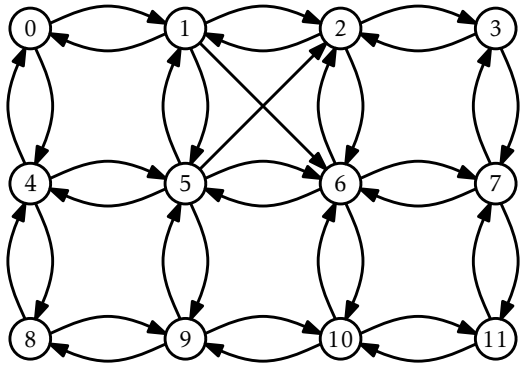
```
int n;
List *adj;
```

(Primer je pokazan na 12.3.) V tej specifični implementaciji, predstavimo vsak seznam adj kot a subclass of ArrayStack, ker želimo doseči konstanten čas dostopov do pozicij. Mogoče so tudi drugačne opcije. Ena opcija je izvedba adj kot DLList.

Operacija addEdge(i, j) doda vrednost j na seznam adj[i]:

AdjacencyLists

```
void addEdge(int i, int j) {
  adj[i].add(j);
}
```



0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	0	1	5	6	4	8	9	10
4	2	3	7	5	2	2	3	9	5	6	7
	6	6		8	6	7	11		10	11	
	5				9	10					
					4						

Slika 12.3: A graph and its adjacency lists

To se izvede v konstantem času.

Operacija `removeEdge(i, j)` pregleda seznam `adj[i]` dokler ne najde `j` in ga odstrani iz seznama:

AdjacencyLists
```
void removeEdge(int i, int j) {
  for (int k = 0; k < adj[i].size(); k++) {
    if (adj[i].get(k) == j) {
      adj[i].remove(k);
      return;
    }
  }
}
```

To se izvede v $O(\deg(i))$ času, kjer $\deg(i)$ (*stopnja vozlišča* `i`) prešteje število robov v E, ki imajo `i` za njihov izvor.

Operacija `hasEdge(i, j)` je podobna; pregleda seznam `adj[i]` dokler ne najde `j` (in vrne true), ali doseže konec seznama (in vrne false):

AdjacencyLists
```
bool hasEdge(int i, int j) {
  return adj[i].contains(j);
}
```

To se izvede v $O(\deg(i))$ času.

Operacija `outEdges(i)` je zelo preprosta;

AdjacencyLists
```
void outEdges(int i, ListT &edges) {
  for (int k = 0; k < adj[i].size(); k++)
    edges.add(adj[i].get(k));
}
```

To se očitno izvede v $O(\deg(i))$ času.

Operacija `inEdges(i)` naredi več dela. Operacija pregleda vsa vozlišča j, in če povezava `(i, j)` obstaja, doda `j` na izhodni seznam.

AdjacencyLists
```
void inEdges(int i, ListT &edges) {
  for (int j = 0; j < n; j++)
    if (adj[j].contains(i))  edges.add(j);
}
```

Operacija je zelo počasna. Pregleda seznam sosednosti vsakega vozlišča in se izvede v času $O(n+m)$.

Naslednji izrek povzema delovanje zgornje podatkovne strukture:

Izrek 12.2. *Podatkovna struktura* `AdjacencyLists` *implementira vmesnik* `Graph`. `AdjacencyLists` *podpira operacije*

- `addEdge(i, j)` *v konstantem času na operacijo;*
- `removeEdge(i, j)` *in* `hasEdge(i, j)` *v* $O(\deg(i))$ *času na operacijo;*
- `outEdges(i)` *v* $O(\deg(i))$ *času na operacijo; in*
- `inEdges(i)` *v* $O(n+m)$ *času na operacijo.*

`AdjacencyLists` *porabi* $O(n+m)$ *prostora.*

Obstaja veliko možnosti kako lahko implementiramo graf kot seznam sosednosti. Ena izmed vprašanj ki se nam porajajo so:

- Kakšno zbirko podatkov uporabiti za shranjevanje vsakega elementa v `adj`? Lahko bi uporabili polje, povezan seznam, ali celo zgoščevalne tabele.
- Lahko bi uporabili drug seznam sosednosti, `inadj`, ki hrani za vsak `i`, seznam vozlišč `j`, tako da $(j, i) \in E$ To lahko močno zmanjša trajanje operacije `inEdges(i)`, ampak rahlo poveča trajanje dodajanja in brisanja povezav.
- Lahko bi vpis za povezavo `(i, j)` v `adj[i]` bil povezan z referenco na ustrezni vpis v `inadj[j]`
- Lahko bi bile povezave prvorazredni objekti z njihovimi asociativnimi podatki Tako bi `adj` vseboval seznam povezav namesto seznama vozlišč (integers).

Pri večini zgornjih vprašanj pride do kompromisa med zahtevnostjo (in prostorom) izvedbe in zmogljivostjo operacij.

12.3 Graph Traversal

V tem poglavju predstavljamo dva algoritma za raziskovanje grafa, z začetkom v eni od njegovih točk, `i`, in zaključkom v vseh točkah ki so dosegljive iz `i`. Oba algoritma sta najbolj primerna za grafe predstavljene s seznamom sosednosti. Zato, ko bomo analizirali te algoritme, bomo predpostavili, da je osnova predstavitev s seznamom sosednosti `AdjacencyLists`.

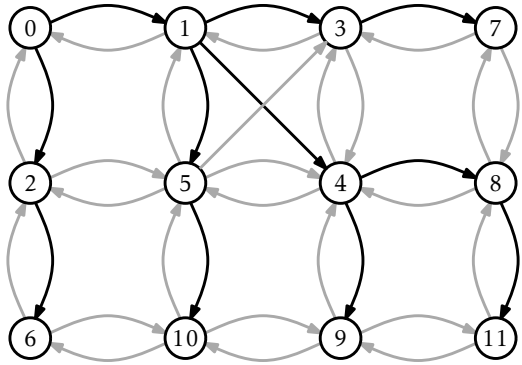
12.3.1 Iskanje v širino

Algoritem iskanje v širino ()*bread-first-search*) začne pri točki `i` in obišče najprej sosede od `i`, nato sosede od sosedov od `i`, nato sosede od sosedov od sosedov od `i` in tako naprej.

Algoritem je posplošitev algoritma za obhod v širino binarnih dreves (6.1.2), in je zelo podoben; uporablja vrsto, `q`, ki sprva vsebuje le `i`. Nato ponavljajoče izloča elemente iz `q` in dodaja svoje sosede v `q`, pod pogojem, da sosedje niso nikoli prej bili v `q`. Edina pomembna razlika med algoritmoma za iskanje v širino za grafe in za drevesa je ta, da algoritem za grafe mora zagotavljati, da ne doda iste točke v `q` več kot enkrat. To naredi s pomožnim boolean poljem, `seen`, ki beleži katere točke so že bile odkrite.

Algorithms

```
 bfs(Graph &g, int r) {
bool *seen = new bool[g.nVertices()];
SLList<int> q;
q.add(r);
seen[r] = true;
while (q.size() > 0) {
  int i = q.remove();
  ArrayStack<int> edges;
  g.outEdges(i, edges);
  for (int k = 0; k < edges.size(); k++) {
    int j = edges.get(k);
    if (!seen[j]) {
      q.add(j);
      seen[j] = true;
    }
  }
}
```



Slika 12.4: Primer iskanja v širino kjer začnemo pri vozlišču 0. Vozlišča so označena z redom po katerem so dodana v q. Povezave, ki izhajajo iz vozlišč dodanih v q, so obarvani v črno, ostale povezave pa v sivo.

```
delete[] seen;
```

Primer poganjanja bfs(g,0) na grafu iz 12.1 je prikazan v 12.4. V odvisnosti od seznama sosednosti so možna različna izvajanja; 12.4 uporablja seznam sosednosti v 12.3.

Analiziranje časa izvajanja algoritma bfs(g, i) je precej enostavno. Uporaba polja seen zagotavlja, da nobena točka ni dodana v q več kot enkrat. Dodajanje (in kasneje odstranjevanje) vsake točke iz q vzame konstanten čas na točko, skupno $O(n)$ časa. Ker je vsaka točka obdelana v notranji zanki največ enkrat je vsak seznam sosednosti obdelan največ enkrat, torej je vsaka povezava od G obdelana največ enkrat. Ta obdelava, ki je izvedena v notranji zanki, porabi konstanten čas na iteracijo, skupno $O(m)$ časa. Zato se celoten algoritem izvede v $O(n+m)$ času.

Naslednji izrek povzema učinkovitost algoritma bfs(g, r).

Izrek 12.3. *Ko je* Graf, g *podan kot vhod, ki je implementiran kot* Seznamsosedov, *potem algoritem* dfs(g, r) *potrebuje* $O(n+m)$ *časa.*

Breadth-first sprehod ima nekaj zelo posebnih lastnosti. Klicanje funkcije bfs(g, r) bo s časoma vrinilo (in s časoma izrinilo) vsako vozljišče j tako, da bo obstajala direktna pot od r do j. Še več, vozljišča na radalji 0

od r (r sam) bodo vstopila v q pred vozlišči na razdalji 1, ki bodo vstopila v q pred vozlišči na razdalji 2 in tako naprej. Torej metoda bfs(g,r) obišče vozlišča v narajaščujočem vrstnem redu razdalje od r in vozlišča, ki ne dosežemo od r niso nikoli obiskana.

Precej uporabna aplikacija breadth-first-search algoritma je torej, v iskanjue najkrajše poti. Da bi izračunali najkrajšo pot od r do vseh ostalih vozljišč, uporabimo različne verzije bfs(g,r) ki uporabljajo pomožni seznam, p, dolžine n. Ko je novo vozljišče j dodano v q, nastavimo p[j] = i. Na ta način, p[j] postane predzadnje vozlišče z najkrajšo razdaljo od r do j. S ponavljanjem tega postopka, če uzamemo p[p[j], p[p[p[j]]], in tako naprej, lahko ponovno zgradimo (v obratnem vrstnem redu) najkrajšo pot od r do j.

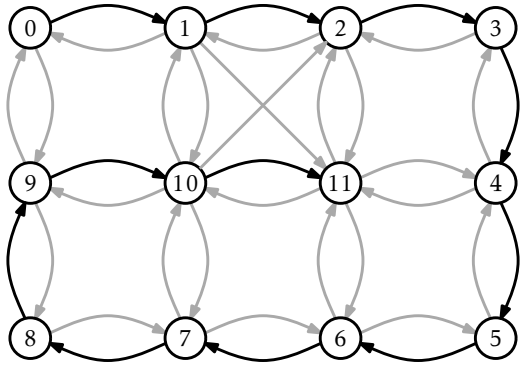
12.3.2 Iskanje v globino

The *depth-first-search* algoritem je podoben standardnemu algoritmu za sprehod binarnih dreves; najprej razišče celotno poddrevo, potem pa se vrne na trenutno vozlišče in nato razišče še drugo poddrevo. Še en način, kako si lahko predstavljamo depth-first-search algoritem je breadth-first search algoritem, z razliko, da depth-first-search uporablja sklad namesto vrste.

Med izvedbo depth-first-search algoritma, vsakemu vozlišču, i, določimo barvo, c[i]: bela če vozlišča še nismo srečali, siva če smo trenutno na tem vozlišču, in crna, če smo končali s tem vozliščem. Depth-first-search algoritem si najlažje predstavljamo kot rekurzivni algoritem. Začnemo tako, da obiščemo r. Ob obisku vozlišča i, ga najprej označimo z sivo barvo. Nato, pogledamo i-jev seznam sosedov in rekurzivno obiščemo vsa bela vozlišča, ki jih najdemo v tem seznamu. Na koncu, ko smo končali s procesiranjem i-ja, ga pobarvamo v crna in vrnemo.

Algorithms

```
 dfs(Graph &g, int i, char *c) {
c[i] = grey;  // currently visiting i
ArrayStack<int> edges;
g.outEdges(i, edges);
for (int k = 0; k < edges.size(); k++) {
  int j = edges.get(k);
  if (c[j] == white) {
```



Slika 12.5: Primer iskanja v globino (DFS) začnemo pri vozlišču 0. Vozlišča so označene po vrstnem redu v katerem so procesirane. Povezave, ki izhajajo iz rekurzivnega klica so obarvane v črno, ostale pa v sivo.

```
    c[j] = grey;
    dfs(g, j, c);
  }
}
c[i] = black; // done visiting i

 dfs(Graph &g, int r) {
char *c = new char[g.nVertices()];
dfs(g, r, c);
delete[] c;
```

Primer izvedbe tega algoritma je prikazan na 12.5.

Čeprav je iskanje v globino najboljše izvedeno skozi rekurzivni algoritem, rekurzija ni najboljša implementacija. Dejanska koda navedena zgoraj ne bo uspela pri večjih grafih zaradi prekoračitve. Alternativna implementacija je zamenjava rekurzivnega sklada z eksplicitnim skladom, s. Naslednja implementacija naredi točno to:

Algorithms

```
 dfs2(Graph &g, int r) {
char *c = new char[g.nVertices()];
SLList<int> s;
s.push(r);
```

```
while (s.size() > 0) {
  int i = s.pop();
  if (c[i] == white) {
    c[i] = grey;
    ArrayStack<int> edges;
    g.outEdges(i, edges);
    for (int k = 0; k < edges.size(); k++)
      s.push(edges.get(k));
  }
}
delete[] c;
```

V zgornji kodi, ko je naslednje vozlišče `i` procesirano, se `i` obarva v `sivo` in zamenja v skladu, z njegovimi sosednjimi vozlišči. V naslednji iteraciji bo eno izmed teh vozlišč obiskano.

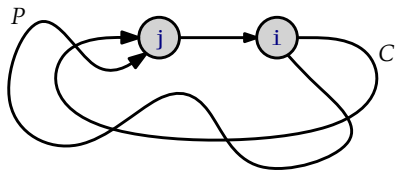
Kot pričakovano ,sta časovni zahtevnosti za `dfs(g,r)` in `dfs2(g,r)` enaki kot tista od `bfs(g,r)`:

Izrek 12.4. *Ko je* `Graf,g` *podan kot vhod, ki je implementiran kot podatkovna struktura* `Seznamsosedov`*, potem oba algoritma* `dfs(g,r)` *in* `dfs2(g,r)` *potrebujeta* $O(n+m)$ *časa.*

Kot pri iskanje-v-širino algoritmu, imamo temeljno drevo, ki je povezano z vsako izvedbo iskanje-v-globino. Ko se vozlišče $i \neq r$ obarva iz `bele` na `sivo` , to se zgodi ,ker je bil `dfs(g,i,c)` rekurzivno klican med procesiranjem nekega vozlišča i'. (V primeru `dfs2(g,r)` algoritma, je i eden od vozlišč ki zamenja i' v skladu.) Če gledamo na i' kot starša od i, potem ohranimo drevo s korenom pri r. V 12.5, je to drevo pot od vozlišča 0 do vozlišča 11.

Pomembna lastnost iskanje-v-globino algoritma je sledeča: Predpostavimo da ko je vozlišče i obarvano `sivo`, obstaja pot od i do nekega drugega vozlišča j ki uporablja le bela vozlišča. Potem bo j prvo obarvan v `sivo` nato pa v `crno` , preden bo i obarvan v `crno`. (To je lahko dokazano s protislovjem, tako da upoštevamo katerokoli pot P od i do j.)

