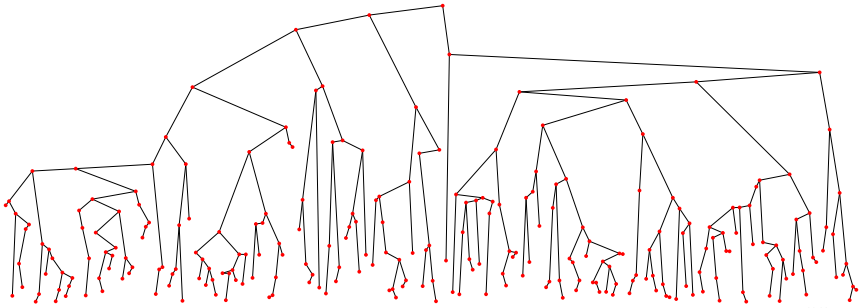


Open Data Structures (za programski jezik C++) v slovenščini

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 Implementacija seznama s poljem **29**

2.1 `ArrayStack`: Implementacija 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 Uravnovešenje . . . 47
2.5.2 Povzetek . . . 49
2.6 RootishArrayStack: A Space-Efficient Array Stack . . . 49
2.6.1 Analysis of Growing and Shrinking . . . 54
2.6.2 Space Usage . . . 54
2.6.3 Summary . . . 55
2.6.4 Computing Square Roots . . . 56
2.7 Discussion and Exercises . . . 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 **87**
4.1 Osnovna struktura . . . 87
4.2 SkiplistSSet: Učinkovit SSet . . . 89
4.2.1 Povzetek . . . 92
4.2.2 Summary . . . 93
4.3 SkiplistList: Učinkovit naključni dostop List . . . 93
4.3.1 Summary . . . 98

4.4 Analiza preskočnega seznama . . . 98
4.5 Discussion and Exercises . . . 102

5 Dvojiška drevesa **103**
5.1 BinaryTree: Osnovno Binarno Drevo . . . 105
5.1.1 Rekurzivni algoritmi . . . 106
5.1.2 Obiskovanje Binarnega drevesa . . . 106
5.2 BinarySearchTree: Neuravnoteženo binarno iskalno drevo 109
5.2.1 Iskanje . . . 110
5.2.2 Vstavljanje . . . 111
5.2.3 Brisanje . . . 114
5.2.4 Povzetek . . . 116
5.3 BinaryTree: Razprava in vaje . . . 116

6 Naključna iskalna binarna drevesa **117**
6.1 Naključna iskalna binarna drevesa . . . 117
6.1.1 Dokaz Lemma 6.1 . . . 120
6.1.2 Povzetek . . . 122
6.2 Treap: Naključno generirano binarno iskalno drevo . . . 123
6.2.1 Povzetek . . . 130

7 Rdeče-Črna Drevesa **133**
7.1 2-4 Trees . . . 134
7.1.1 Dodajanje lista . . . 135
7.1.2 Odstranjevanje lista . . . 135
7.2 RedBlackTree: A Simulated 2-4 Tree . . . 138
7.2.1 Rdeče-Črna drevesa in 2-4 Drevesa . . . 139
7.2.2 Levo-viseca Rdece-Crna Drevesa . . . 142
7.2.3 Dodajanje . . . 144
7.2.4 Odstranitev . . . 147
7.3 Summary . . . 152
7.4 Discussion and Exercises . . . 153

8 Kopice **159**
8.1 BinarnaKopica: inplicitno binarno drevo . . . 159
8.1.1 Summary . . . 163
8.2 MeldableHeap: A Randomized Meldable Heap . . . 165

8.2.1 Analysis of merge(h1,h2) . . . 168
8.2.2 Summary . . . 169
8.3 Discussion and Exercises . . . 170

9 Grafi **175**
9.1 AdjacencyMatrix: Predstavitev grafov z uporabo matrik . 177
9.2 AdjacencyLists: A Graph as a Collection of Lists . . . 180
9.3 Graph Traversal . . . 184
9.3.1 Breadth-First Search . . . 184
9.3.2 Depth-First Search . . . 186
9.4 Discussion and Exercises . . . 189

10 Podatkovne strukture za cela števila **193**
10.1 BinaryTrie: digitalno iskalno drevo . . . 194
10.2 XFastTrie: Iskanje v dvojnem logaritmičnem času . . . 200
10.3 YFastTrie: Dvokratni-Logaritmični Čas SSet . . . 203
10.4 Razprava in vaje . . . 208

11 Iskanje v zunanjem pomnilniku **211**
11.1 Block Store . . . 213
11.2 B-drevesa . . . 213
11.2.1 Searching . . . 215
11.2.2 Addition . . . 218
11.2.3 Removal . . . 223
11.2.4 Amortized Analysis of *B*-Trees . . . 229
11.3 Discussion and Exercises . . . 232

12 External Memory Searching **237**
12.1 The Block Store . . . 239
12.2 B-drevesa . . . 240
12.2.1 Searching . . . 242
12.2.2 Addition . . . 244
12.2.3 Removal . . . 249
12.2.4 Amortized Analysis of *B*-Trees . . . 255
12.3 Discussion and Exercises . . . 258

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. Section **??** 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

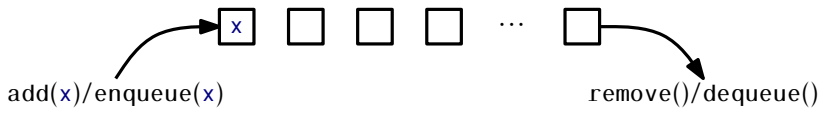
Pri razpravi o podatkovnih strukturah je pomembno poznati razliko med vmesnikom podatkovne strukture in njegovo implementacijo. Vmesnik opisuje kaj podatkovna struktura počne, medtem ko implementacija opisuje kako to počne.

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.

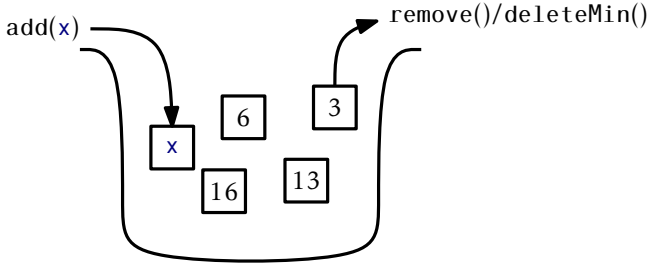
Implementacija podatkovne strukture, po drugi strani, vsebuje notranjo predstavitev podatkovne strukture, vključno z definicijami algoritmov, ki implementirajo operacije, podprte s strani podatkovne strukture. Zato imamo lahko veliko implementacij enega samega vmesnika. Na primer v Chapter 2 bomo videli implementacije vmesnika `seznama` z uporabo polj in v Chapter 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.



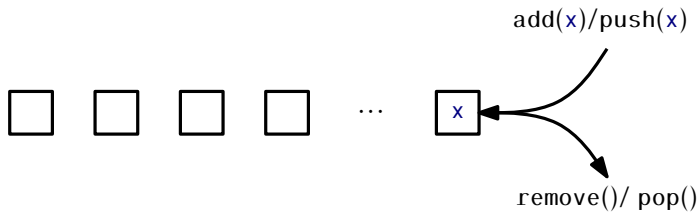
Slika 1.2: Vrsta s prednostjo.