Ena vloga te lasnosti je prepoznavanje ciklov. Glejte 12.6. Preučimo nek cikel C, ki je dosegljiv iz r. Naj bo i prvo vozlišče od C , ki bo obarvano `sivo` in naj bo vozlišče j predhodnik od i v ciklu C. Potem, preko



Slika 12.6: Algoritem iskanje-v-globino lahko uporabimo za odkrivanje ciklov v G. Vozlišče j je obarvano sivo dokler je i obarvan sivo. To pomeni , da obstaja pot P od i do j v iskanje-v-globinu drevesu. Povezava (j,i) pomeni da je P tudi cikel.

zgoraj omenjene lastnosti bo j obarvan sivo in algoritem bo obravnaval povezavo (j,i) medtem, ko je vozlišče i še vedno sivo. Zato lahko algoritem sklepa, da obstaja pot P od i do j pri iskanju-v-globino, zato obstaja tudi povezava (j,i), kar pomeni da je P prav tako cikel.

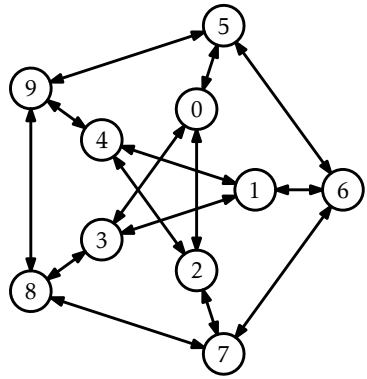
12.4 Diskusija in vaje

Časovna zahtevnost iskanje-v-globino in iskanje-v-širino algoritmov so nekoliko precenjene preko izreka 12.3 in 12.4. Definiraj n_r kot število vozlišč , i od G , za katerega obstaja pot od r do i. Definiraj m_r kot število povezav, ki imajo ta vozlišča za svoj izvor. Potem bo v sledečem izreku časovna zahtevnost iskanje-v-globino in iskanje-v-širino algoritmov , bolj točno navedena. (Ta bolj točen izrek je uporaben v nekaterih vlogah algoritmov , podčranih v vajah.)

Izrek 12.5. *Ko je* Graf,g *podan kot vhod, ki je implementiran kot podatkovna struktura* Seznamsosedov *, potem se vsak algoritem* bfs(g,r),dfs(g,r) *in* dfs2(g,r) *izvaja* $O(n_r + m_r)$ *časa.*

Izgleda, da sta Iskanje-v-širino neodvisno odkrila Moore [?] in Lee [?] v kontekstu raziskovanja labirintov in preusmerjanja tokokroga.

Predstavitev grafov kot seznam sosedov sta demonstrirala Hopcroft in Tarjan [?] kot alternativo (bolj pogostim) predstavitvam z matrikami sosednosti. Ta predstavitev, kot tudi iskanje-v-globino igrata veliko vlogo v znameniti Hopcroft-Tarjan ravninskem testnem algoritmu ki lahko določi v $O(n)$ času, če se lahko graf nariše v ravnini in v takem načinu, da noben



Slika 12.7: Primer grafa.

par vozlišč ne preseka drug drugega. [**?**].

V naslednji vajah, imamo neusmerjen graf v enem ki za vsaki `i` in `j`, povezavo (`i`, `j`) je predstavljen , če in samo če je povezava (`j`, `i`) prisotna.

Naloga 12.1. Narišite seznam sosednosti ter matriko sosednosti za graf na sliki 12.7.

Naloga 12.2. *Matrika neodvisnosti* za graf, G, je $\mathtt{n} \times \mathtt{m}$ matrika, A, kjer velja

$$A_{i,j} = \begin{cases} -1 & \text{če je točka } i \text{ vir množice } j \\ +1 & \text{če je točka } i \text{ tarča množice } j \\ 0 & \text{sicer.} \end{cases}$$

1. Narišite incidenčno matriko za graf na sliki 12.7.
2. Načrtajte, analizirajte ter implementirajte incidenčno matriko za dan graf. Analizirajte porabo prostora ter ceno za `addEdge(i, j)`, `removeEdge(i, j)`, `hasEdge(i, j)`, `inEdges(i)` in `outEdges(i)`.

Naloga 12.3. Prikažite izvedbo algoritmov `bfs(G,0)` in `dfs(G,0)` na grafu, `G`, na sliki 12.7.

Naloga 12.4. Naj bo G neusmerjen graf. G je *povezan* graf, če za vsak par vozlišč `i` in `j` v G velja, da obstaja pot iz vozlišča `i` v vozlišče `j` (dokler je G neusmerjen, obstaja tudi pot iz `j` v `i`). Dokažite, da pri povezanem grafu G velja časovna zahtevnost $O(\mathtt{n}+\mathtt{m})$.

Naloga 12.5. Naj bo G neusmerjen graf. *Označevanje povezanih komponent* v grafu G razdeli vozlišča od G v maksimalne množice, kjer vsaka ustvari povezan podgraf. Pokaži kakoizračunati označevanje povezanih komponent grafa G v času $O(n+m)$.

Naloga 12.6. Naj bo graf G neusmerjen graf. *Vpeto drevo* grafa G je skupek dreves, kjer povezave ter vozlišča posameznih dreves, pripadajo grafu G. Izračunajte vpeto drevo grafa G pri časovni zahtevnosti $O(n+m)$.

Naloga 12.7. Rekli smo, da je graf G *krepko povezani*, če za vsak par vozlišč `i` in `j` v G, obstaja pot iz vozlišča `i` v vozlišče `j`. Dokažite, da je pri krepko povezanem grafu G časovna zahtevnost $O(n+m)$.

Naloga 12.8. Podan je graf $G = (V, E)$ ter nekaj točk, kjer je $r \in V$, Izračunajte dolžino najkrajše poti iz točke `r` v `i` za vsako točko, kjer je $i \in V$.

Naloga 12.9. Podajte primer, kjer metoda `dfs(g, r)` obišče vozlišča grafa v nasprotnem vrstnem redu, kot metoda `dfs2(g, r)`. Napišite novo verzijo metode `dfs2(g, r)`, ki obišče vozlišča danega grafa v enakem vrstnem redu kot metoda `dfs(g, r)`. (Namig: Sledite izvršitvi vsakega algoritma na grafu kjer je `r` vir več kot 1 množice.)

Naloga 12.10. A *universal sink* v grafu G je točka, ki je tarča $n-1$ povezav in ni vir nobene množice.[1] Oblikujte in implementirajte algoritem, ki preveri, če ima graf G, predstavljen kot `AdjacencyMatrix`, universal sink. Časovna zahtevnost vašega algoritma bi morala biti $O(n)$.

[1] universal sink, `v`, včasih imenujemo tudi *celebrity*: Vsi zbrani v nekem prostoru prepoznajo `v`, toda `v` ne prepozna nobenega v tem prostoru.

Poglavje 13

Podatkovne strukture za cela števila

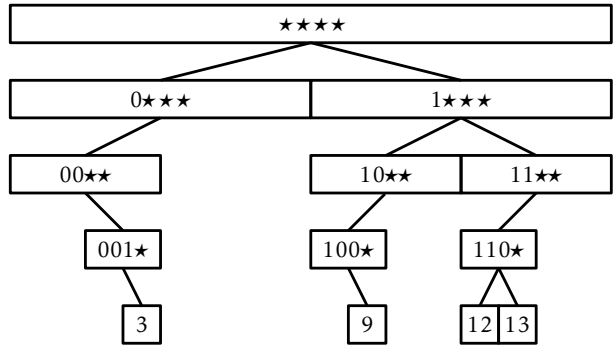
V tem poglavju se bomo vrnili k problemu implementiranja SSet-a. Razlika v implementaciji je ta, da zdaj privzamemo, da so elementi shranjeni v SSet-u, w-bitna cela števila. To pomeni da hočemo implementirati metode add(x), remove(x) in find(x), kjer velja da $x \in \{0,\ldots,2^w-1\}$. Če malo pomislimo obstaja veliko aplikacij, kjer imamo podatke, oziroma vsaj ključe za sortiranje podatkov, ki so cela števila.

Govorili bomo o treh podatkovnih strukturah, vsaka izmed njih bo temeljila na idejah že prej omenjenih podatkovnih strukturah. Prva struktura, BinaryTrie, lahko izvrši vse tri SSet operacije v času $O(w)$. To sicer ni tako zelo impresivno, saj ima vsaka podmnožica $\{0,\ldots,2^w-1\}$ velikost $n \leq 2^w$, tako da je $\log n \leq w$. Vse ostale SSet implementacije, s katerimi imamo opravka v tej knjigi lahko izvedejo vse operacije v $O(\log n)$ času, torej so vse vsaj toliko hitre kot BinaryTrie.

Druga struktura, XFastTrie, pohitri iskanje v BinaryTrie z uporabo razpršenja. S to pohitritvijo se find(x) operacija izvede v $O(\log w)$ času, vendar pa add(x) in remove(x) operaciji v XFastTrie še vedno potrebujeta $O(w)$ časa. Prostor, ki ga XFastTrie potrebuje pa je $O(n \cdot w)$.

Tretja podatkovna struktura, YFastTrie, uporablja XFastTrie za shranjevanje le vzorca enega oz. okoli enega, od vsakih w elementov in preostale elemente shranjuje v standardno SSet strukturo. Ta trik zmanjša čas izvajanja operacij add(x) in remove(x) na $O(\log w)$ in zmanjša prostorsko zahtevnost na $O(n)$.

Implementacije uporabljene kot primeri v tem poglavju lahko shranjujejo katerikoli tip podatkov, dokler je lahko ta podatek nekako pred-



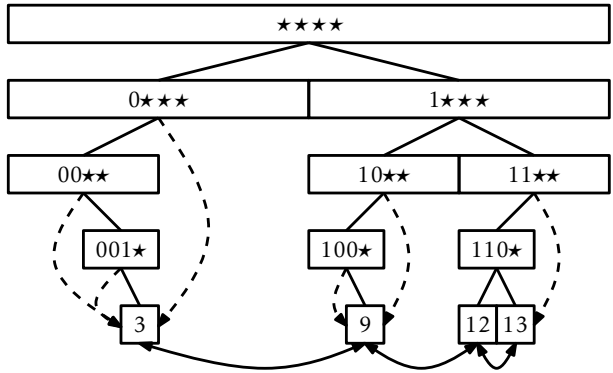
Slika 13.1: Cela števila shranjena v binary trie so zakodirana kot poti od korena do lista.

stavljen tudi kot celo število. V primerih programske kode, predstavlja spremenljivka `ix` vedno, vrednost celega števila, ki pripada `x`. Metoda `intValue(x)` pa pretvori `x` v njegovo pripadajoče celo število. V besedilu bomo enostavno uporabljali `x` kot celo število.

13.1 `BinaryTrie`: digitalno iskalno drevo

`BinaryTrie` zakodira niz `w`-bitnih celih števil v binarno drevo. Vsi listi v drevesu imajo globino `w` in vsako celo število je prikazano kot pot od korena do lista. Pot za celo število `x` na nivoju `i` nadaljuje pot proti levemu poddrevesu, če je `i`-ti najpomembnejši bit (most significant bit) `x` enak 0 oz. nadaljuje pot proti desnemu poddrevesu, če je ta bit enak 1. 13.1 prikazuje primer, ko je $w = 4$, in trie shranjuje cela števila 3(0011), 9(1001), 12(1100), in 13(1101).

Ker iskalna pot za vrednost `x` odvisi od bitov `x`-a, nam bo koristilo, če otroka vozlišča poimenujemo `u`, `u.child`[0] (`left`) in `u.child`[1] (`right`). Tile kazalci na otroke bodo pravzaprav služili dvema namenoma. Ker listi v binary trie nimajo nobenega otroka, so kazalci uporabljeni za povezavo listov v dvojno povezan seznam. Za list v binary trie je `u.child`[0] (`prev`) je vozlišče, ki je pred `u`-jem v seznamu in `u.child`[1] (`next`) je vozlišče, ki



Slika 13.2: `BinaryTrie` z `jump` kazalci, prikazanami kot prekinjene ukrivljene povezave.

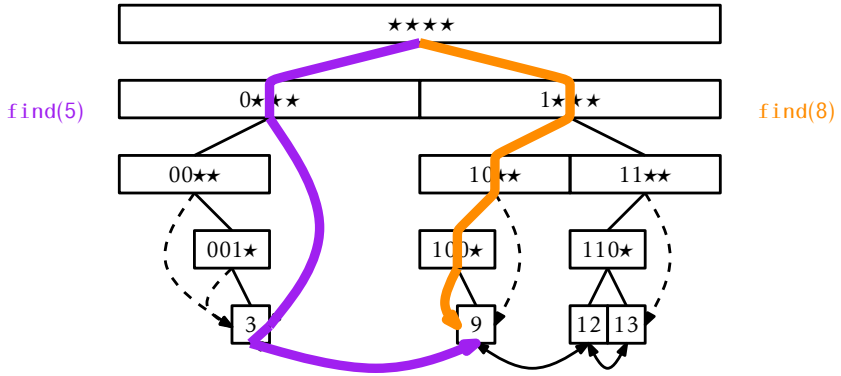
sledi u-ju v seznamu. Posebno vozlišče `dummy`, je uporabljeno pred prvim vozliščem in za zadnjim vozliščem v seznamu. (glej 3.2). V primerih kode se `u.child[0]`, `u.left`, in `u.prev` nanašajo na enako polje v vozlišču `u`, kot `u.child[1]`, `u.right`, i `u.next`.

Vsako vozlišče, `u`, vsebuje tudi dodatni kazalec `u.jump`. Če je `u` brez svojega levega otroka, potem `u.jump` kaže na najmanjši list v `u`-jevem poddrevesu. Če pa je `u` brez svojega desnega otroka potem `u.jump` kaže na največji list v `u`-jevem poddrevesu. Primer `BinaryTrie`, ki prikazuje `jump` kazalce in dvojno povezan seznam na nivoju listov, je prikazan na 13.2.

`find(x)` operacija je v `BinaryTrie` precej enostavna. Najprej sledimo iskalni poti za `x` v trie. Če dosežemo list, potem smo našli `x`. Če pa naletimo na vozlišče iz katerega potem ne moremo napredovati (ker `u`-ju manjka otrok), potem sledimo `u.jump` kazalcu, ki nam kaže ali na najmanjši list, ki je še večji od `x` ali na največji list, ki je še manjši od `x`. Kateri od teh dveh primerov se zgodi odvisi od tega ali `u`-ju manjka njegov levi ali desni otrok. V prvem primeru (`u`-ju manjka njegov levi otrok), smo že prišli do vozlišča do katerega hočemo. V kasnejšem primeru (`u`-ju manjka njegov desni otrok), pa lahko uporabimo povezan seznam, da pridemo do vozlišča do katerega hočemo. Vsak od teh primerov je prikazan na 13.3.

`BinaryTrie`

```
T find(T x) {
  int i, c = 0;
```



Slika 13.3: Poti po katerih gre `find(5)` in `find(8)`.

```
  unsigned ix = intValue(x);
  Node *u = &r;
  for (i = 0; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
    if (u->child[c] == NULL) break;
    u = u->child[c];
  }
  if (i == w) return u->x;  // found it
  u = (c == 0) ? u->jump : u->jump->next;
  return u == &dummy ? null : u->x;
}
```

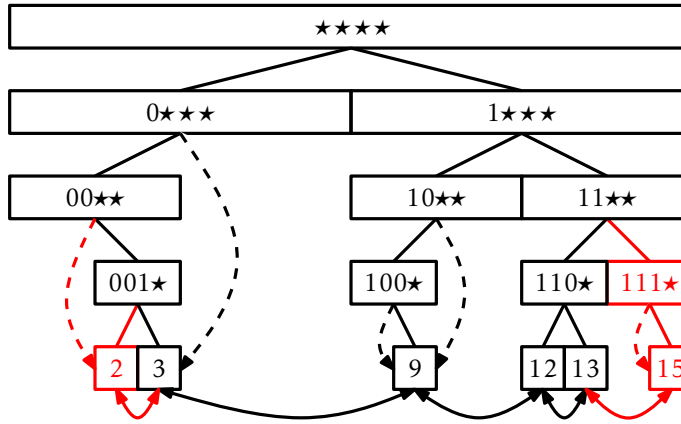
Čas izvajanja metode `find(x)` je določena z časom, ki ga struktura potrebuje, da pride po poti iz korena do lista. Torej je časovna kompleksnost $O(w)$.

Tudi `add(x)` operacija je v `BinaryTrie` precej enostavna, vendar ima še vedno veliko za narediti:

1. Sledi iskalni poti za `x` dokler ne doseže vozlišča `u`, kjer ne more več nadeljevati.

2. Ustvari ostanek iskalne poti od `u` do lista, ki vsebuje `x`.

3. Vozlišče `u'`, ki vsebuje `x`, se doda povezanemu seznamu listov (metoda ima dostop do prednika, `pred`, `u'`-ja v povezanem seznamu



Slika 13.4: Dodajanje vrednosti 2 in 15 v `BinaryTrie` na 13.2.

jump kazalca zadnjega vozlišča `u`, na katerega smo naleteli v koraku 1.)

4. Sledi nazaj po iskalni poti za `x` in sproti popravlja `jump` kazalce na vozliščih, kjer bi zdaj moral `jump` kazalec kazati na `x`.

Dodajanje v strukturo je prikazano na 13.4.

BinaryTrie

```
bool add(T x) {
  int i, c = 0;
  unsigned ix = intValue(x);
  Node *u = &r;
  // 1 - search for ix until falling out of the trie
  for (i = 0; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
    if (u->child[c] == NULL) break;
    u = u->child[c];
  }
  if (i == w) return false; // already contains x - abort
  Node *pred = (c == right) ? u->jump : u->jump->left;
  u->jump = NULL;  // u will have two children shortly
  // 2 - add path to ix
  for (; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
```

```
    u->child[c] = new Node();
    u->child[c]->parent = u;
    u = u->child[c];
  }
  u->x = x;
  // 3 - add u to linked list
  u->prev = pred;
  u->next = pred->next;;
  u->prev->next = u;
  u->next->prev = u;
  // 4 - walk back up, updating jump pointers
  Node *v = u->parent;
  while (v != NULL) {
    if ((v->left == NULL
        && (v->jump == NULL || intValue(v->jump->x) > ix))
    || (v->right == NULL
        && (v->jump == NULL || intValue(v->jump->x) < ix)))
      v->jump = u;
    v = v->parent;
  }
  n++;
  return true;
}
```

Ta metoda naredi en sprehod navzdol po iskalni poti `x`-a in en sprehod nazaj navzgor. Vsak korak od teh sprehodov potrebuje konstantno časa, torej je časovna zahtevnost `add(x)` enaka $O(w)$.

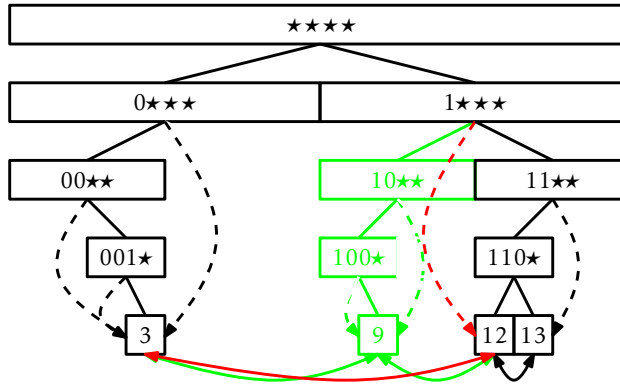
`remove(x)` operacija razveljavi, kar naredi `add(x)` operacija. Prav tako kot `add(x)`, ima tudi `remove(x)` veliko za postoriti:

1. Najprej sledi iskalni poti za `x` dokler ne doseže lista `u`, ki vsebuje `x`.

2. Izbriše `u` iz dvojno povezanega seznama.

3. Izbriše `u` in se sprehodi nazaj navzgor po iskalni poti za `x` ter sproti briše vozlišča dokler ne doseže vozlišča `v`, ki ima otroka, ki ni del iskalne poti za `x`.

4. Sprehodi se še navzgor od `v`-ja do korena in spreminja `jump` kazalce, ki kažejo na `u`.



Slika 13.5: Odstranjevanje vrednosti 9 iz `BinaryTrie` na 13.2.

Odstranjevanje je prikazano na 13.5.

BinaryTrie

```
bool remove(T x) {
  // 1 - find leaf, u, containing x
  int i = 0, c;
  unsigned ix = intValue(x);
  Node *u = &r;
  for (i = 0; i < w; i++) {
    c = (ix >> (w-i-1)) & 1;
    if (u->child[c] == NULL) return false;
    u = u->child[c];
  }
  // 2 - remove u from linked list
  u->prev->next = u->next;
  u->next->prev = u->prev;
  Node *v = u;
  // 3 - delete nodes on path to u
  for (i = w-1; i >= 0; i--) {
    c = (ix >> (w-i-1)) & 1;
    v = v->parent;
    delete v->child[c];
    v->child[c] = NULL;
    if (v->child[1-c] != NULL) break;
  }
  // 4 - update jump pointers
  v->jump = u;
```

```
  for (; i >= 0; i--) {
    c = (ix >> (w-i-1)) & 1;
    if (v->jump == u)
      v->jump = u->child[1-c];
    v = v->parent;
  }
  n--;
  return true;
}
```

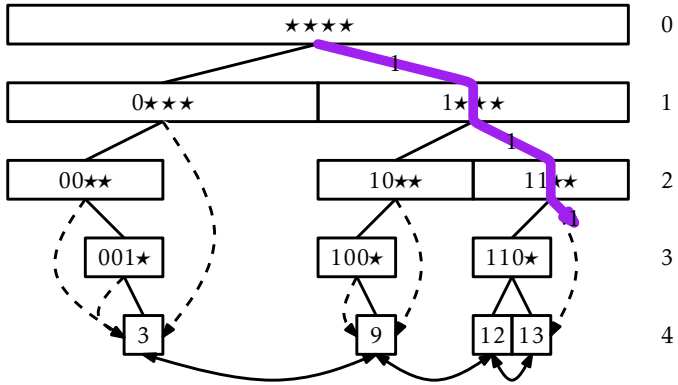
Izrek 13.1. *`BinaryTrie` implementira `SSet` vmesnik za `w`-bitna cela števila. `BinaryTrie` podpira operacije* `add(x)`, `remove(x)` *in* `find(x)` *v časovni kompleksnosti $O(w)$ na operacijo. Prostor, ki ga `BinaryTrie` uporablja za shranjevanje `n` vrednosti je $O(n \cdot w)$.*

13.2 `XFastTrie`: Iskanje v dvojnem logaritmičnem času

Hitrost izvajanja `BinaryTrie` strukture ni ravno impresivna. Število elementov `n` shranjenih v podatkovi strukturi je najmanj 2^w torej $\log n \leq w$. Z drugimi besedami, vse primerjalne `SSet` strukture opisane v drugih poglavnih te knjige so vsaj tako učinkovite kot `BinaryTrie` in niso omejene samo na shranjevanje celih števil.

V slednjem besedilu je opisana `XFastTrie`, ki je v osnovi `BinaryTrie` z $w+1$ razpršilnimi tabelami—ena za vsak nivo trie. Te razpršilne tabele se uporabljajo za pohitritev `find(x)` operacije na $O(\log w)$ čas. `find(x)` operacija v `BinaryTrie` je skoraj končana, ko dosežemo vozlišče `u` kjer gre iskalna pot proti `x` `u.right` (oziroma `u.left`), ampak `u` nima desnega (oziroma levega) otroka. Na tej točki iskanje uporablja `u.jump` za skok do lista `v`, ki se nahaja v `BinaryTrie` in vrne ali `v` ali pa svojega naslednika v povezanem seznamu listov. `XFastTrie` pohitri proces iskanja z uporabo binarnega iskanja na nivojih trie za lociranje vozlišča `u`.

Za uporabo binarnega iskanja moramo izvedeti ali je vozlišče `u`, ki ga iščemo, nad določenim nivojem `i` ali pod nivojem `i`. Ta informacija je podana prvimi `i` biti binarnega zapisa `x`; ti biti določajo iskalno pot, ki jo naredi `x` od korena do nivoja `i`. Na primer sklicujoč na 13.6; na sliki je zadnje vozlišče `u` na iskalni poti za število 14 (katerga binarni zapis je



Slika 13.6: Ker na sliki ni vozlišča označenega z 111★ se iskalna pot za 14 (1110) konča pri vozlišču 11★★ .

1110) označeno z 11★★ na nivoju 2, ker na nivoju tri ni nobenega vozlišča označenega z 111★. Tako lahko označimo vsako vozlišče na nivoju `i` z `i`-bitnim celim številom. Tako bi bilo vozlišče `u`, ki ga iščemo, na nivoju ali nižje od nivoja `i`, če in samo če obstaja vozlišče na nivoju `i` čigar oznaka se sovpada z prvimi `i` biti binarnega zapisa `x`.

Pri `XFastTrie` za vsak $i \in \{0,\ldots,w\}$ shranjujemo vsa vozlišča na nivoju `i` v `USet t[i]`, ki je implementiran kot razpršilna tabela (5). Uporaba `USet` nam omogoča preverjanje v konstantnem času, če obstaja vozlišče na nivoju `i`, ki se sovpada s prvimi `i` biti `x`. V bistvu lahko to vozlišče najdemo z uporabo `t[i].find(x >> (w - i))`

Razpršilne tabele $t[0],\ldots,t[w]$ nam omogočajo binarno iskanje za iskanje `u`. Vemo, da se `u` nahaja na nekem nivoju `i` z $0 \le i < w+1$. Tako torej inicializiramo $l = 0$ in $h = w+1$ in ponavljajoče gledamo v razpršilno tabelo $t[i]$ kjer $i = \lfloor (l+h)/2 \rfloor$. Če $t[i]$ vsebuje vozlišče katerega oznaka se sovpada z `i` prvimi biti `x` določimo $l = i$ (`u` je na nivoju ali nižje od nivoja `i`); v nasprotnem primeru določimo $h = i$ (`u` je nižje od nivoja `i`). Ta proces se konča ko $h - l \le 1$, ko lahko sklepamo, da je `u` na nivoju `l`. Potem zaključimo `find(x)` operacijo z uporabo `u.jump` in dvojno povezanega seznama listov.

XFastTrie

```
T find(T x) {
  int l = 0, h = w+1;
```

```
  unsigned ix = intValue(x);
  Node *v, *u = &r;
  while (h-l > 1) {
    int i = (l+h)/2;
    XPair<Node> p(ix >> (w-i));
    if ((v = t[i].find(p).u) == NULL) {
      h = i;
    } else {
      u = v;
      l = i;
    }
  }
  if (l == w) return u->x;
  Node *pred = (((ix >> (w-l-1)) & 1) == 1)
               ? u->jump : u->jump->prev;
  return (pred->next == &dummy) ? nullt : pred->next->x;
}
```

Vsaka iteracija `while` zanke v zgornji metodi zmanjša $h-l$ za približno faktor ali dva, tako da ta zanka najde `u` po $O(\log w)$ iteracijah. Vsaka iteracija opravi konstantno količino dela in eno `find(x)` operacijo v `USet`, ki porabi konstanten čas. Preostanek dela zavzame samo konstanten čas. Tako `find(x)` methoda v `XFastTrie` potrebuje samo $O(\log w)$ časa.

Metodi `add(x)` in `remove(x)` za `XFastTrie` sta skoraj identični enakim metodam v `BinaryTrie`. Edina razlika je upravljanje z razpršilnimi tabelami `t[0]`,...,`t[w]`. Ob izvajanju operacije `add(x)`, ko je ustvarjeno novo vozlišče na nivoju `i`, je potem to vozlišče dodano v `t[i]`. Ob izvajanju `remove(x)` operacije, ko je vozlišče odstranjeno z nivoja `i`, je potem to vozlišče odstranjeno iz `t[i]`. Ker vstavljanje in brisanje iz razpršilne tabele traja konstanten čas, to ne poveča časa izvajanja `add(x)` in `remove(x)` za več kot konstanten faktor. Koda za `add(x)` in `remove(x)` je izpuščena, ker je skoraj identična (dolgi) kodi, ki se nahaja v implementaciji operacij za `BinaryTrie`.

Sledeči teorem povzame delovanje `XFastTrie`:

Izrek 13.2. *`XFastTrie` implementira `SSet` vmesnik za `w`-bitna cela števila. `XFastTrie` podpira operacije*

- `add(x)` *in* `remove(x)` *v času* $O(w)$ *na operacijo in*

- `find(x)` *v času* $O(\log w)$ *na operacijo*

Prostorska zahtevnost `XFastTrie`*, ki shrani* n *vrednosti je* $O(n \cdot w)$.

13.3 YFastTrie: Dvokratni-Logaritmični Čas SSet

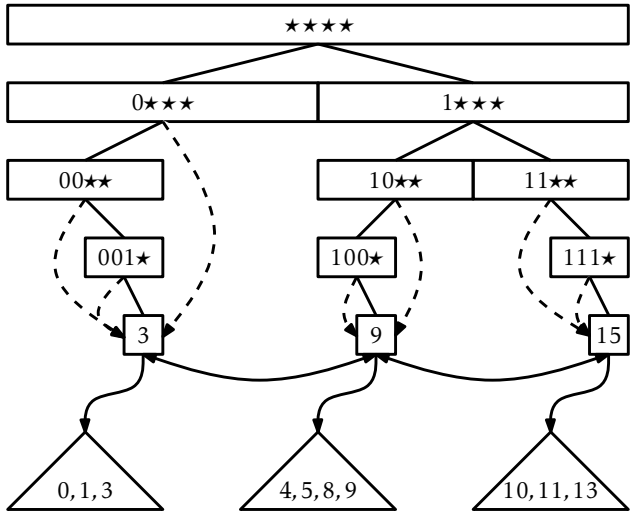
`XFastTrie` je velika, celo eksponentna, izboljšava `BinaryTrie` v kategoriji poizvedbenega časa, vendar operaciji `add(x)` in `remove(x)` še nista veliko hitrejši. Poleg tega je poraba prostora $O(n \cdot w)$ večja kot pri drugih SSet implementacijah, predstavljenih v tej knjigi, ki uporabljajo $O(n)$ prostora. Možno je, da sta ta dva problema med sabo povezana; če n `add(x)` operacij gradi strukturo velikosti $n \cdot w$, potem operacija `add(x)` potrebuje vsaj w časa (in prostora) na operacijo.

`YFastTrie`, o katerem bomo govorili naprej, izboljša hkrati porabo prostora in hitrosti `XFastTrie`. `YFastTrie` uporablja `XFastTrie`, `xft`, a le shranjuje $O(n/w)$ vrednosti v `xft`. Na ta način `xft` v celoti uporabi samo $O(n)$ prostora. Poleg tega je samo ena od vseh w operacij `add(x)` ali `remove(x)` v `YFastTrie` enaka operaciji `add(x)` ali `remove(x)` v `xft`. Na tak način je povprečna zahtevnost nastalih klicev na `xft` operacije `add(x)` in `remove(x)` konstantna.

Tako se lahko vprašamo: če `xft` shranjuje samo n/w elementov, kam gre preostalih $n(1 - 1/w)$ elementov? Ti elementi se shranijo v *pomožnih strukturah*, v tem primeru je to podaljšana verzija treaps (7.2). Obstaja približno n/w takšnih pomožnih struktur – tako v povprečju vsaka shranjuje $O(w)$ primerov. Treaps so podprte z operacijami v logaritmičnem času SSet, tako pa bodo operacije treaps delale s časom $O(\log w)$, kot je to potrebno.