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 Figure 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 Figure 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 Figure 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

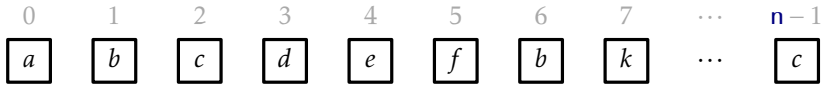
Ta knjiga govori zelo malo o FIFO `vrsti`, `skladu` ali `deque` vmesnikih, ker so vmesniki vključeni z vmesnikom `seznama`. Vmesnik `seznama` vključuje naslednje operacije:

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 Chapter **??** `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

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

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* $\mathtt{n}$.

Kako lahko primerjamo veliko-O notacijo dveh različnih funkcij prikazuje Figure 1.5, ki primerja stopnjo rasti $f_1(\mathtt{n}) = 15\mathtt{n}$ proti $f_2(n) = 2\mathtt{n}\log\mathtt{n}$. 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 $f_1(\mathtt{n})$ večji od $f_2(n)$ pri manjših vrednostih $\mathtt{n}$, velja nasprotno za velike vrednosti $\mathtt{n}$. Po določenem času bo $f_1(\mathtt{n})$ zmagal zaradi stalne povečave širine marže. Analize, ki uporabljajo veliko-O notacijo, kažejo da se bo to zgodilo, ker je $O(\mathtt{n}) \subset O(\mathtt{n}\log\mathtt{n})$.

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

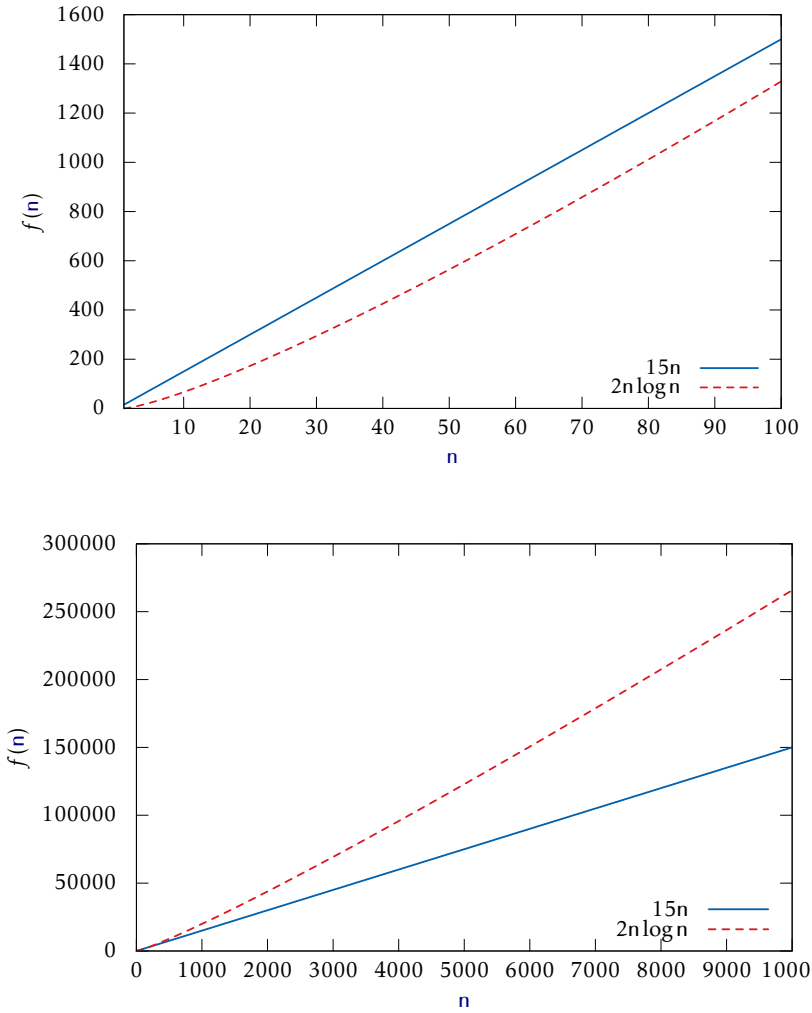
$$\mathrm{E}[X] = \sum_{x \in U} x \cdot \Pr\{X = x\} .$$

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 Section 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 Section **??**. Figure 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 Section **??** 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 [**?**].

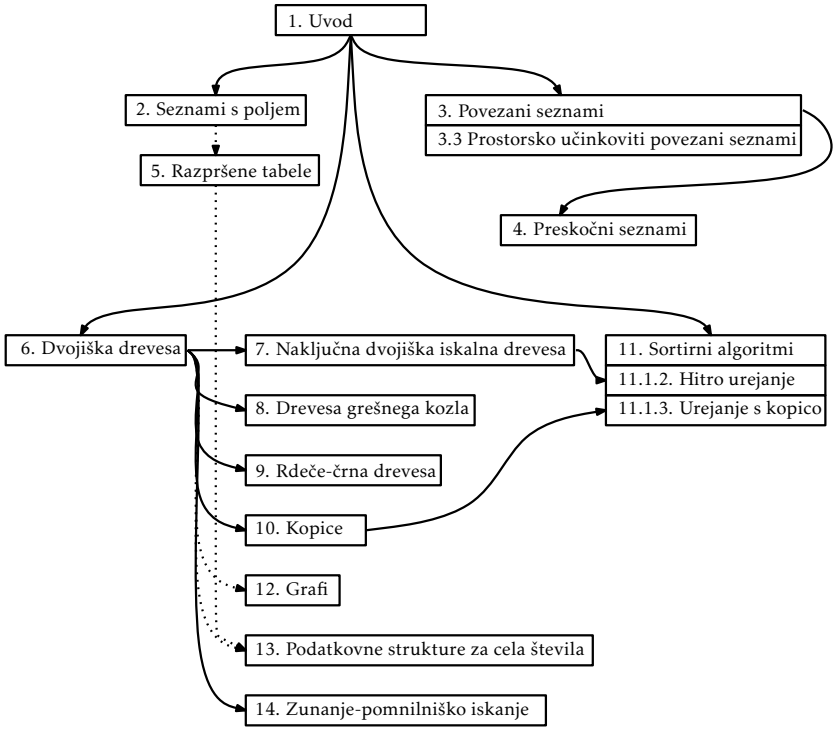
Exercise 1.1. Naloga je sestavljena tako, da bralca seznani s pravilnim izbiranjem najbolj ustrezne podatkovne strukture za dani primer. Če je del

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	$O(1)^{E}$	$O(1)^{A,E}$	§ ??
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`	$O(\log n)^{E}$	$O(\log n)^{E}$	§ 6.2
`ScapegoatTree`	$O(\log n)$	$O(\log n)^{A}$	§ ??
`RedBlackTree`	$O(\log n)$	$O(\log n)$	§ 7.2
`BinaryTrie`^{I}	$O(w)$	$O(w)$	§ 10.1
`XFastTrie`^{I}	$O(\log w)^{A,E}$	$O(w)^{A,E}$	§ 10.2
`YFastTrie`^{I}	$O(\log w)^{A,E}$	$O(\log w)^{A,E}$	§ 10.3

(Priority) `Queue` implementations			
	`findMin()`	`add(x)`/`remove()`	
`BinaryHeap`	$O(1)$	$O(\log n)^{A}$	§ ??
`MeldableHeap`	$O(1)$	$O(\log n)^{E}$	§ 8.2

[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.