Če govorimo bolj konkretno, `YFastTrie` vsebuje `XFastTrie`, `xft`, ki vsebuje naključne primere podatkov, kjer se vsak element pojavi v primerih neodvisno z verjetnostjo $1/w$. Zaradi prikladnosti je vrednost $2^w - 1$ vedno vsebovana v `xft`. Naj $x_0 < x_1 < \cdots < x_{k-1}$ označuje elemente, ki so vsebovani v `xft`. Povezan z vsakem elementom x_i je treap t_i, ki shranjuje vse vrednosti v dosegu $x_{i-1} + 1, \ldots, x_i$. To je ilustrirano na 13.7.

`find(x)` operacija v `YFastTrieX` je dokaj enostavna. V `xft` iščemo x in najdemo nekaj vrednosti x_i povezanih z treap t_i. Potem uporabimo



Slika 13.7: A `YFastTrie` containing the values 0, 1, 3, 4, 6, 8, 9, 10, 11, and 13.

metodo treap `find(x)` na t_i za odgovor na poizvedbo. Ta metoda se lahko v celoti zapiše v eni vrstici:

YFastTrie
```
T find(T x) {
  return xft.find(YPair<T>(intValue(x))).t->find(x);
}
```

Prva `find(x)` operacija (na `xft`) vzame $O(\log w)$ časa. Druga `find(x)` operacija (nad treap) vzame $O(\log r)$ časa, kjer je r velikost treap. Kasneje v tem razdelku, bomo pokazali,da je pričakovana velikost treap $O(w)$ torej ta operacija vzame $O(\log w)$ časa.[1]

Dodajanje elementa v `YFastTrie` je tudi dokaj preprosto—večino časa. `Add(x)` metoda pokliče `xft.find(x)` ta alocira treap, `t`, v katerega bo `x` lahko vstavljen. Ta potem pokliče `t.add(x)` za dodajanje `x` k `t`. Pri tej točki, meče nepristranski kovanec katerih glave pridejo z verjetnostjo $1/w$ in tudi repi z verjetnostjo $1 - 1/w$. Če na kovancu dobimo glave, potem bo `x` dodan k `xft`.

[1] To je aplikacija *Jensenove neenakosti*: If $\mathrm{E}[r] = w$, then $\mathrm{E}[\log r] \leq \log w$.

Tukaj stvari postanejo malce bolj zapletene. Ko je x dodan k xft, mora biti treap t razdeljeno na dva treaps, t1 in t′. Treaps t1 vsebuje vse vrednosti manjše ali enake od x; t′ je prvotno treap, t, z vsemi odstranjenimi elementi t1. Ko je to narejeno, dodamo par (x,t1) k xft. 13.8 prikazuje primer.

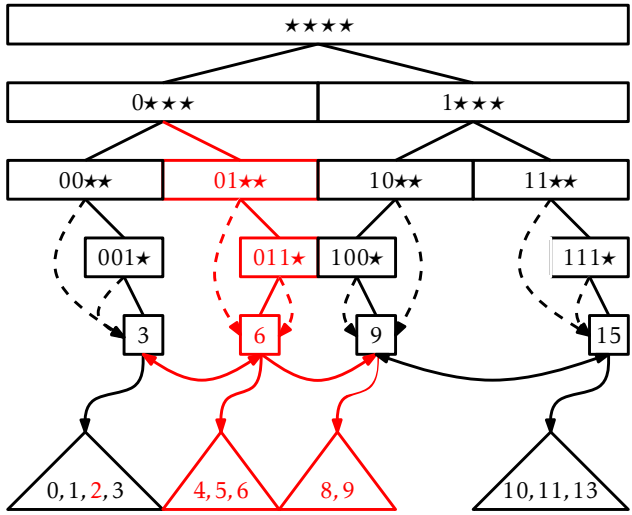
YFastTrie
```
bool add(T x) {
  unsigned ix = intValue(x);
  Treap1<T> *t = xft.find(YPair<T>(ix)).t;
  if (t->add(x)) {
    n++;
    if (rand() % w == 0) {
      Treap1<T> *t1 = (Treap1<T>*)t->split(x);
      xft.add(YPair<T>(ix, t1));
    }
    return true;
  }
  return false;
  return true;
}
```

Dodajanje x k t vzame $O(\log w)$ časa. **??** prikazuje, da je razdelitev t v t1 in t′ lahko narejena v $O(\log w)$ pričakovanem času. Dodajanje para (x,t1) k xft vzame $O(w)$ časa, ampak se zgodi samo z verjetnostjo $1/w$. Zato je, pričakovan čas poteka add(x) operacije

$$O(\log w) + \frac{1}{w} O(w) = O(\log w) \ .$$

Remove(x) metoda razveljavi delo,ki se izvede z add(x). xft uporabimo, da najdemo list u, in xft, ki vsebuje odgovor za xft.find(x). Iz u, dobimo treap, t, ki vsebuje x in ta x odstrani iz t. Če je bil x shranjen v xft (in x ni enak 2^w-1) potem odstranimo x iz xft in dodamo elemente iz x-tega treap v treap, t2, ki je shranjen v u-tem nasledniku v povezanem seznamu. To je prikazano v 13.9.

YFastTrie
```
bool remove(T x) {
  unsigned ix = intValue(x);
  XFastTrieNode1<YPair<T> > *u = xft.findNode(ix);
  bool ret = u->x.t->remove(x);
```

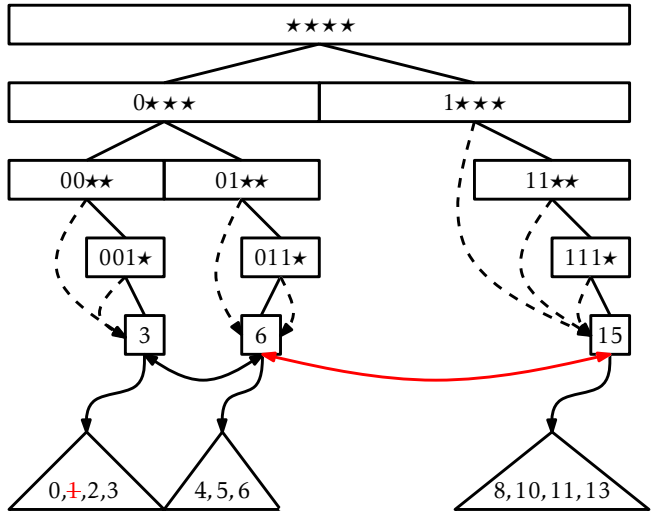


Slika 13.8: Dodajanje vrednosti 2 in 6 v `YFastTrie`. Pri metu kovanca za 6 pridejo glave, torej je bila 6 dodana k `xft` in treap, ki je vsebovalo 4, 5, 6, 8, 9 je bilo razdeljeno.

```
  if (ret) n--;
  if (u->x.ix == ix && ix != UINT_MAX) {
    Treap1<T> *t2 = u->child[1]->x.t;
    t2->absorb(*u->x.t);
    xft.remove(u->x);
  }
  return ret;
}
```

Iskanje člena `u` in `xft` vzame $O(\log w)$ pričakovanega časa. Odstranjevanje `x` iz `t` vzame $O(\log w)$ pričakovanega časa. Spet, **??** prikazuje, da je združevanje vseh elementov `t` v `t2` lahko storjena v $O(\log w)$ času. Če je potrebno, odstranjevanje `x` iz `xft` vzame $O(w)$ časa, toda `x` je vsebovan v `xft` z verjetnostjo $1/w$. Zato je pričakovan čas odstranjevanja elementa iz `YFastTrie` enak $O(\log w)$.

Prej v razpravi smo prestavili debato o velikosti poddreves znotraj te strukture. Pred zaključkom poglavja smo dokazali potreben rezultat.



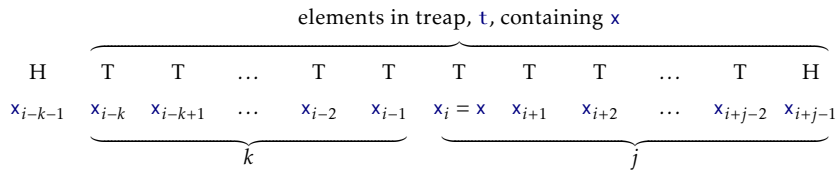
Slika 13.9: Odstranjevanje vrednosti 1 in 9 iz `YFastTrie` in 13.8.

Lema 13.1. *Naj bo* x *celo število shranjeno v* `YFastTrie`*, spremenljivka* n_x *pa naj predstavlja število elementov v poddrevesu* t*, ki vsebuje* x*. Velja* $E[n_x] \leq 2w - 1$.

Dokaz. Omenjeno v 13.10. Naj $x_1 < x_2 < \cdots < x_i = x < x_{i+1} < \cdots < x_n$ opisuje elemente shranjene v `YFastTrie`. Poddrevo t vsebuje nekatere elemente večje kot, ali enake x. Ti elementi so $x_i, x_{i+1}, \ldots, x_{i+j-1}$, kjer je x_{i+j-1} edini od teh elementov, pri katerem je met kovanca izveden v metodi `add`(x) vrnil grb. Z drugimi besedami, $E[j]$ je pričakovano število metov kovanca, ki jih potrebujemo, da pridobimo prvi grb. [2] Vsak met kovanca je neodvisen, grb se pojavi z vrjetnostjo $1/w$, velja $E[j] \leq w$. (Oglej si **??** za analizo primera $w = 2$.)

Podobno, elementi t, ki so manjši kot x so $x_{i-1}, \ldots, x_{i-k}$, kjer se je v vseh k metov kovanca pojavila cifra, in met kovanca x_{i-k-1} predstavlja grb. Torej velja, $E[k] \leq w - 1$, ker je to isto metanje kovanca glede na prejšnji odstavek, vendar v tem primeru zadnji met ni bil štet. V povzetku

[2] Ta analiza ignorira dejstvo, da j nikoli ne preseže $n - i + 1$. Kakorkoli, to zgolj zmanjša vrednost $E[j]$, zgornja meja pa je še vedno enaka.



Slika 13.10: Število elementov v poddrevesu t, ki vsebujejo x je določeno z metanjem dveh kovancev.

$n_x = j + k$, torej velja

$$\mathrm{E}[n_x] = \mathrm{E}[j+k] = \mathrm{E}[j] + \mathrm{E}[k] \leq 2w - 1 \ . \qquad \square$$

13.1 Je zadnji del v dokazu teorema, ki povzema učinkovitost `YFastTrie`:

Izrek 13.3. *`YFastTrie` impelementira `SSet` vmestnik za `w`-bitna cela števila. `YFastTrie` podpira operacije* `add(x)`, `remove(x)`, *in* `find(x)` *v pričakovanem času* $O(\log w)$ *na operacijo. Prostor, ki ga `YFastTrie` porabi za hrambo* `n` *vrednosti je* $O(n + w)$.

Dodaten člen w pri prostorski zahtevnosti prihaja iz dejstva, da `xft` vedno hrani vrednost $2^w - 1$. Implementacija je lahko drugačna (v zakup moramo vzeti dodajanje kode) in ni potrebno hraniti te vrednosti. V tem primeru prostorska zahtevnost teorema postane $O(n)$.

13.4 Razprava in vaje

Prvo podaktovno strukturo, ki zagotavlja časovno zahtevnost $O(\log w)$ za operacije `add(x)`, `remove(x)`, in `find(x)` je predlagal van Emde Boas in je od takrat poznana kot *van Emde Boas* (or *razslojeno*) *drevo* [**?**]. Prvotna van Emde Boas struktura je imela velikost 2^w in je bila zato nepraktična za večja cela števila.

Podatkovni strukturi `XFastTrie` in `YFastTrie` je odkril Willard [**?**]. Struktura `XFastTrie` je močno povezana z drevesom van Emde Boas; na primer, razpršene tabele v `XFastTrie` nadomestijo matrike v drevesu van Emde Boas. To pomeni, da drevo van Emde Boas hrani matriko dolžine 2^i namesto razpršene tabele `t[i]`.

Druga struktura za hranitev celih števil so Fredman in Willardova fuzijska drevesa [**?**]. Ta struktura lahko hrani `n` `w`-bitnih števil v prostoru $O(n)$ tako, da se operacija `find(x)` izvede v času $O((\log n)/(\log w))$. S kombinacijo fuzijskih dreves, ko je $\log w > \sqrt{\log n}$ in `YFastTrie`, ko je $\log w \leq \sqrt{\log n}$, pridobimo prostorno podatkovno strukturo $O(n)$, ki lahko implementira operacijo `find(x)` v času $O(\sqrt{\log n})$. Nedavni rezultati spodnje meje Pătraşcu in Thorup [**?**] kažejo na to, da so ti rezultati bolj ali manj optimalni, vsaj kar se tiče struktur, ki porabijo le $O(n)$ prostora.

Naloga 13.1. Sestavi in implementiraj poenostavljeno različico `BinaryTrie`, ki nima kazalcev povezanega seznama ali skakalnih kazalcev, operacija `find(x)` pa teče v $O(w)$ času.

Naloga 13.2. Sestavi in izpelji poenostavljeno implementacijo `XFastTrie`, ki ne uporablja dvojiškega drevesa. Namesto tega naj vaša implementacija vse hrani v dvojno povezanem seznamu in v $w+1$ razpršenih tabelah.

Naloga 13.3. `BinaryTrie` si lahko predstavljamo kot strukturo, ki hrani bitne nize dolžine `w` na tak način, da je vsak bitni niz predstavljen kot pot, od korena do lista. Uporabite to idejo pri izvedbi `SSet`, ki hrani nize spremenljive dolžine in implementira `add(s)`, `remove(s)`, in `find(s)` v času sorazmernem dolžini `s`.
Namig: Vsako vozlišče v vaši podatkovni strukturi naj hrani razpršeno tabelo, ki je indeksirana z vrednostjo znaka.

Naloga 13.4. Za število `x` $\in \{0,\ldots 2^w - 1\}$, kjer $d(x)$ pomeni razliko med `x` in vrednostjo, ki jo vrne `find(x)` [če `find(x)` vrne `null`, potem določi $d(x)$ kot 2^w]. Na primer, če `find(23)` vrne 43, potem $d(23) = 20$.

1. Sestavi in implementiraj spremenjeno različico operacije `find(x)` v `XFastTrie`, ki se izvaja v času $O(1 + \log d(x))$. Nasvet: Razpršena tabela $t[w]$ vsebuje vse vrednosti, `x`, kot so $d(x) = 0$, torej bi bilo tu najbolje začeti.

2. Sestavi in implementiraj spremenjeno različico operacije `find(x)` v `XFastTrie`, ki se izvaja v času $O(1 + \log\log d(x))$.

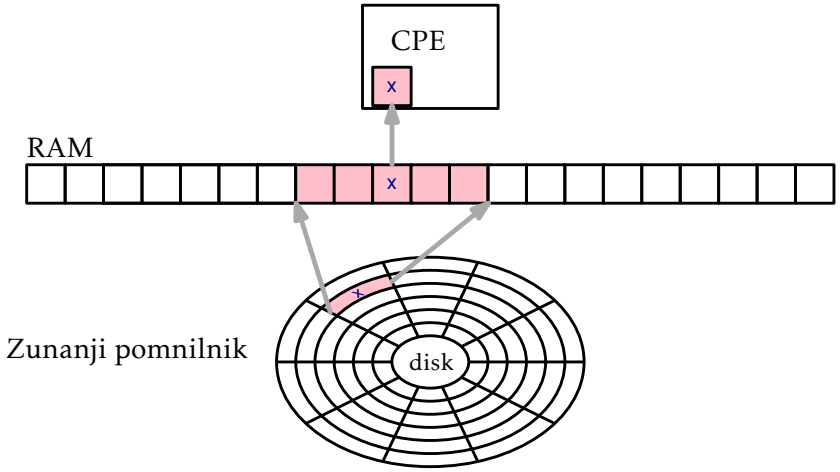
Poglavje 14

Iskanje v zunanjem pomnilniku

Skozi knjigo smo uporabljali w-bitni besedni-RAM model računanja, katerega smo opredelili v 1.4. Implicitna predpostavka tega modela je, da ima naš računalnik dovolj velik bralno-pisalni pomnilnik za shranjevanje vseh podatkov v podatkovni strukturi. V nekaterih primerih ta predpostavka ni veljavna. Obstajajo zbirke podatkov, ki so tako velike, da noben računalnik nima dovolj glavnega pomnilnika za njihovo shranjevanje. V takih primerih se mora aplikacija zateči k shranjevanju podatkov na pomožni, zunanji pomnilniški medij, kot je trdi disk, SSD disk ali celo omrežni datotečni strežnik (ki ima lastno zunanje shranjevanje).