Exercise 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.

Exercise 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.

Exercise 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`.

Exercise 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`.

Exercise 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)

Exercise 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

Implementacija 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: Implementacija 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

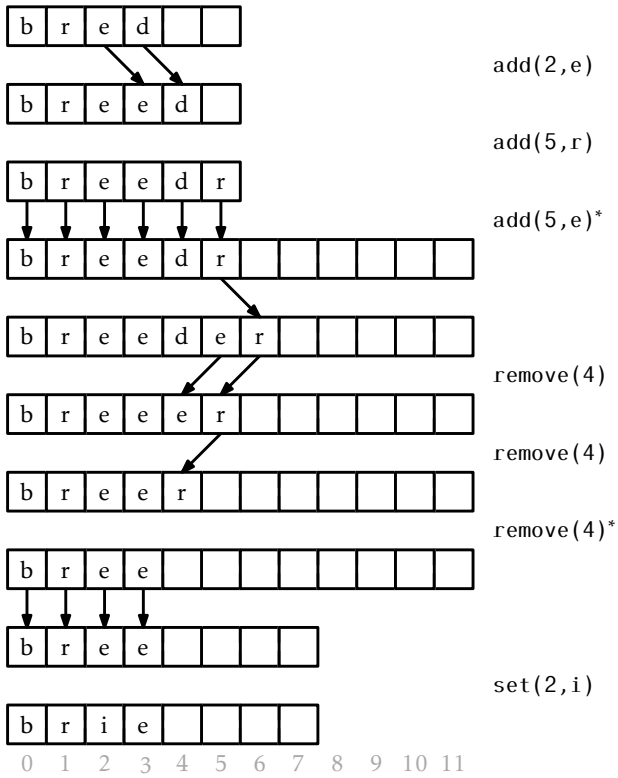
```
array<T> a;
int n;
int size() {
  return n;
}
```

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 Figure 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:

Lemma 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`:

Theorem 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 z 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 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 neskočnem polju. ArrayQueue to simulira z uporabo končnega polja in *modularne aritmetike*. To je vrsta aritmetike, ki jo uporabljamo pri napovedovanju 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 neskončnega polje, ker `i` mod `a.length` vedno vrne vrednost na intervalu $0,\ldots,$ `a.length` -1. Z uporabo modularne aritmetike 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`.

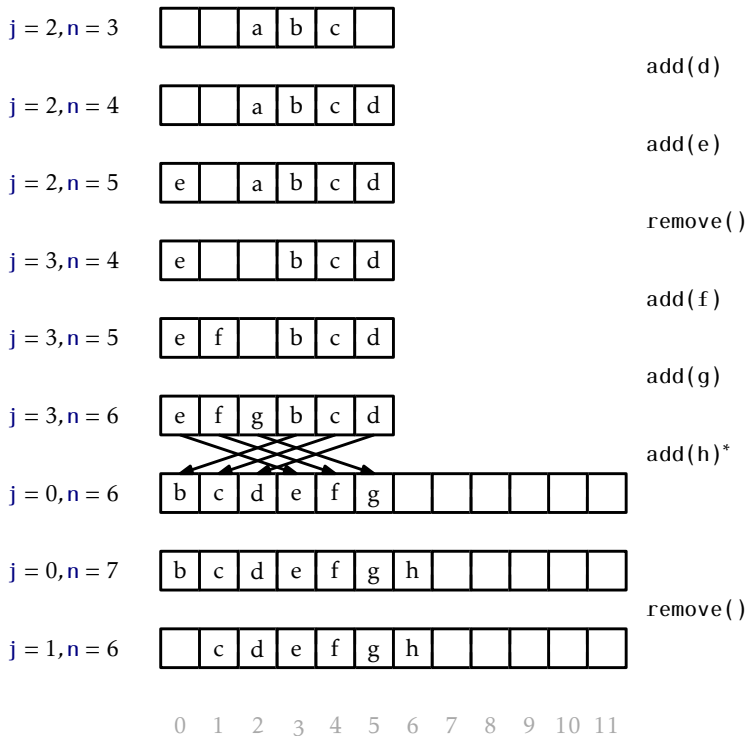
ArrayQueue
```
array<T> a;
int j;
int n;
```

Zaporedje operacij `add(x)` in `remove()` nad `ArrayQueue` je prikazano na Figure 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`:

Theorem 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 skozi vse klice na* `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.

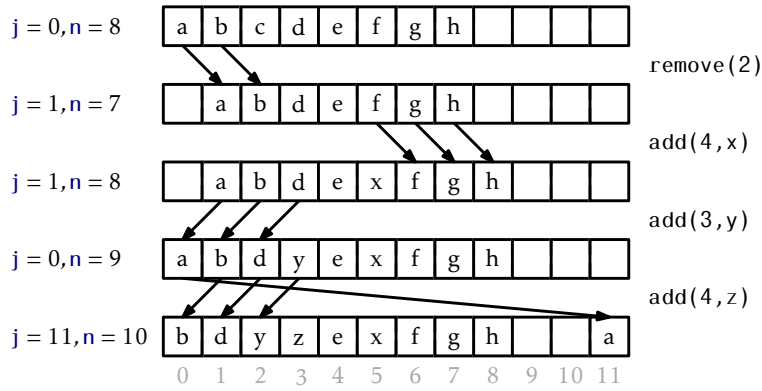
ArrayDeque
```
array<T> a;
int j;
int n;
```

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. Figure 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:

Theorem 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 skozi vse klice na* `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 `ArrayStack`ov. Spomnimo se, da `ArrayStack` deluje hitro, ko operacije nad njim spreminjajo elementa z njegovega konca. `DualArrayDeque` sestoji iz dveh `ArrayStack`ov, enega `spredaj` (`front`) in enega `zadaj` (`back`), s konci nasproti, da to operacije hitre na obeh straneh.

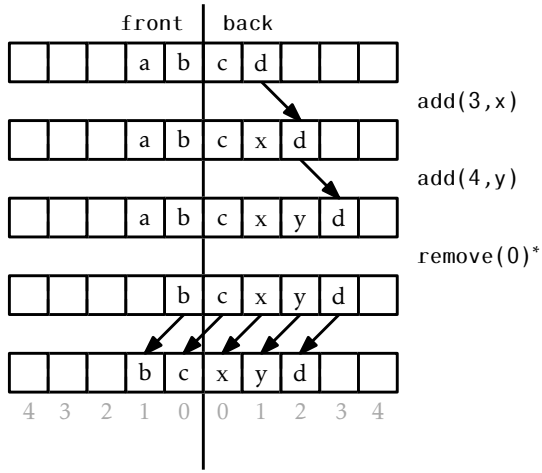
DualArrayDeque
```
ArrayStack<T> front;
ArrayStack<T> back;
```

`DualArrayDeque` ne hrani eksplicitno števila elementov, `n`, ki jih vsebuje. Števila ne rabi 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 normalnem 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 klicom `balance()`, so označene z zvezdico.