Dostopanje do elementa v zunanjem pomnilniku je zelo počasno. Trdi disk v računalniku, na katerem je bila spisana ta knjiga, ima povprečen čas dostopa 19 ms, SSD disk pa ima povprečen čas dostopa 0,3 ms. Za primerjavo, bralno-pisalni pomnilnik v računalniku ima povprečen čas dostopa manj kot 0,000113 ms. Dostop do RAM-a je več kot 2.500-krat hitrejši kot dostop do SSD diska, ter več kot 160.000-krat hitrejši kot dostop do trdega diska.

Te hitrosti so dokaj tipične; dostopanje do naključnega bajta v RAM-u je tisočkrat hitrejše kot dostopanje do naključnega bajta na trdem disku ali SSD disku. Čas dostopa pa vseeno ne pove vsega. Ko dostopamo do bajta na trdem disku ali SSD disku je prebran celoten *blok* diska. Vsak izmed diskov na računalniku ima velikost bloka 4 096; vsakič, ko preberemo en bajt, nam disk vrne blok, ki vsebuje 4 096 bajtov. Če našo podatkovno strukturo skrbno organiziramo, to pomeni, da z vsakim dostopom do diska dobimo 4 096 bajtov, ki so nam v pomoč pri dokončanju



Slika 14.1: V modelu zunanjega pomnilnika, dostop do posameznega elementa x v zunanjem pomnilniku, zahteva branje celotnega bloka, ki vsebuje x, v glavni pomnilnik.

operacije.

To je ideja računanja z *modelom zunanjega pomnilnika*, shematsko prikazanega v 14.1. Pri tem modelu ima računalnik dostop do velikega zunanjega pomnilnika, kjer so vsi podatki. Ta pomnilnik je razdeljen na spominske *bloke*, kjer vsak vsebuje B besed. Računalnik ima tudi omejen notranji pomnilnik na katerem lahko opravlja izračune. Čas za prenos bloka med notranjim in zunanjim pomnilnikom je konstanten. Izračuni izvedeni v notranjem pomnilniku so *zanemarljivi*; ne vzamejo nič časa. Da so izračuni na notranjem pomnilniku zanemarljivi, se morda sliši malo nenavadno, vendar le preprosto poudarja dejstvo, da je zunanji pomnilnik toliko počasnejši od RAM-a.

V popolnem modelu zunanjega pomnilnika je velikost notranjega pomnilnika tudi parameter. Vendar pa za podatkovne strukture opisane v tem poglavju zadošča, da imamo notranji pomnilnik velikosti $O(B + \log_B n)$. To pomeni, da mora biti pomnilnik sposoben shraniti konstantno število blokov in rekurziven sklad višine $O(\log_B n)$. V večini primerov, izraz $O(B)$ prevladuje pri zahtevah po pomnilniku. Na primer, tudi pri relativno majhni vrednosti $B = 32$, $B \geq \log_B n$ za vse $n \leq 2^{160}$. V desetiškem

zapisu, $B \geq \log_B \mathtt{n}$ za vse

$$\mathtt{n} \leq 1\,461\,501\,637\,330\,902\,918\,203\,684\,832\,716\,283\,019\,655\,932\,542\,976 \ .$$

14.1 Bločna shramba

Pojem zunanjega pomnilnika vključuje veliko število različnih naprav, od katerih ima vsaka svojo velikost bloka in je dostopna s svojo zbirko sistemskih klicev. Da poenostavimo razlago tega poglavja in se osredotočimo na skupne ideje, povzamemo zunanje pomnilniške naprave z objektom bločna shramba. Bločna shramba hrani zbirko spominskih blokov, kjer ima vsak velikost B. Vsak blok je enolično določen s celoštevilskim indeksom. Bločna shramba podpira sledeče operacije:

1. `readBlock(i)`: Vrne vsebino bloka z indeksom `i`.
2. `writeBlock(i,b)`: Zapiše vsebino bloka `b` v blok z indeksom `i`.
3. `placeBlock(b)`: Vrne nov indeks in shrani vsebino bloka `b` na ta indeks.
4. `freeBlock(i)`: Sprosti blok z indeksom `i`. To nakazuje, da vsebina tega bloka ni več v uporabi in, da se zunanji pomnilnik, ki je bil dodeljen temu bloku, lahko ponovno uporabi.

Bločno shrambo si najlažje predstavljamo tako, da si ga zamislimo kot shranjevanje datoteke na disk, kateri je razdeljen na bloke, kjer vsak vsebuje B bajtov. Na ta način `readBlock(i)` in `writeBlock(i,b)` preprosto bereta in zapisujeta bajte $\mathtt{i}B,\ldots,(\mathtt{i}+1)B-1$ te datoteke. Poleg tega bi preprosta bločna shramba lahko vodila *prosti seznam* blokov, ki so na voljo za uporabo. Bloki, sproščeni s `freeBlock(i)`, so dodani prostemu seznamu. Na ta način lahko `placeBlock(b)` uporabi blok iz prostega seznama ali, če nobeden ni na voljo, doda nov blok na konec datoteke.

14.2 B-drevesa

V tem poglavju bomo razpravljali o posplošitvah dvojiških dreves, imenovanih B-drevesa, ki so učinkovita predvsem v zunanjem pomnilniškem

modelu. Alternativno se na B-drevesa lahko gleda kot na posplošitev 2-4 dreves, opisana v poglavju 9.1. (2-4 drevo je posebni primer B-drevesa, ki ga dobimo z določitvijo $B = 2$.)

Za katerokoli število $B \geq 2$ je *B-drevo*, drevo, pri katerem imajo vsi listi enako globino in vsako notranjo vozlišče (z izjemo korena), `u`, ima najmanj B otrok in največ $2B$ otrok. Otroci vozlišča `u` so shranjeni v polju `u.children`. Zahtevano število otrok ne velja pri korenu, ki pa ima lahko število otrok med 2 in $2B$.

Če je višina B-drevesa h, iz tega sledi, da število listov v B-drevesu ℓ, izpolnjuje naslednji neenakosti:

$$2B^{h-1} \leq \ell \leq 2(2B)^{h-1} \ .$$

Vzamemo logaritem iz prve neenakosti in preuredimo. Dobimo:

$$\begin{aligned} h &\leq \frac{\log \ell - 1}{\log B} + 1 \\ &\leq \frac{\log \ell}{\log B} + 1 \\ &= \log_B \ell + 1 \ . \end{aligned}$$

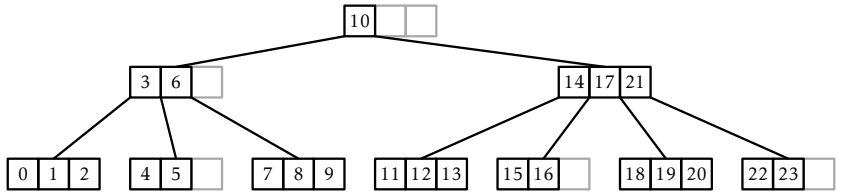
Višina B-drevesa je sorazmerna logaritmu števila listov z osnovo B.

Vsako vozlišče, `u`, v B-drevesu shranjuje polje ključev `u.keys`$[0],\ldots,$`u.keys`$[2B-1]$. Če je `u` notranje vozlišče z k otroci, potem je število ključev, ki so shranjeni v `u` natanko $k-1$ in ti so shranjeni v `u.keys`$[0],\ldots,$`u.keys`$[k-2]$. Ostalih $2B-k+1$ mest v polju `u.keys` je nastavljeno na `null`. Če je `u` notranje vozlišče in ni koren, potem `u` vsebuje med $B-1$ in $2B-1$ ključev. Ključi v B-drevesu so razvrščeni podobno kot ključi v dvojiškem iskalnem drevesu. Za vsako vozlišče `u`, ki shranjuje $k-1$ ključev velja:

$$\texttt{u.keys}[0] < \texttt{u.keys}[1] < \cdots < \texttt{u.keys}[k-2] \ .$$

Če je `u` notranje vozlišče, potem za vsak `i` $\in \{0,\ldots,k-2\}$ velja, da `u.keys[i]` je večji od vseh ključev shranjenih v poddrevesu zakoreninjenega na `u.children[i]` vendar manjši od vseh ključev shranjenih v poddrevesu, ki je zakoreninjen na `u.children[i + 1]`.

$$\texttt{u.children}[\texttt{i}] \prec \texttt{u.keys}[\texttt{i}] \prec \texttt{u.children}[\texttt{i}+1] \ .$$



Slika 14.2: B-drevo, $B = 2$.

Primer B-drevesa z $B = 2$ je prikazan na sliki 14.2.

Upoštevajte, da so podatki shranjeni v vozliščih B-drevesa velikosti $O(B)$. Zato je v nastavitvah zunanjega pomnilnika vrednost B za B-drevo določena tako, da celotno vozlišče lahko ustreza enemu zunanje pomnilniškemu bloku. V tem primeru je čas izvajanja operacij na B-drevesu v zunanjem spominskem modelu sorazmerno številu vozlišč, ki jih obiščemo (branje ali pisanje) med operacijo.

Poglejmo si primer. Če ključe predstavljajo 4 bajtna števila in indeksi vozlišč so prav tako veliki 4 bajte, potem nastavitev $B = 256$ pomeni, da vsako vozlišče hrani

$$(4 + 4) \times 2B = 8 \times 512 = 4096$$

bajtov podatkov. To bi bila odlična vrednost B za trdi disk ali pogon SSD (predstavljen v uvodu tega poglavja), kateri ima velikost bloka 4096 bajtov.

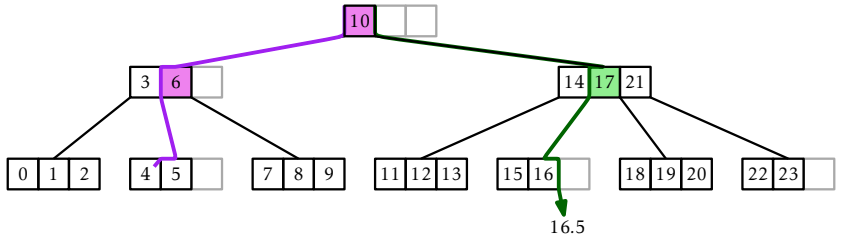
`BTree` razred, ki implementira B-drevo, vsebuje `BlockStore`, `bs`,ki vsebuje `BTree` vozlišča in prav tako indeks, `ri`, korena. Kot ponavadi, število `n` predstavlja količino podatkov v podatkovni strukturi:

`BTree`

```
int n; // number of elements stored in the tree
int ri; // index of the root
BlockStore<Node*> bs;
```

14.2.1 Iskanje

Implementacija operacije `find(x)`, ilustrirana v 14.3, je posplošitev operacije `find(x)` v dvojiškem iskalnem drevesu. Iskanje `x`-a se začne v korenu.



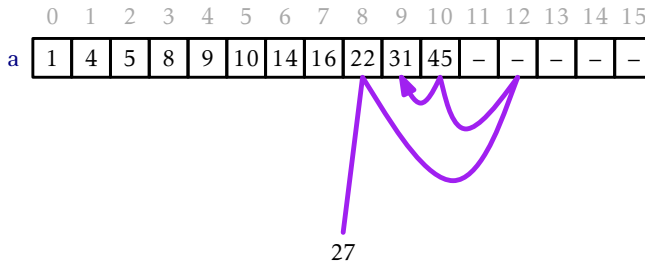
Slika 14.3: Uspešno iskanje (vrednosti 4) in neuspešno iskanje (za vrednost 16.5) v *B*-drevesu. Obarvana vozlišča predstavljajo, kje se je vrednost med iskanjem `z`ja spremenila.

Z uporabo ključev, shranjenih v vozlišču, `u`, določimo v katerem otroku od `u` bomo nadaljevali iskanje.

Bolj natančno, v vozlišču `u` iskanje preveri če je `x` shranjen v `u.keys`. Če je, je bil `x` najden in iskanje je zaključeno. V nasprotnem primeru, najdemo najmanjše število `i`, da je `u.keys[i]` > `x` in nadaljujemo iskanje v poddrevesu zakoreninjenem na `u.children[i]`. Če noben ključ v `u.keys` ni večji od `x`, potem iskanje nadaljujemo v najbolj desnem otroku od `u`. Tako kot pri dvojiškem iskalnem drevesu, si algoritem zapolni nedavno viden ključ, `z`, ki je večji od `x`. V primeru, ko `x` ni najden, se `z` vrne kot najmanjša vrednost, ki je večja ali enaka `x`.

BTree
```
T find(T x) {
  T z = null;
  int ui = ri;
  while (ui >= 0) {
    Node *u = bs.readBlock(ui);
    int i = findIt(u->keys, x);
    if (i < 0) return u->keys[-(i+1)]; // found it
    if (u->keys[i] != null)
      z = u->keys[i];
    ui = u->children[i];
  }
  return z;
}
```

Osrednjega pomena za metodo `find(x)` je metoda `findIt(a,x)`, ki išče v `null`-napolnjeno urejeno polje, `a`, vrednost `x`. Ta metoda, predstavljena



Slika 14.4: Izvajanje metode `findIt(a,27)`.

v 14.4, deluje za vsako polje, `a`, kjer je $a[0],\ldots,a[k-1]$ urejeno zaporedje ključev in so $a[k],\ldots,a[a.length-1]$ vsi postavljeni na `null`. Če je `x` v polju na mestu `i`, potem metoda `findIt(a,x)` vrne $-i-1$. V nasprotnem primeru vrne najmanjši indeks, `i`, za katerga velja, da $a[i] > x$ ali $a[i] =$ `null`.

BTree
```
int findIt(array<T> &a, T x) {
  int lo = 0, hi = a.length;
  while (hi != lo) {
    int m = (hi+lo)/2;
    int cmp = a[m] == null ? -1 : compare(x, a[m]);
    if (cmp < 0)
      hi = m;       // look in first half
    else if (cmp > 0)
      lo = m+1;     // look in second half
    else
      return -m-1; // found it
  }
  return lo;
}
```

Metoda `findIt(a,x)` uporabi dvojiško iskanje , ki razpolovi iskanje pri vsakem koraku. Za delovanje porabi $O(\log(a.length))$ časa. V našem primeru, $a.length = 2B$, zato `findIt(a,x)` porabi $O(\log B)$ časa.

Čas delovanja obeh operacij B-drevesa `find(x)` lahko analiziramo v običajnem besednem-RAM modelu (kjer štejemo vsak ukaz) in v zunanjem pomnilniškem modelu (kjer štejemo samo število obiskanih vozlišč). Ker vsak list v B-drevesu shranjuje vsaj en ključ in je višina B-drevesa z ℓ listi $O(\log_B \ell)$, je višina od B-drevesa, ki shranjuje `n` ključev $O(\log_B n)$.