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

Opazimo da, če je indeks `i` < `front.size()`, potem 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 Figure 2.4. Operacija `add(i,x)` doda element `spredaj` ali `zadaj`, odvisno od situacije:

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 klicem 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 $3 \cdot \texttt{back.size}() \geq \texttt{front.size}()$.

Naprej bomo analizirali ceno add(i,x), ignorirali bomo ceno klicev balance(). Če je $\texttt{i} < \texttt{front.size}()$, potem je add(i,x) implementirana z klicem front.add(front.size() − i − 1,x). Saj je front ArrayStack in je cena tega

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

Po drugi strani, 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 $\texttt{i} < \texttt{n}/4$. Drugi primer (2.2) se pojavi $\texttt{i} \geq 3\texttt{n}/4$. Ko velja $\texttt{n}/4 \leq \texttt{i} < 3\texttt{n}/4$, ne moremo biti prepričani ali delovanje vpliva na front ali back, v obeh primerih operacija traja $O(\texttt{n}) = O(\texttt{i}) = O(\texttt{n} - \texttt{i})$ časa, saj je $\texttt{i} \geq \texttt{n}/4$ in $\texttt{n} - \texttt{i} > \texttt{n}/4$. Če povzamemo situacijo imamo

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

Čeprav, je čas izvajanja add(i,x), če ignoriramo 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 ter pripadajoče analize.

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 Uravnovešenje

Končno se osredotočimo na metodo balance() izvedeno z metodo add(i,x) in remove(i). Ta metoda 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 ⌊n/2⌋ elementov in ⌈n/2⌉ 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;
  }
}
```

Tukaj je potrebno malo analizirati stvari. Če se metoda balance() uravnoteži, potem premakne $O(n)$ elmentov in za to potrebuje $O(n)$ časa.

To je slabo, saj je metoda `balance()` poklicana z vsakim klicem metod `add(i,x)` in `remove(i)`. Kakorkoli, sledeč dokaz pokaže, da v povprečju metoda `balance()` porabi samo konstantno količino časa na operacijo.

Lemma 2.2. *Če je prazen* `DualArrayDeque` *ustvarjen in če katero zaporedje od* $m \geq 1$ *izvede klice metod* `add(i,x)` *in* `remove(i)`*, potem je skupen zapravljen čas med klici metode* `balance()` $O(m)$.

Dokaz. Pokazali bomo, da če je metoda `balance()` prisiljena premešati elemente potem je število `add(i,x)` in `remove(i)` operacij od kar so bili elementi nazadnje premešani z metodo `balance()` znaša vsaj $n/2 - 1$. Kot v dokazu Lemma 2.1, to zadošča, da lahko dokažemo da je skupen porabljen čas metode `balance()` $O(m)$.

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

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

Zanimiva stvar glede potenciala je to, da klic metode `add(i,x)` ali `remove(i)`, katera ne opravi nobenega uravnovešenja lahko poveča potenical skoraj največ za 1.

Upoštevat je potrebno to, 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 situaciji takoj pred klicem funkcije `balance()`, ki premeša elemente in domnevajte, brez izgube splošnosti, da `balance()` premeša elemente zaradi $3\texttt{front.size()} < \texttt{back.size()}$. To opazimo v tem 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 sčasoma 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, število klicev metode `add(i,x)` ali `remove(i)` od kar je metoda `balance()` nazadnje premešala emelente, znaša najmanj $\Phi_1 - \Phi_0 > \texttt{n}/2 - 1$. To zaključuje dokaz. □

2.5.2 Povzetek

Naslednji izrek povzame lastnosti od `DualArrayDeque`:

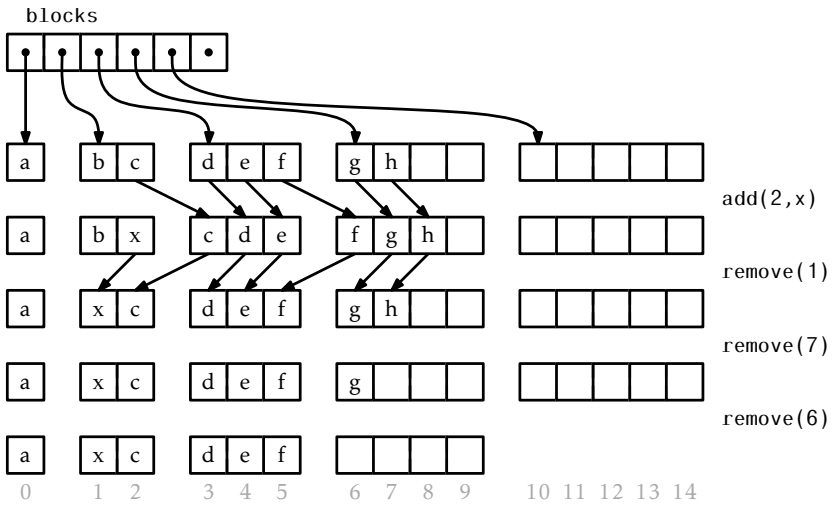
Theorem 2.4. *`DualArrayDeque` implementira `List` vmesnik. Z ignoriranjem cene klicev metod `resize()` in `balance()`, `DualArrayDeque` podpira operacije*

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

Poleg tega, če začnemo z praznim `DualArrayDeque` potem zaporedje od 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`: A Space-Efficient Array Stack

One of the drawbacks of all previous data structures in this chapter is that, because they store their data in one or two arrays and they avoid resizing these arrays too often, the arrays frequently are not very full. For example, immediately after a `resize()` operation on an `ArrayStack`, the backing array `a` is only half full. Even worse, there are times when only 1/3 of `a` contains data.



Slika 2.5: A sequence of `add(i,x)` and `remove(i)` operations on a `RootishArrayStack`. Arrows denote elements being copied.

In this section, we discuss the `RootishArrayStack` data structure, that addresses the problem of wasted space. The `RootishArrayStack` stores n elements using $O(\sqrt{n})$ arrays. In these arrays, at most $O(\sqrt{n})$ array locations are unused at any time. All remaining array locations are used to store data. Therefore, these data structures waste at most $O(\sqrt{n})$ space when storing n elements.

A `RootishArrayStack` stores its elements in a list of r arrays called *blocks* that are numbered $0, 1, \ldots, r-1$. See Figure 2.5. Block b contains $b+1$ elements. Therefore, all r blocks contain a total of

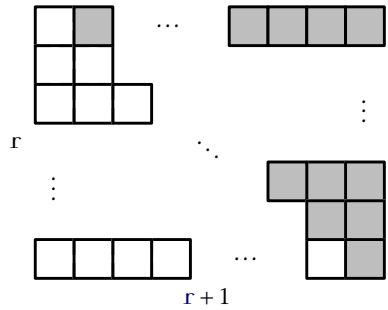
$$1 + 2 + 3 + \cdots + r = r(r+1)/2$$

elements. The above formula can be obtained as shown in Figure 2.6.

`RootishArrayStack`

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

As we might expect, the elements of the list are laid out in order within the blocks. The list element with index 0 is stored in block 0, ele-



Slika 2.6: The number of white squares is $1+2+3+\cdots+r$. The number of shaded squares is the same. Together the white and shaded squares make a rectangle consisting of $r(r+1)$ squares.

ments with list indices 1 and 2 are stored in block 1, elements with list indices 3, 4, and 5 are stored in block 2, and so on. The main problem we have to address is that of determining, given an index i, which block contains i as well as the index corresponding to i within that block.

Determining the index of i within its block turns out to be easy. If index i is in block b, then the number of elements in blocks $0,\ldots,b-1$ is $b(b+1)/2$. Therefore, i is stored at location

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

within block b. Somewhat more challenging is the problem of determining the value of b. The number of elements that have indices less than or equal to i is $i+1$. On the other hand, the number of elements in blocks $0,\ldots,b$ is $(b+1)(b+2)/2$. Therefore, b is the smallest integer such that

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

We can rewrite this equation as

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

The corresponding quadratic equation $b^2+3b-2i=0$ has two solutions: $b=(-3+\sqrt{9+8i})/2$ and $b=(-3-\sqrt{9+8i})/2$. The second solution makes no sense in our application since it always gives a negative value. Therefore, we obtain the solution $b=(-3+\sqrt{9+8i})/2$. In general, this solution

is not an integer, but going back to our inequality, we want the smallest integer b such that $b \geq (-3+\sqrt{9+8i})/2$. This is simply

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

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

With this out of the way, the get(i) and set(i,x) methods are straightforward. We first compute the appropriate block b and the appropriate index j within the block and then perform the appropriate operation:

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;
}
```

If we use any of the data structures in this chapter for representing the blocks list, then get(i) and set(i,x) will each run in constant time.

The add(i,x) method will, by now, look familiar. We first check to see if our data structure is full, by checking if the number of blocks r is such that $r(r+1)/2 = n$. If so, we call grow() to add another block. With this done, we shift elements with indices $i,\ldots,n-1$ to the right by one position to make room for the new element with index 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);
}
```

The grow() method does what we expect. It adds a new block:

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

Ignoring the cost of the grow() operation, the cost of an add(i,x) operation is dominated by the cost of shifting and is therefore $O(1+n-i)$, just like an ArrayStack.

The remove(i) operation is similar to add(i,x). It shifts the elements with indices $i+1,\ldots,n$ left by one position and then, if there is more than one empty block, it calls the shrink() method to remove all but one of the unused blocks:

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--;
    }
}
```

Once again, ignoring the cost of the `shrink()` operation, the cost of a `remove(i)` operation is dominated by the cost of shifting and is therefore $O(n - i)$.

2.6.1 Analysis of Growing and Shrinking

The above analysis of `add(i,x)` and `remove(i)` does not account for the cost of `grow()` and `shrink()`. Note that, unlike the `ArrayStack.resize()` operation, `grow()` and `shrink()` do not copy any data. They only allocate or free an array of size `r`. In some environments, this takes only constant time, while in others, it may require time proportional to `r`.

We note that, immediately after a call to `grow()` or `shrink()`, the situation is clear. The final block is completely empty, and all other blocks are completely full. Another call to `grow()` or `shrink()` will not happen until at least $r - 1$ elements have been added or removed. Therefore, even if `grow()` and `shrink()` take $O(r)$ time, this cost can be amortized over at least $r - 1$ `add(i,x)` or `remove(i)` operations, so that the amortized cost of `grow()` and `shrink()` is $O(1)$ per operation.

2.6.2 Space Usage

Next, we analyze the amount of extra space used by a `RootishArrayStack`. In particular, we want to count any space used by a `RootishArrayStack` that is not an array element currently used to hold a list element. We call all such space *wasted space*.

The `remove(i)` operation ensures that a `RootishArrayStack` never has more than two blocks that are not completely full. The number of blocks, `r`, used by a `RootishArrayStack` that stores `n` elements therefore satisfies

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

Again, using the quadratic equation on this gives

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

The last two blocks have sizes r and $r - 1$, so the space wasted by these two blocks is at most $2r - 1 = O(\sqrt{n})$. If we store the blocks in (for example) an `ArrayList`, then the amount of space wasted by the `List` that stores those r blocks is also $O(r) = O(\sqrt{n})$. The other space needed for storing n and other accounting information is $O(1)$. Therefore, the total amount of wasted space in a `RootishArrayStack` is $O(\sqrt{n})$.

Next, we argue that this space usage is optimal for any data structure that starts out empty and can support the addition of one item at a time. More precisely, we will show that, at some point during the addition of n items, the data structure is wasting an amount of space at least in $\sqrt{n}$ (though it may be only wasted for a moment).

Suppose we start with an empty data structure and we add n items one at a time. At the end of this process, all n items are stored in the structure and distributed among a collection of r memory blocks. If $r \geq \sqrt{n}$, then the data structure must be using r pointers (or references) to keep track of these r blocks, and these pointers are wasted space. On the other hand, if $r < \sqrt{n}$ then, by the pigeonhole principle, some block must have a size of at least $n/r > \sqrt{n}$. Consider the moment at which this block was first allocated. Immediately after it was allocated, this block was empty, and was therefore wasting $\sqrt{n}$ space. Therefore, at some point in time during the insertion of n elements, the data structure was wasting $\sqrt{n}$ space.

2.6.3 Summary

The following theorem summarizes our discussion of the `RootishArrayStack` data structure:

Theorem 2.5. *A `RootishArrayStack` implements the `List` interface. Ignoring the cost of calls to `grow()` and `shrink()`, a `RootishArrayStack` supports the operations*

- `get(i)` *and* `set(i,x)` *in* $O(1)$ *time per operation; and*
- `add(i,x)` *and* `remove(i)` *in* $O(1 + n - i)$ *time per operation.*

Furthermore, beginning with an empty `RootishArrayStack`*, any sequence of* m `add(i,x)` *and* `remove(i)` *operations results in a total of* $O(m)$ *time spent during all calls to* `grow()` *and* `shrink()`.

The space (measured in words)[3] *used by a* `RootishArrayStack` *that stores* n *elements is* $n + O(\sqrt{n})$.

2.6.4 Computing Square Roots

A reader who has had some exposure to models of computation may notice that the `RootishArrayStack`, as described above, does not fit into the usual word-RAM model of computation (Section 1.4) because it requires taking square roots. The square root operation is generally not considered a basic operation and is therefore not usually part of the word-RAM model.

In this section, we show that the square root operation can be implemented efficiently. In particular, we show that for any integer $x \in \{0,\ldots,n\}$, $\lfloor\sqrt{x}\rfloor$ can be computed in constant-time, after $O(\sqrt{n})$ preprocessing that creates two arrays of length $O(\sqrt{n})$. The following lemma shows that we can reduce the problem of computing the square root of x to the square root of a related value x'.

Lemma 2.3. *Let* $x \geq 1$ *and let* $x' = x - a$*, where* $0 \leq a \leq \sqrt{x}$*. Then* $\sqrt{x'} \geq \sqrt{x} - 1$.