Zato je v zunanjem pomnilniškem modelu čas, ki ga porabi operacija `find(x)` $O(\log_B n)$. Da določimo čas delovanja v RAM modelu, moramo računati čas klicanja operacije `findIt(a,x)` za vsako vozlišče, ki ga obiščemo. Čas delovanja operacije `find(x)` v modelu besedni RAM je

$$O(\log_B n) \times O(\log B) = O(\log n) \ .$$

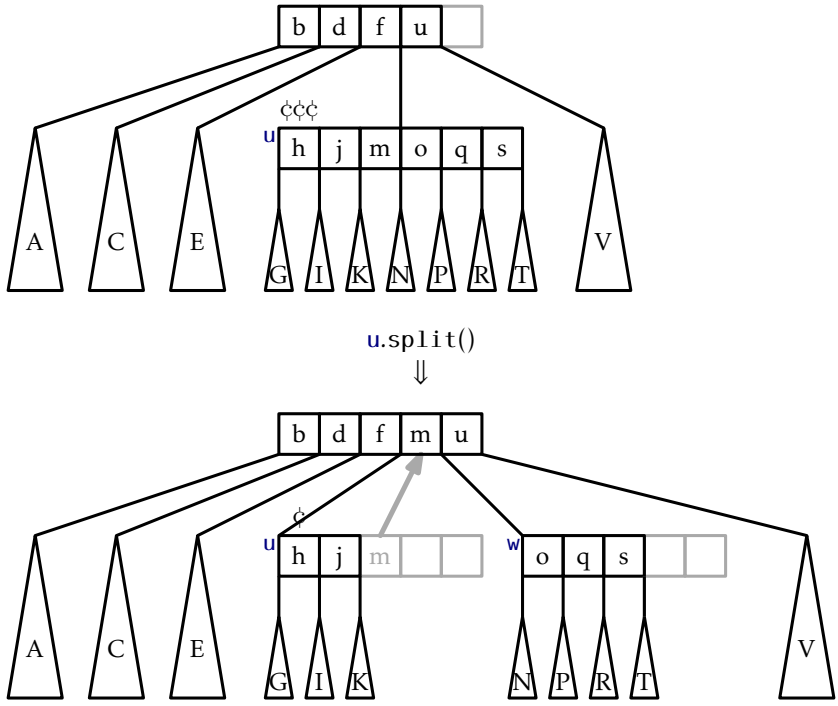
14.2.2 Dodajanje

Ena glavnih razlik med podatkovnima strukturama B-dreves in dvojiških iskalnih dreves (6.2) je, da vozlišča B-dreves ne hranijo kazalcev na njihove starše. Vzrok tega bomo razložili malce kasneje. Ker kazalci na starše ne obstajajo, pomeni, da je operaciji `add(x)` in `remove(x)` v B-drevesih najlažje implementirani s pomočjo rekurzije.

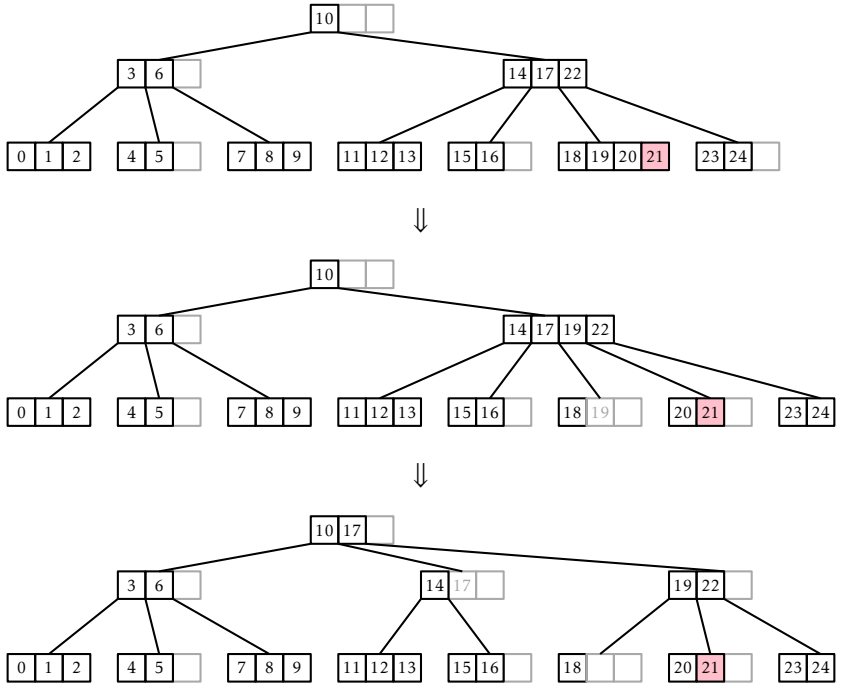
Kot za vsa uravnotežena iskalna drevesa je tudi tu potrebno uravnoteženje drevesa, če se pri izvajanju operacije `add(x)` drevo izrodi. Pri B-drevesih za to skrbi *razdeljevanje* vozlišč. Za nadaljevanje glejte 14.5. Čeprav razdeljevanje deluje na dveh plasteh drevesa, je najbolj razumljivo, kot operacija, ki vzame vozlišče `u`, ki vsebuje $2B$ ključev in ima $2B+1$ otrok. Ustvari novo vozlišče `w`, ki podeduje `u.children`$[B],\ldots,$`u.children`$[2B]$. Novo vozlišče `w` prav tako vzame največje ključe B, `u.keys`$[B],\ldots,$`u.keys`$[2B-1]$ od vozlišča `u`. Na tej točki ima `u` B otrok in B ključev. Dodaten ključ, `u.keys`$[B-1]$, se posreduje staršem vozlišča `u`, posreduje pa se tudi vozlišče `w`.

Opazimo, da operacija razdeljevanja spreminja tri vozlišča: `u`, starše vozlišča `u` in novo vozlišče `w`. Sedaj smo prišli do odgovora, zakaj vozlišča B-dreves ne ohranjajo kazalcev na starše. Če bi jih, bi morali vsem $B+1$ otrokom, ki so podedovani vozlišču `w` popraviti kazalce na njihove starše. Število dostopov do zunanjega pomnilnika bi se povečalo s 3 na $B+4$ dostope. To bi poslabšalo učinkovitost B-drevesa pri večjih številih B.

Metoda `add(x)` v B-drevesih je prikazana v 14.6. V višji plasti metoda poišče list, `u`, v katerega bo dodala vrednost `x`. Če dodajanje pozroči, da `u` postane prepoln (ker že vsebuje $B-1$ ključev), se `u` razdeli. Lahko se zgodi, da postanejo tudi starši prepolni. V tem primeru se razdelijo tudi starši. To lahko spet povzroči deljenje prastaršev vozlišča `u` in tako naprej. To se vzpenja po drevesu toliko časa, dokler ne doseže vozlišča,



Slika 14.5: Razdeljvanje vozlišča u v B-drevesu ($B = 3$). Opazimo, da se ključ u.keys[2] = m posreduje iz u njegovim staršem.



Slika 14.6: Operacija add(x) v *B*-drevesu. Dodajanje vrednosti 21, dva vozlišča se razdelita

ki ni prepoln ali dokler se koren drevesa ne razdeli. V prvem primeru se postopek ustavi. V drugem primeru, se ustvari novo vozlišče, katerega otroci postanejo pridobljena vozlišča pri razdelitvi prvotnega korena.

Povzetek metode add(x) je, da se sprehaja od korena do iskanega(x) lista, doda x v ta list, se začne pomikati nazaj proti korenu, razdeli vsa prepolna vozlišča na katere naleti na poti navzgor. S tem preletom v mislih, se lahko sedaj spustimo v detajle, kako naj bo ta rekurzivna metoda implementirana.

Večino dela add(x) je narejenega z metodo addRecursive(x,ui), katera doda vrednost x v poddrevo, katerega koren u, ima identifikator ui. Če je u list, se x enostavno vsavi v u.keys, sicer se doda rekurzivno v poddrevo ustreznega sina u' od u. Rezultat tega rekurzivnega klica je ponavadi null, ampak lahko je tudi referenca na novo kreirano vozlišče w, kateri je

nastal zaradi razdelitve u'. V tem primeru u podeduje w in vzame njegovo prvo vrednost, ter dokonča razdelitev na u'.

Ko je bila vrednost x dodana (ali v u ali v potomce u), metoda addRecursive(x,ui) preveri, če u hrani preveč (več kot $2B-1$) ključev. V primeru ko jih hrani preveč, se mora u *razdeliti* z klicom metode u.split(). Rezultat klica u.split() je novo vozlišče, ki je uporabljeno kot rezultat metode addRecursive(x,ui).

BTree
```
Node* addRecursive(T x, int ui) {
  Node *u = bs.readBlock(ui);
  int i = findIt(u->keys, x);
  if (i < 0) throw(-1);
  if (u->children[i] < 0) { // leaf node, just add it
    u->add(x, -1);
    bs.writeBlock(u->id, u);
  } else {
    Node* w = addRecursive(x, u->children[i]);
    if (w != NULL) {  // child was split, w is new child
      x = w->remove(0);
      bs.writeBlock(w->id, w);
      u->add(x, w->id);
      bs.writeBlock(u->id, u);
    }
  }
  return u->isFull() ? u->split() : NULL;
}
```

Metoda addRecursive(x,ui) je pomožna metoda metode add(x), katera kliče addRecursive(x,ri), da vstavi x v koren B-drevesa. Če addRecursive(x,ri) povzroči, da se koren razdeli, se ustvari nov koren in si za svoje otroke vzame otroke starega korena in otroke novega vozlišča, pridobljenega pri razdelitvi starega korena.

BTree
```
bool add(T x) {
       Node *w;
       try {
         w = addRecursive(x, ri);
       } catch (int e) {
         return false; // adding duplicate value
       }
```

```
        if (w != NULL) {   // root was split, make new root
    Node *newroot = new Node(this);
    x = w->remove(0);
    bs.writeBlock(w->id, w);
    newroot->children[0] = ri;
    newroot->keys[0] = x;
    newroot->children[1] = w->id;
    ri = newroot->id;
    bs.writeBlock(ri, newroot);
        }
        n++;
        return true;
}
```

Metodo `add(x)` in pomožno metodo `addRecursive(x,ui)` lahko analiziramo v dveh fazah:

faza ugrezanja: Med fazo ugrezanja rekurzije, preden je `x` dodan, imamo dostop do zaporedja vozlišč B-dreves in nad vsakim vozliščem kličemo metodo `findIt(a,x)`. Kot pri metodi `find(x)` to potrebuje $O(\log_B n)$ časa v zunanjem spominskem modelu in $O(\log n)$ časa v modelu RAM.

faza vzpenjanja: Med fazo vzpenjanja rekurzije, po tem ko je `x` dodan, lahko to izvede največ $O(\log_B n)$ delitev. Vsaka razdelitev vsebuje tri vozlišča, tako,da ta faza porabi $O(\log_B n)$ časa v zunanjem spominskem modelu. Vendar vsaka razdelitev zahteva premikanje B ključev in otrok iz enega vozlišča na drugega, tako da porabi $O(B \log n)$ časa v modelu RAM.

Spomnimo, da je lahko vrednost B precej velika, veliko večja kot $\log n$. Zato je v modelu RAM, dodajanje vrednosti v B-drevo lahko veliko počaseje kot dodajanje v uravnovešeno binarno iskalno drevo. Kasneje v 14.2.4, bomo pokazali, da situacija ni tako zelo slaba; amortizacijska številka operacij razdelitve med izvajanjem operacije `add(x)` je konstantna. To kaže na to, da (amortiziran) izvajalni čas operacije `add(x)` v modelu RAM $O(B + \log n)$.

14.2.3 Odstranjevanje

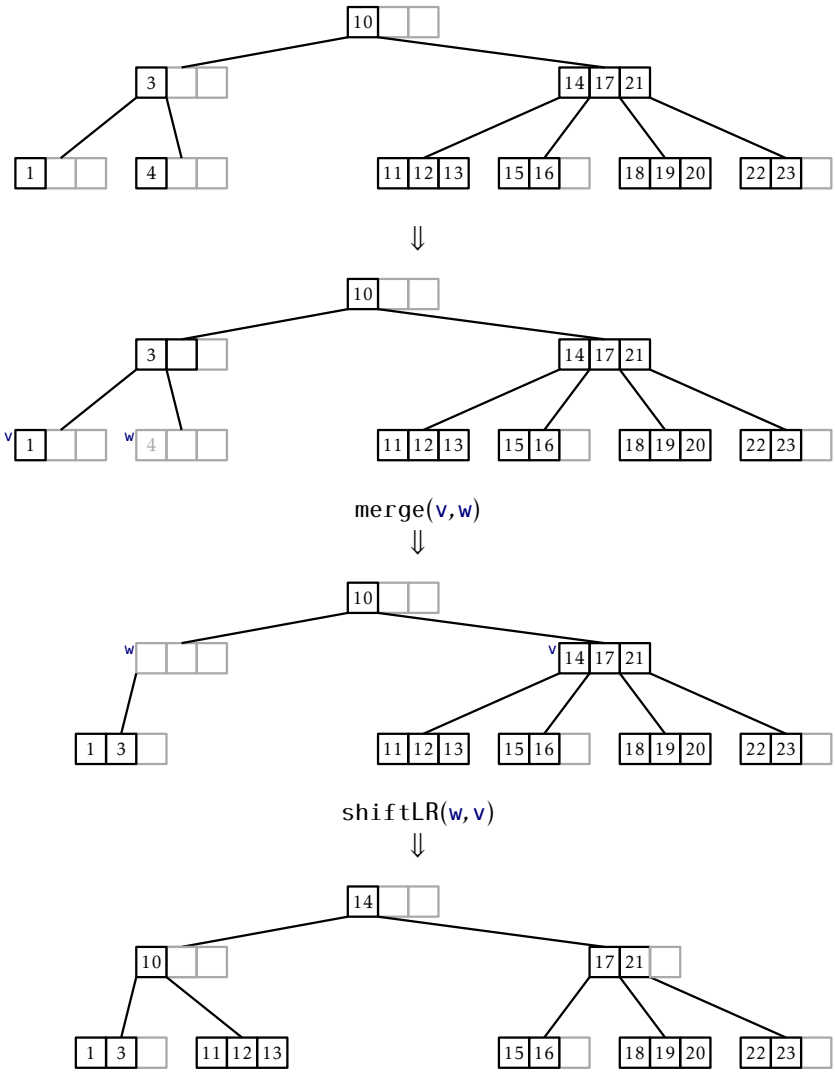
Operacija `remove(x)` v `BTree` je prav tako najlažje implementirana kot rekurzivna metoda. Čeprav rekurziven način implementacije metode `remove(x)` razširi kompleksnost čez več metod, je celoten proces, kot je prikazan v 14.7, dokaj preprost. S prestavljanjem ključev okrog problem skrčimo na odstranitev vrednosti, `x'`, iz določenega lista, `u`. Odstranitev `x'` lahko pusti `u` z manj kot $B-1$ ključi; takšen dogodek se imenuje *spodnja prekoračitev*.

V primeru spodnje prekoračitve, si `u` sposodi ključe od ali je združen z enim od svojih sorodnikov. Če pride do združitve `u` s sorodnikom, bo sedaj `u`-jev starš imel enega otroka in enega ključa manj, kar lahko povzroči spodnjo prekoračitev `u`-jevega starša; to je ponovno popravljeno z izposojo ali združitvijo, vendar združitev lahko povzroči spodnjo prekoračitev `u`-jevega starega starša. Ta proces se ponavlja vse nazaj do korena, dokler ne pride več do prekoračitve ali se korenova zadnja otroka združita v enega samega. Če se zgodi slednje, je koren odstranjen in njegov preostali otrok postane nov koren.