Dokaz. It suffices to show that

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

Square both sides of this inequality to get

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

and gather terms to get

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

which is clearly true for any $x \geq 1$. □

Start by restricting the problem a little, and assume that $2^r \leq x < 2^{r+1}$, so that $\lfloor \log x \rfloor = r$, i.e., x is an integer having $r+1$ bits in its binary representation. We can take $x' = x - (x \bmod 2^{\lfloor r/2 \rfloor})$. Now, x' satisfies the

[3] Recall Section 1.4 for a discussion of how memory is measured.

conditions of Lemma 2.3, so $\sqrt{x} - \sqrt{x'} \le 1$. Furthermore, x' has all of its lower-order $\lfloor r/2 \rfloor$ bits equal to 0, so there are only

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

possible values of x'. This means that we can use an array, `sqrttab`, that stores the value of $\lfloor \sqrt{x'} \rfloor$ for each possible value of x'. A little more precisely, we have

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

In this way, `sqrttab`$[i]$ is within 2 of $\sqrt{x}$ for all $x \in \{i2^{\lfloor r/2 \rfloor}, \ldots, (i+1)2^{\lfloor r/2 \rfloor} - 1\}$. Stated another way, the array entry `s = sqrttab[x>>⌊r/2⌋]` is either equal to $\lfloor \sqrt{x} \rfloor$, $\lfloor \sqrt{x} \rfloor - 1$, or $\lfloor \sqrt{x} \rfloor - 2$. From `s` we can determine the value of $\lfloor \sqrt{x} \rfloor$ by incrementing `s` until $(s+1)^2 > x$.

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;
}
```

Now, this only works for $x \in \{2^r, \ldots, 2^{r+1} - 1\}$ and `sqrttab` is a special table that only works for a particular value of $r = \lfloor \log x \rfloor$. To overcome this, we could compute $\lfloor \log n \rfloor$ different `sqrttab` arrays, one for each possible value of $\lfloor \log x \rfloor$. The sizes of these tables form an exponential sequence whose largest value is at most $4\sqrt{n}$, so the total size of all tables is $O(\sqrt{n})$.

However, it turns out that more than one `sqrttab` array is unnecessary; we only need one `sqrttab` array for the value $r = \lfloor \log n \rfloor$. Any value x with $\log x = r' < r$ can be *upgraded* by multiplying x by $2^{r-r'}$ and using the equation

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

The quantity $2^{r-r'}x$ is in the range $\{2^r, \ldots, 2^{r+1} - 1\}$ so we can look up its square root in `sqrttab`. The following code implements this idea to compute $\lfloor \sqrt{x} \rfloor$ for all non-negative integers x in the range $\{0, \ldots, 2^{30} - 1\}$ using an array, `sqrttab`, of size 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;
}
```

Something we have taken for granted thus far is the question of how to compute $r' = \lfloor \log x \rfloor$. Again, this is a problem that can be solved with an array, `logtab`, of size $2^{r/2}$. In this case, the code is particularly simple, since $\lfloor \log x \rfloor$ is just the index of the most significant 1 bit in the binary representation of `x`. This means that, for $x > 2^{r/2}$, we can right-shift the bits of `x` by $r/2$ positions before using it as an index into `logtab`. The following code does this using an array `logtab` of size 2^{16} to compute $\lfloor \log x \rfloor$ for all `x` in the range $\{1, \ldots, 2^{32} - 1\}$.

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

Finally, for completeness, we include the following code that initializes `logtab` and `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;
  }
}
```

To summarize, the computations done by the i2b(i) method can be implemented in constant time on the word-RAM using $O(\sqrt{n})$ extra memory to store the sqrttab and logtab arrays. These arrays can be rebuilt when n increases or decreases by a factor of two, and the cost of this rebuilding can be amortized over the number of add(i,x) and remove(i) operations that caused the change in n in the same way that the cost of resize() is analyzed in the ArrayStack implementation.

2.7 Discussion and Exercises

Most of the data structures described in this chapter are folklore. They can be found in implementations dating back over 30 years. For example, implementations of stacks, queues, and deques, which generalize easily to the ArrayStack, ArrayQueue and ArrayDeque structures described here, are discussed by Knuth [**?**, Section 2.2.2].

Brodnik *et al.* [**?**] seem to have been the first to describe the RootishArrayStack and prove a $\sqrt{n}$ lower-bound like that in Section 2.6.2. They also present a different structure that uses a more sophisticated choice of block sizes in order to avoid computing square roots in the i2b(i) method. Within their scheme, the block containing i is block $\lfloor \log(i+1) \rfloor$, which is simply the index of the leading 1 bit in the binary representation of $i+1$. Some computer architectures provide an instruction for computing the index of the leading 1-bit in an integer.

A structure related to the RootishArrayStack is the two-level *tiered-vector* of Goodrich and Kloss [**?**]. This structure supports the get(i,x) and set(i,x) operations in constant time and add(i,x) and remove(i) in $O(\sqrt{n})$ time. These running times are similar to what can be achieved with the more careful implementation of a RootishArrayStack discussed in Exercise 2.10.

Exercise 2.1. The List method addAll(i,c) inserts all elements of the Collection c into the list at position i. (The add(i,x) method is a special case where c = {x}.) Explain why, for the data structures in this chapter, it is not efficient to implement addAll(i,c) by repeated calls to add(i,x). Design and implement a more efficient implementation.

Exercise 2.2. Design and implement a *RandomQueue*. This is an implementation of the Queue interface in which the remove() operation removes an element that is chosen uniformly at random among all the elements currently in the queue. (Think of a RandomQueue as a bag in which we can add elements or reach in and blindly remove some random element.) The add(x) and remove() operations in a RandomQueue should run in constant time per operation.

Exercise 2.3. Design and implement a Treque (triple-ended queue). This is a List implementation in which get(i) and set(i,x) run in constant time and add(i,x) and remove(i) run in time

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

In other words, modifications are fast if they are near either end or near the middle of the list.

Exercise 2.4. Implement a method rotate(a,r) that "rotates" the array a so that a[i] moves to a[(i+r) mod a.length], for all $i \in \{0,\ldots,$a.length$\}$.

Exercise 2.5. Implement a method rotate(r) that "rotates" a List so that list item i becomes list item (i+r) mod n. When run on an ArrayDeque, or a DualArrayDeque, rotate(r) should run in $O(1+\min\{r, n-r\})$ time.

Exercise 2.6. Modify the ArrayDeque implementation so that the shifting done by add(i,x), remove(i), and resize() is done using the faster System.arraycopy(s method.

Exercise 2.7. Modify the ArrayDeque implementation so that it does not use the % operator (which is expensive on some systems). Instead, it should make use of the fact that, if a.length is a power of 2, then

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

(Here, & is the bitwise-and operator.)

Exercise 2.8. Design and implement a variant of ArrayDeque that does not do any modular arithmetic at all. Instead, all the data sits in a consecutive block, in order, inside an array. When the data overruns the

beginning or the end of this array, a modified `rebuild()` operation is performed. The amortized cost of all operations should be the same as in an `ArrayDeque`.
Hint: Getting this to work is really all about how you implement the `rebuild()` operation. You would like `rebuild()` to put the data structure into a state where the data cannot run off either end until at least $n/2$ operations have been performed.

Test the performance of your implementation against the `ArrayDeque`. Optimize your implementation (by using `System.arraycopy(a,i,b,i,n)`) and see if you can get it to outperform the `ArrayDeque` implementation.

Exercise 2.9. Design and implement a version of a `RootishArrayStack` that has only $O(\sqrt{n})$ wasted space, but that can perform `add(i,x)` and `remove(i,x)` operations in $O(1+\min\{i, n-i\})$ time.

Exercise 2.10. Design and implement a version of a `RootishArrayStack` that has only $O(\sqrt{n})$ wasted space, but that can perform `add(i,x)` and `remove(i,x)` operations in $O(1+\min\{\sqrt{n}, n-i\})$ time. (For an idea on how to do this, see Section 3.3.)

Exercise 2.11. Design and implement a version of a `RootishArrayStack` that has only $O(\sqrt{n})$ wasted space, but that can perform `add(i,x)` and `remove(i,x)` operations in $O(1+\min\{i, \sqrt{n}, n-i\})$ time. (See Section 3.3 for ideas on how to achieve this.)

Exercise 2.12. Design and implement a `CubishArrayStack`. This three level structure implements the `List` interface using $O(n^{2/3})$ wasted space. In this structure, `get(i)` and `set(i,x)` take constant time; while `add(i,x)` and `remove(i)` take $O(n^{1/3})$ amortized time.

Poglavje 3

Povezani seznam

V tem poglavju nadaljujemo z implementacijo Seznama, 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 povezana zaporedoma 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 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 seznam, element po element, 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 Vozlisc. 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;
  }
};
```