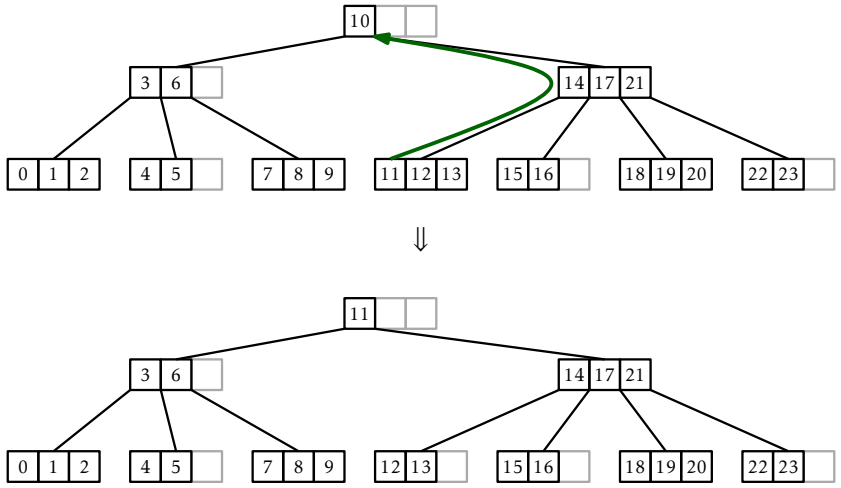
Sledi podroben ogled načina implementacije posameznega koraka. Prva naloga metode `remove(x)` je poiskati element `x`, ki ga želimo dstraniti. Če se `x` nahaja v listu, sledi odstranitev `x` iz tega lista. V nasprotnem primeru, če je `x` najden v `u.keys[i]` za neko notranje vozlišče `u`, algoritem odstrani najmanjšo vrednost, `x'`, v poddrevesu s korenom, ki se nahaja na `u.children[i+1]`. Vrednost `x'` je najmanjša vrednost shranjena v `BTree`, ki je večja od `x`. Vrednost `x'`-a nato zamenja vrednost `x` v `u.keys[i]`. Ta proces je prikazan v 14.8.

Metoda `removeRecursive(x,ui)` je rekurzivna implementacija predhodnega algoritma:

BTree
```
T removeSmallest(int ui) {
  Node* u = bs.readBlock(ui);
  if (u->isLeaf())
    return u->remove(0);
  T y = removeSmallest(u->children[0]);
  checkUnderflow(u, 0);
  return y;
}
```



Slika 14.7: Odstranitev vrednosti 4 iz *B*-drevesa povzroči eno združitev in eno izposojo.



Slika 14.8: Operacija `remove(x)` v B – `drevesu`. Da odstranimo vrednost `x` = 10, jo zamenjamo z $x' = 11$ in odstranimo 11 iz lista, ki jo vsebuje.

```
bool removeRecursive(T x, int ui) {
  if (ui < 0) return false;  // didn't find it
  Node* u = bs.readBlock(ui);
  int i = findIt(u->keys, x);
  if (i < 0) { // found it
    i = -(i+1);
    if (u->isLeaf()) {
      u->remove(i);
    } else {
      u->keys[i] = removeSmallest(u->children[i+1]);
      checkUnderflow(u, i+1);
    }
    return true;
  } else if (removeRecursive(x, u->children[i])) {
    checkUnderflow(u, i);
    return true;
  }
  return false;
}
```

Po rekurzivnem odstranjevanju vrednosti `x` iz `i`-tega otroka `u`-ja mora

`removeRecursive(x,ui)` zagotoviti, da ima ta otrok še vedno vsaj $B-1$ ključev. V predhodni kodi je to zagotovljeno z metodo `checkUnderflow(x, i)`, ki preveri podkoračitev v `i`-temu otroku `u`-ja in jo po potrebi popravi. Naj bo `w` `i`-ti otrok `u`-ja. Če ima `w` samo $B-2$ ključev, ga je treba popraviti, za kar pa potrebujemo `w`-jevega brata, ki je lahko `u`-jev otrok z indeksom $i+1$ ali z indeksom $i-1$. Ponavadi izberemo tistega z indeksom $i-1$, ki je `w`-jev brat neposredno na njegovi levi. Recimo mu `v`. Edini primer v katerem to ne deluje je kadar je $i=0$. V tem primeru uporabimo brata, ki je neposredno na `w`-jevi desni.

BTree
```
void checkUnderflow(Node* u, int i) {
  if (u->children[i] < 0) return;
  if (i == 0)
    checkUnderflowZero(u, i); // use u's right sibling
  else
    checkUnderflowNonZero(u, i);
}
```

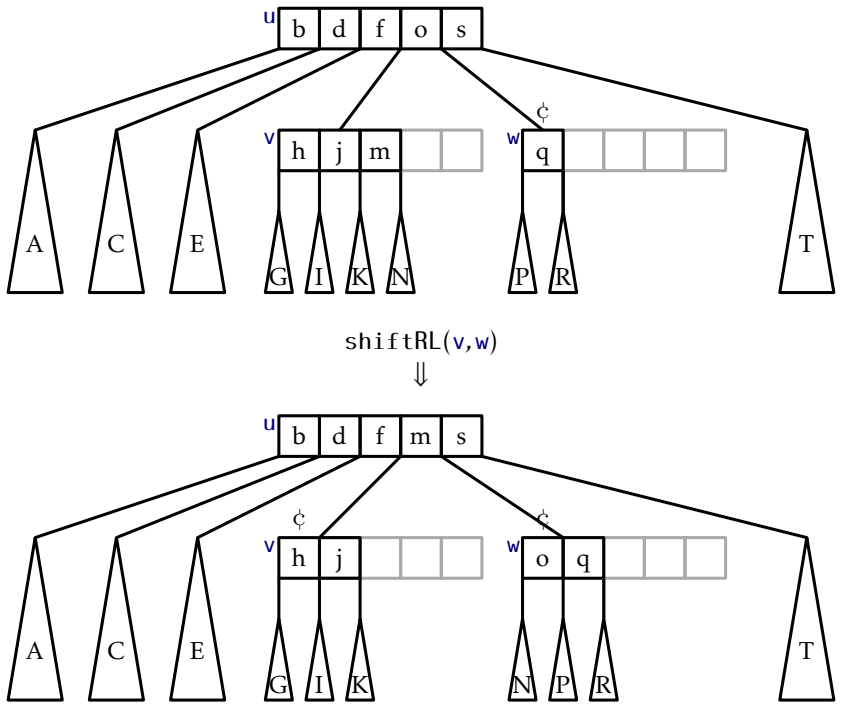
Sedaj se osredotočimo na primer, ko je $i \neq 0$, tako da bo kakršnakoli podkoračitev pri `i`-temu otroku vozlišča `u` popravljena s pomočjo njegovega otroka z indeksom $(i-1)$. Primer, ko je $i=0$ je podoben. Podrobnosti so v izvorni kodi. Da popravimo podkoračitev v vozlišču `w`, moramo temu vozlišču najti več ključev (in po možnosti tudi otrok). To lahko storimo na dva načina:

Izposojanje: Če ima `w` brata `v` z več kot $B-1$ ključi, si lahko `w` od `v`-ja izposodi nekaj ključev (in po možnosti tudi otrok). Natančneje, če ima `v` `size(v)` ključev, imata `v` in `w` skupaj

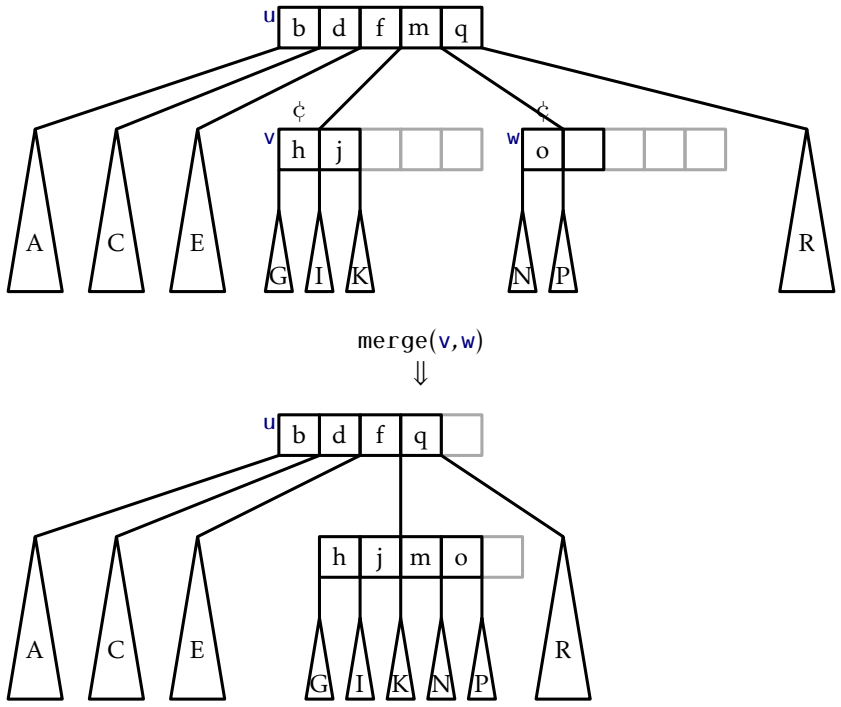
$$B-2+\texttt{size}(\texttt{w}) \geq 2B-2$$

ključev. Torej lahko `v`-jeve ključe prestavimo `w`-ju tako, da imata `v` in `w` vsaj $B-1$ ključev. Ta proces je prikazan v 14.9.

Združevanje: Če ima `v` samo $B-1$ ključev, moramo narediti nekaj bolj zahtevnega, saj `v` ne more posoditi nobenega ključa `w`-ju. Zato vozlišči `w` in `v` *združimo*, kot je prikazano v 14.10. Združevanje je nasprotna operacija razdelitve. Dve vozlišči, ki imata skupaj $2B-3$



Slika 14.9: Če ima v več kot $B-1$ ključev, jih lahko posodi w-ju.



Slika 14.10: Merging two siblings v and w in a B-tree ($B = 3$).

ključev in ju združi v eno samo vozlišče v $2B - 2$ ključi. Dodaten ključ dobimo zato, ker ima po združevanju v-ja in w-ja njun starš u enega otroka manj in mora zato oddati en ključ.

BTree

```
void checkUnderflowZero(Node *u, int i) {
  Node *w = bs.readBlock(u->children[i]);
  if (w->size() < B-1) {  // underflow at w
    Node *v = bs.readBlock(u->children[i+1]);
    if (v->size() > B) { // w can borrow from v
      shiftRL(u, i, v, w);
    } else { // w will absorb w
      merge(u, i, w, v);
      u->children[i] = w->id;
    }
```

```
  }
}
void checkUnderflowNonZero(Node *u, int i) {
  Node *w = bs.readBlock(u->children[i]);
  if (w->size() < B-1) {  // underflow at w
    Node *v = bs.readBlock(u->children[i-1]);
    if (v->size() > B) {  // w can borrow from v
      shiftLR(u, i-1, v, w);
    } else { // v will absorb w
      merge(u, i-1, v, w);
    }
  }
}
```

Da povzamemo, metoda remove(x) v B-drevesu gre od korenskega vozlišča do lista, odstrani ključ x' iz lista u in nato izvede nič ali več operacij združevanja med u-jem in njegovimi predniki in največ eno operacijo izposojanja. Ker pri vsaki operaciji združevanja in izposojanja spreminjamo največ tri vozlišča, in ker se izvede samo $O(\log_B n)$ takih operacij, to v modelu zunanjega pomnilnika porabi $O(\log_B n)$ časa. Kakorkoli že, vsaka operacija združevanja in izposojanja potrebuje $O(B)$ časa v besednem-RAM modelu, zato lahko (za zdaj) za časovno zahtevnost operacije remove(x) trdimo, da spada v razred $O(B \log_B n)$.

14.2.4 Amortizirana analiza B-Dreves

Do sedaj smo pokazali, da je

1. v modelu zunanjega pomnilnika časovna zahtevnost operacij find(x), add(x), in remove(x) v B-drevesu $O(\log_B n)$, in da je
2. v besednem-RAM modelu časovna zahtevnost operacije find(x) $O(\log n)$, časovna zahtevnost operacij add(x) in remove(x) pa $O(B \log n)$.

Naslednja trditev pokaže, da smo precenili število operacij združevanja in razdelitev v B-drevesih.

Lema 14.1. *Če imamo prazno B-drevo in izvedemo m* add(x) *in* remove(x) *operacij, se izvede največ* $3m/2$ *razdelitev, združevanj in izposojanj.*

Dokaz. Dokaz za to je že bil nakazan v 9.3 za poseben primer, ko je $B = 2$. Trditev lahko dokažemo z

1. vsaka razdelitev, združevanje ali izposoja se plača z dvema kovancema (plača se vsakič ko se izvede ena izmed teh operacij); in

2. največ trije kovanci so na razpolago med katerokoli `add(x)` ali `remove(x)` operacijo.

Ker je na razpolago največ m kovancev in vsaka razdelitev, združevanje in izposoja stane dva kovanca, sledi, da se izvede največ $3m/2$ razdelitev, združevanj in izposoj. Kovanci so prikazani z simbolom ¢ v Slikah 14.5, 14.9, in 14.10.

Da lahko vodimo evidenco o kovancih, dokaz uporablja naslednjo *invarianco kovancev*:

Vsako nekorensko vozlišče z $B-1$ ključi shrani tri kovance. Vozlišču, ki ima najmanj B in največ $2B-2$ ključev ni potrebno hraniti kovancev. Sedaj moramo samo še pokazati, da lahko ohranjamo invarianco kovancev in se hkrati držimo trditev 1 in 2 (zgoraj) pri vsaki `add(x)` in `remove(x)` operaciji.

Dodajanja: Metoda `add(x)` ne uporabi nobenih združevanj ali izposojanj, zato lahko pri klicih te metode upoštevamo samo operacije razdelitve.

Vsaka operacija razdelitve ima za vzrok dodajanje ključa vozlišču u, ki že ima $2B-1$ ključev. Ko pride do tega, se u razdeli na dve vozlišči - vozlišče u′ z $B-1$ ključi in vozlišče u″ z B ključi. Pred to operacijo je imelo vozlišče u $2B-1$ ključev in zato tri kovance. Dva kovanca porabimo za operacijo razdelitve in preostali kovanec prenesemo na u′ (ki ima $B-1$ ključev) da ohranimo invarianco kovancev. Tako lahko plačamo za razdelitev in hkrati ohranjamo invarianco konvancev med vsakio operacijo razdelitve.

Edina druga sprememba v vozliščih pri operaciji `add(x)` se zgodi šele po vseh opravljenih razdelitvah, če sploh do njih pride. Ta sprememba vključuje dodajanje novega ključa vozlišču u′. Če je imelo pred tem vozlišče u′ $2B-2$ otrok, jih ima sedaj $2B-1$ in zato prejme tri kovance. Ti kovanci so edini, ki jih dodeli metoda `add(x)`.

Odstranjevanje: Med operacijo remove(x) pride do nič ali več operacij združevanja, katerim lahko sledi ena opercija izposoje. Do združevanja pride ko sta vozliči v in w (vsako z po $B-1$ ključi pred klicem medode remove(x)) združeni v eno vozlišče z $2B-2$ ključi. Vsako takšno združevanje sprosti dva kovanca, s katerima lahko plačamo združevanje.

Po vseh opravljenih operacijah združevanja lahko pride do največ ene operacije izposoje (po tej operaciji ne pride več do združevanj ali izposojanj). Do te operacije izposoje pride samo v primeru, da iz lista v, ki ima $B-1$ ključev, odstranimo ključ. Vozlišče v ima tako en kovanec, ki se porabi za to operacijo izposoje. Ker pa en kovanec ni dovolj, moramo ustvariti še enega.