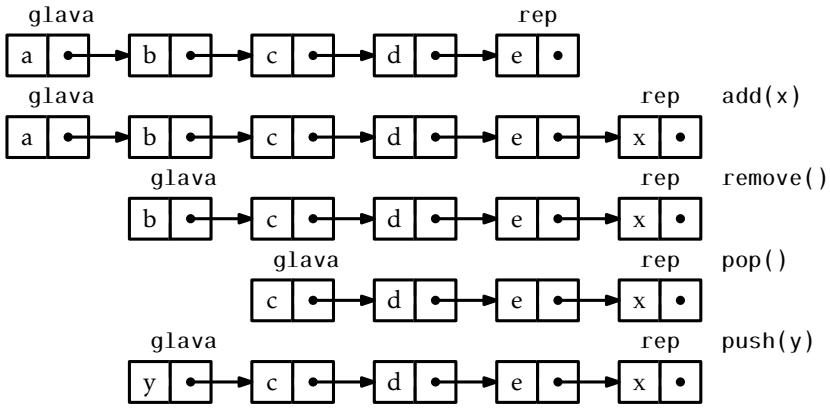
Za boljšo učinkovitost delovanja SLList uporablja spremnljivki head (glava) in tail (rep) za beleženje prvega ter zadnjega vozlišča. Za beleženje dolžine seznama, pa hrani še celoštevilsko spremenljivko n:

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

Zaporedje ukazov Sklada in Vrste nad enostransko povezanim seznamom je prikazana na Figure 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:

Theorem 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.

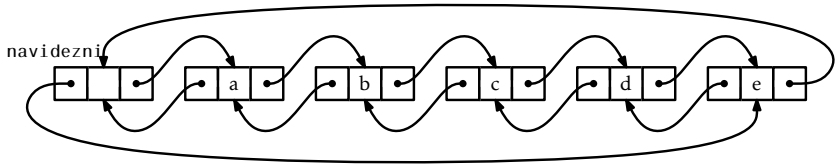
3.2 DLList: Obojestransko povezan seznam

(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, ki ne vsebuje nobenih podatkov, ampak deluje kot ograda,tako da ni posebnih vozlišč; vsako vozlišče ima tako next kot prev, z dummy, ki deluje kot navidezno vozlišče, ki sledi zadnjemu vozlišču v seznamu in je predhodnik prvega vozlišča v seznamu. Na ta način so vozlišča v seznamu obojestransko povezana v cikel, kot je prikazano naFigure 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;
}
```

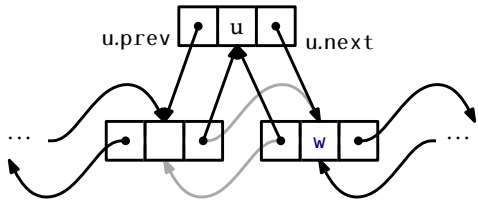
Čas izvajanja teh operacij je določen z strani časa, ki potrebujemo, da najdemo `i`-to vozlišče in je zato $O(1+\min\{i, n-i\})$.

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 Figure 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;
```



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

```
  n++;
  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`:

Theorem 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 Chapter 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 Section 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 Section 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:

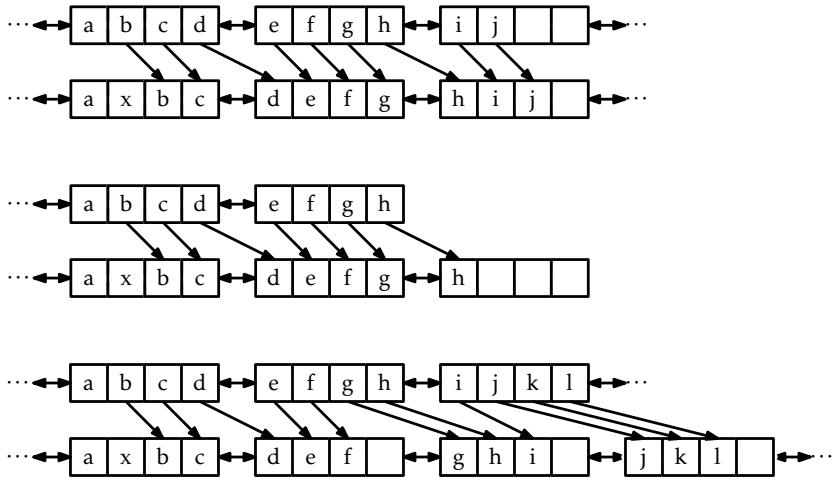
SEList

```
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 Figure 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 i ja in izvajanja vstavljanja elementa x $O(b+\min\{i, n-i\}/b)$.

3.3.4 Odstranjevanje elementov

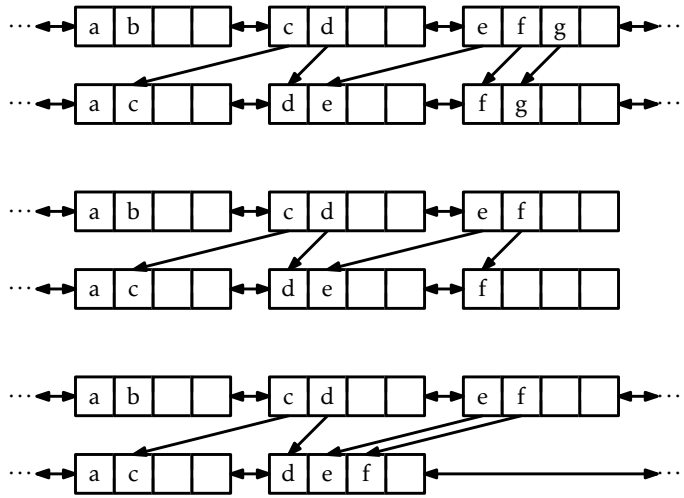
Odstranjevanje elementa iz podatkovne strukture SEList je podobno dodajanju elementov vanjo. Najprej lociramo vozlišče u, ki vsebuje element z indeksom i. Zdaj moramo biti pripravljeni na primer, ko elementa ne moremo zbrisati iz vozlišča u, ne da bi u-jev blok postal manjši od $b-1$.

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 (Figure 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)`.

Lemma 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:

Theorem 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 Section 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 [**?**] .

Exercise 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()?

Exercise 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.

Exercise 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.

Exercise 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.

Exercise 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.

Exercise 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.

Exercise 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.

Exercise 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.

Exercise 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.