Ustvarili smo en kovanec in moramo sedaj pokazati, da lahko ohranjamo invarianco kovancev. V najslabšem primeru ima v-jev brat w natanko B ključev pred izposojo, tako da imata oba (v in w) po izposoji $B-1$ ključev. To pomeni da bi morala vsak imeti po en kovanec po končani operaciji. V tem primeru tako ustvarimo dodatna dva kovanca za vozlišči v in w. Ker se operacija izposoje zgodi največ enkrat na klic metode remove(x) to pomeni, da ustvarimo skupaj največ tri kovance, kar ne krši pravil.

Če v metodi remove(x) ne pride do operacije izposoje je to zato, ker se konča z odstranjevanjem ključa iz vozlišča, ki je imelo pred operacijo B ali več ključev. V najslabšem primeru je imelo to vozliče natanko B ključev, zato jih ima po operaciji $B-1$ in potrebuje en kovanec, ki ga ustvarimo.

V vsakem primeru - če se odstranjevanje konča z operacijo izposoje ali ne - je potrebno ustvariti največ tri kovance pri klicu metode remove(x), da se ohranja incarianco kovancev. Dokaz je s tem zaključen. □

Namen dokaza 14.1 je pokazati, da je pri besednem-RAM modelu časovna zahtevnost operacij razdelitev, združevanje in povezovanje pri m add(x) in remove(x) operacijah le $O(Bm)$. To pomeni, da je amortizirana časovna zahtevnost na operacijo samo $O(B)$, torej je amortizirana časovna zahtevnost metod add(x) in remove(x) v besednem-RAM modelu $O(B+\log n)$. To je povzeto v naslednjih trditvah:

Izrek 14.1 (B-Drevesa v zunanjem pomnilniku)**.** *Razred BTree implementira vmesnik SSet. V modelu zunanjega pomnilnika podpira razred BTree operacije* add(x), remove(x) *in* find(x), *katerih časovna zahtevnost je* $O(\log_B n)$.

Izrek 14.2 (Besedni RAM *B*-Drevesa)**.** *Razred BTree implementira vmesnik SSet. V besednem-RAM modelu podpira razred BTree operacije* add(x), remove(x) *in* find(x), *katerih časovna zahtevnost je* $O(\log n)$, *pri čemer zanemarimo ceno razdelitev, združevanj in izposojanj. Če začnemo z praznim BTree in opravimo m* add(x) *in* remove(x) *operacij je časovna zahtevnost razdelitev, združevanj in izposojanj* $O(Bm)$.

14.3 Razprave in vaje

Model računanja v zunanjem pomnilniku sta predstavila Aggarwall in Vitter [**?**].Včasih se imenuje tudi *V/I model* (ang. *I/O model*) ali pa *diskovno dostopni model* (ang. *DAM*).

B-drevesa so pri iskanju v zunanjem pomnilniku to, kar so dvojiška iskalna drevesa pri iskanju v notranjem pomnilniku. *B*-drevesa sta uvedla Bayer in McCreight [**?**] leta 1970 in manj kot deset let kasneje jih naslov članka v ACM computing surveys obravnava kot vseprisotne [**?**]. Tako kot binarnih iskalnih dreves, obstaja veliko različic *B*-dreves, vključno B^+-drevesa, B^*-drevesa, in štetje *B*-dreves. *B*-drevesa so resnično vseprisotna in so primarna podatkovna struktura v mnogih datotečnih sistemih, vključno z Apple-ov HFS+, Microsoftov NTFS, in Linuxov Ext4; vsak večji sistem podatkovnih baz; in shrambah *key-value*, ki se uporablja v računalništvu v oblaku. Nedavna raziskava Graefe-a [**?**] zagotavlja pregled 200+ strani, mnogih sodobnih aplikacij, variant in optimizacij *B*-dreves.

B-drevesa implementirajo vmesnik SSet. Kadar je potreben le vmesnik USet, se lahko uporablja hashing zunanjega pomnilnika Obstajajo programi za hashing zunanjega pomnilnika; za primer glej, Jensen in Pagh [**?**]. V teh primerih implementirajo USet operacije v pričakovanem času $O(1)$ v modelu zunanjega pomnilnika. Vendar pa zaradi različnih razlogov veliko vlog še vedno uporabljajo *B*-drevesa, čeprav so zahtevali le operacije USet.

Eden od razlogov, da so *B*-drevesa tako priljubljena izbira je, da so pogosto uspešnejši od njihove $O(\log_B n)$ predlagane meje časa delovanja. Razlog za to je, ker je vrednost *B* v nastavitvah zunanjega pomnilnika običajno precej velik - na stotine ali celo tisoče. To pomeni, da je 99% ali

celo 99.9% podatkov B-drevesa shranjenih v listih. V sistemu baze podatkov z velikim pomnilnikom, je mogoče shraniti vsa notranja vozlišča B-drevesa v RAM, saj predstavljajo le 1% ali 0.1 celotnega nabora podatkov. Ko se to zgodi, to pomeni, da je iskanje v B-drevesu vključuje zelo hitro iskanje v RAM-u, preko notranjih vozlišč, ki mu sledi enojni dostop do zunanjega pomnilnika za nalaganje listov..

Naloga 14.1. Pokaži kaj se zgodi z ključema 1.5 ter nato z 7.5, ko ju vstavimo v B-drevo, 14.2.

Naloga 14.2. Pokaži kaj se zgodi z ključema 3 in 4, ko ju odstranimo iz B-drevesa v 14.2.

Naloga 14.3. Kakšno je največje število notranjih vozlišč v B-drevesu, ki hrani n ključev (kot funkcija n in B)?

Naloga 14.4. V uvodu trdimo, da B-drevesa potrebujejo notranji pomnilnik velikosti $O(B+\log_B n)$. Vendar implementacija podana tukaj ima večjo pomnilniško zahtevnost.

1. Pokaži, da implementacija za `add(x)` in `remove(x)` metodi podani v tem poglavju uporabljata notranji pomnilnik preporcionalen $B\log_B n$
2. Opiši kako bi lahko te metode preoblikovali, tako da bi zmanjšali njihovo pomnilniško zahtevnost na $O(B+\log_B n)$.

Naloga 14.5. Nariši kredite uporabljene v dokazu 14.1 na drevesih v Figures 14.6 in 14.7. Potrdi, da (z tremi dodatnimi krediti) si je mogoče privoščiti rezcepitev, združitve in sposojanja ter hkrati obdržati kreditno invarianto.

Naloga 14.6. Naredi spremenjeno verzijo B-drevesa, katera ima lahko od B do $3B$ naslednjikov (in zato od $B-1$ do $3B-1$ ključev). Dokaži, da ta nova verzija B-drevesa izvaja samo $O(m/B)$ razcepitve, združitve, in izposojanja v času zaporedja m operacij. (Nasvet: Da bo to delovalo, boste morali biti bolj agresivni z združevanjem, občasno združiti dve vozlišči preden bo to nujno potrebno.)

Naloga 14.7. V tej vaji boste zasnovali spremenjeno metodo za delitev in združevanje v B-drevesih, ki asimptotično zmanjša število delitev, izposojanj in združevanj z upoštevanjem treh vozlišč naenkrat.

1. Naj bo u prepolno vozlišče in naj bo v brat takoj desno od u. Obstajata dva načina, da popravimo prekoračitev pri u:

   (a) u lahko preseli nekaj svojih ključev na v; ali
   (b) u se lahko razdeli in ključi u in v se lahko enakomerno razdelijo med u, v in novo nastalo vozlišče w.

   Pokažite, da se to vedno lahko naredi na način, da imajo po operaciji vsa udeležena vozlišča (največ 3) vsaj $B + \alpha B$ ključev in kvečemu $2B - \alpha B$ ključev, za neko konstantno $\alpha > 0$.

2. Naj bo u vozlišče s premalo ključi in naj bosta v ter w brata vozlišča u. Obstajata dva načina kako popraviti praveliko praznost pri u:

   (a) ključi se lahko enakomerno razdelijo med u, v in w; ali
   (b) u, v, w združimo v dve vozlišči ter razdelimo ključe vozlišč u, v, in w med novonastali vozlišči

   Pokažite, da se, da to vedno narediti na način, tako, da imajo po operaciji vsa udeležena vozlišča (največ 3) vsaj $B + \alpha B$ ključev in kvečjemu $2B - \alpha B$ ključev, za neko konstanto $\alpha > 0$.

3. Pokažite, da je s temi spremembami, število združevanj, izposojanj in delitev, ki se zgodijo nad m operacijami enako $O(m/B)$.

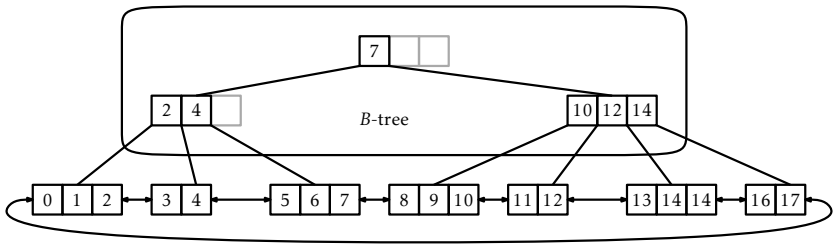
Naloga 14.8. B^+-drevo, ilustrirano na 14.11 hrani vsak ključ v listih, vsak list pa je shranjen kot dvojno povezani seznam. Kot ponavadi, vsak list hrani med $B - 1$ in $2B - 1$ ključi. Nad listi je običajno B-drevo, ki hrani največjo vrednost vsakega lista razen zadnjega.

1. Opišite hitre implementacije metod `add(x)`, `remove(x)` in `find(x)` v B^+-drevesu.

2. Razložite kako učinkovito implementirati metodo `findRange(x,y)` , ki vrne vse vrednosti večje od `x` in manjše ali enake `y` v B^+-drevesu.

3. Implementirajte razred, `BPlusTree`, ki implementira `find(x)`, `add(x)`, `remove(x)`, in `findRange(x,y)`.

4. B^+-drevo podvoji nekatere ključe, saj so shranjeni hkrati v B-drevesu ter v listu. Razložite zakaj se to podvajanje ne pozna toliko na velikih vrednostih od B.



Slika 14.11: B^+-drevo je B-drevo na vrhu dvojno povezanega seznama blokov.

Literatura

[1] Free eBooks by Project Gutenberg. URL: http://www.gutenberg.org/ [cited 2011-10-12].

[2] IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008. doi:10.1109/IEEESTD.2008.4610935.

[3] G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(1259-1262):4, 1962.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] A. Andersson. Balanced search trees made simple. In F. K. H. A. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Algorithms and Data Structures, Third Workshop, WADS '93, Montréal, Canada, August 11–13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1993.

[6] A. Bagchi, A. L. Buchsbaum, and M. T. Goodrich. Biased skip lists. In P. Bose and P. Morin, editors, *Algorithms and Computation, 13th International Symposium, ISAAC 2002 Vancouver, BC, Canada, November 21–23, 2002, Proceedings*, volume 2518 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2002.

[7] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *SIGFIDET Workshop*, pages 107–141. ACM, 1970.

[8] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15–19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 79–79. Springer, 1999.

[9] P. Bose, K. Douïeb, and S. Langerman. Dynamic optimality for skip lists and b-trees. In S.-H. Teng, editor, *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20–22, 2008*, pages 1106–1114. SIAM, 2008.

[10] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In Dehne et al. [**?**], pages 37–48.

[11] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[12] C. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Computer Science Department, Stanford University, 1972.

[13] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, pages 29–44, 2003.

[14] F. K. H. A. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, editors. *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11–14, 1999, Proceedings*, volume 1663 of *Lecture Notes in Computer Science*. Springer, 1999.

[15] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In C. Puech and R. Reischuk, editors, *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, February 22–24,*

1996, Proceedings, volume 1046 of *Lecture Notes in Computer Science*, pages 567–580. Springer, 1996.

[16] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In W. Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13–17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer, 1992.

[17] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[18] A. Elmasry. Pairing heaps with $O(\log \log n)$ decrease cost. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 471–476. Society for Industrial and Applied Mathematics, 2009.

[19] F. Ergun, S. C. Sahinalp, J. Sharp, and R. Sinha. Biased dictionaries with fast insert/deletes. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 483–491, New York, NY, USA, 2001. ACM.

[20] M. Eytzinger. *Thesaurus principum hac aetate in Europa viventium (Cologne)*. 1590. In commentaries, 'Eytzinger' may appear in variant forms, including: Aitsingeri, Aitsingero, Aitsingerum, Eyzingern.

[21] R. W. Floyd. Algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[22] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[23] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[24] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436, 1993.

[25] A. Gambin and A. Malinowski. Randomized meldable priority queues. In *SOFSEM'98: Theory and Practice of Informatics*, pages 344–349. Springer, 1998.

[26] M. T. Goodrich and J. G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Dehne et al. [**?**], pages 205–216.

[27] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2010.

[28] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.

[29] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, 16–18 October 1978, Proceedings*, pages 8–21. IEEE Computer Society, 1978.

[30] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[31] J. E. Hopcroft and R. E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[32] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[33] HP-UX process management white paper, version 1.3, 1997. URL: `http://h21007.www2.hp.com/portal/download/files/prot/files/STK/pdfs/proc_mgt.pdf` [cited 2011-07-20].

[34] M. S. Jensen and R. Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008.

[35] P. Kirschenhofer, C. Martinez, and H. Prodinger. Analysis of an optimized search algorithm for skip lists. *Theoretical Computer Science*, 144:199–220, 1995.

[36] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 31:775–792, 1994.

[37] D. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[38] D. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[39] D. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[40] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transaction on Electronic Computers*, EC-10(3):346–365, 1961.

[41] E. Lehman, F. T. Leighton, and A. R. Meyer. *Mathematics for Computer Science*. 2011. URL: http://courses.csail.mit.edu/6.042/spring12/mcs.pdf [cited 2012-09-06].

[42] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms (SODA'92)*, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[43] Oracle. *The Collections Framework*. URL: http://download.oracle.com/javase/1.5.0/docs/guide/collections/ [cited 2011-07-19].

[44] T. Papadakis, J. I. Munro, and P. V. Poblete. Average search and update costs in skip lists. *BIT*, 32:316–332, 1992.

[45] M. Pătraşcu and M. Thorup. Randomization does not help searching predecessors. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7–9, 2007*, pages 555–564. SIAM, 2007.

[46] W. Pugh. A skip list cookbook. Technical report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, 1989. URL: ftp://ftp.cs.umd.edu/pub/skipLists/cookbook.pdf [cited 2011-07-20].

[47] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[48] Redis. URL: http://redis.io/ [cited 2011-07-20].

[49] S. M. Ross. *Probability Models for Computer Science*. Academic Press, Inc., Orlando, FL, USA, 2001.

[50] R. Sedgewick. Left-leaning red-black trees, September 2008. URL: http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf [cited 2011-07-21].

[51] H. H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Massachusetts Institute of Technology, Digital Computer Laboratory, 1954.

[52] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM conference LISP and Functional Programming (LFP'94)*, pages 185–195, New York, 1994. ACM.

[53] P. Sinha. A memory-efficient doubly linked list. *Linux Journal*, 129, 2005. URL: http://www.linuxjournal.com/article/6828 [cited 2013-06-05].

[54] SkipDB. URL: http://dekorte.com/projects/opensource/SkipDB/ [cited 2011-07-20].

[55] D. Sleator and R. Tarjan. Self-adjusting binary trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25–27 April, 1983, Boston, Massachusetts, USA*, pages 235–245. ACM, ACM, 1983.

[56] S. P. Thompson. *Calculus Made Easy*. MacMillan, Toronto, 1914. Project Gutenberg EBook 33283. URL: http://www.gutenberg.org/ebooks/33283 [cited 2012-06-14].

[57] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.

[58] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[59] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

[60] J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.