Exercise 3.10. Napišite metodo dvojno-povezanega seznama `DLList absorb(12)`, ki za vhodni parameter prejme dvojno-povezan seznam `DLList` 12, ter sprazni njegovo vsebino in jo pripne na konec svojega seznama. Naprimer, če 11 vsebuje a,b,c in 12 vsebuje d,e,f, po klicu 11.`absorbe(12)` 11 vesbuje a,b,c,d,e,f, 12 pa bo prazen.

Exercise 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 11 vsebuje a,b,c,d,e,f, potem bo po klicu 11.`deal()` vseboval a,c,e, metoda pa bo vrnila seznam , ki vsebuje elemente b,d,f.

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

Exercise 3.13. V tej vaji boste implementirali urejanje pod. strukture `DLList` z zlivanjem, kot je opisano v poglavju Section **??**.

1. Napišite metodo pod. strukture `DLList takeFirst(12)`, ki odstrani prvo vozlišče iz 12 ter ga doda na konec seznama, nad katerim je bila metoda klicana. Metoda je enakovredna klicu `add(size(),12.remove(0))`, vendar pri tem ne ustvari novega vozlišča.

2. Napišite statično metodo pod. strukture `DLList merge(11,12)`, ki kot argument dobi dva urejena seznama 11 in 12, ju združi ter vrne nov urejen seznam. Seznama 11 ter 12 se v metodi izpraznita. Naprimer, če 11 vsebuje a,c,d in 12 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.

Exercise 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.

Exercise 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.

Exercise 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`).

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

Exercise 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.

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

Poglavje 4

Preskočni seznami

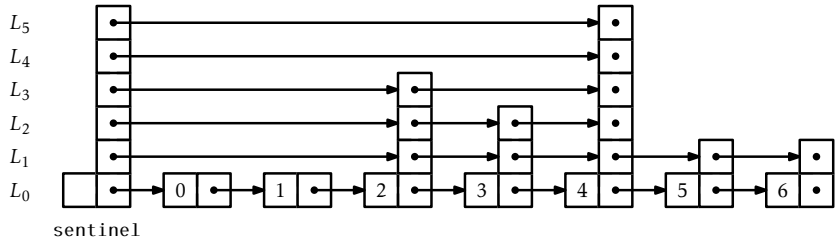
V tem poglavju bomo govorili o lepi podatkovni strukturi: preskočnem seznamu, ki ima veliko možnosti uporabe. Z uporabo preskočnega seznama lahko implementiramo `List`, ki ima $O(\log n)$ časovno implementacijo `get(i)`, `set(i,x)`, `add(i,x)`, and `remove(i)`. Prav tako lahko implementiramo `SSet`, 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 namenskega generatorja.

4.1 Osnovna struktura

Konceptualno je preskočni seznam sekvenca 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 Figure 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.

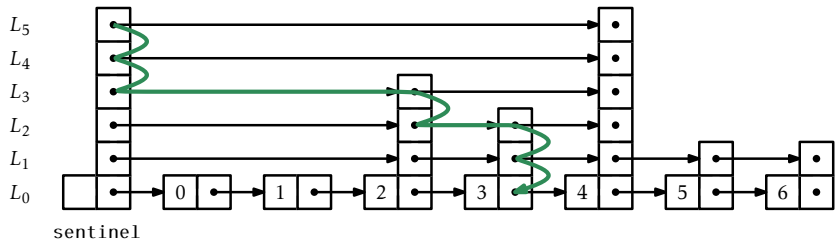
Na koncu vsakega seznama je posebno vozlišče, imanovano *stražar* od stražarja v L_h do vsakega vozlišča v L_0. Narediti pot iskanja za posamezno vozlišče uje preprosto (glej Figure 4.2) : Začnemo v zgornjem levem kotu preskočnega seznama (stražar je v L_h) in se premikamo desno toliko časa, dokler ne 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 izvedemo 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 Section **??**, nam pokaže, da je pot iskanja dokaj kratka:

Lemma 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 kazalcev next, kjer u.next[i] kaže na naslednika u-ja v seznamu L_i. Na



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

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 tega poglavja 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 temi dvemi strukturami 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 Lemma 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 binarnem 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 $2^{32} = 4294967296$.

Za implementirati metodo 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):

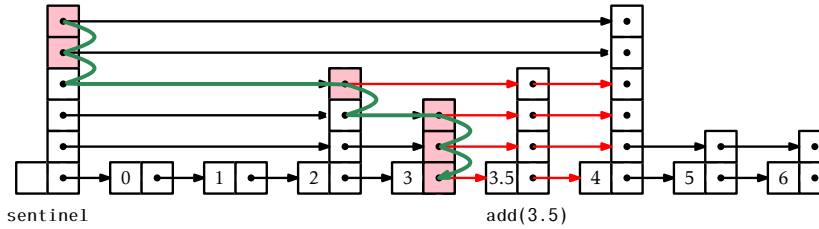
SkiplistSSet

```
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:

SkiplistSSet

```
bool remove(T x) {
  bool removed = false;
```

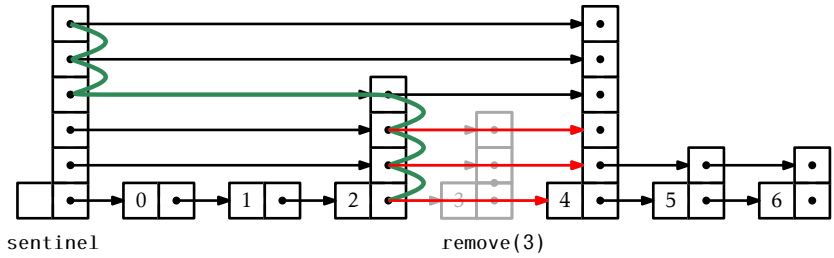


Slika 4.3: Dodajanje vozlišča 3.5 v preskočni seznam. Vozlišča shranjena v `sklad` so označena.

```
  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;
}
```

4.2.1 Povzetek

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



Slika 4.4: Brisanje vozlišča 3 iz preskočnega seznama.

Theorem 4.1. *`SkiplistSSet` je uporabljen za implementacijo `SSet` vmesnika. `SkiplistSSet` opravi operacije* `add(x)` *(dodaj),* `remove(x)` *(odstrani) in* `find(x)` *(najdi) v* $O(\log n)$ *pričakovanega časa za operacijo.*

4.2.2 Summary

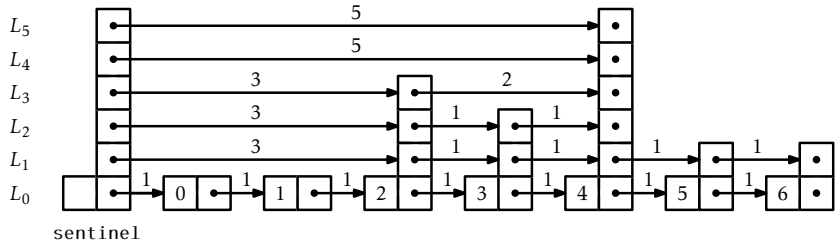
Sledeči teorem povzema uporabnost preskočnega seznama, ko ga uporabljamo pri implementaciji urejenih sklopov:

Theorem 4.2. *`SkiplistSSet` implementira `SSet` vmesnik. A `SkiplistSSet` vsebuje operacije* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log n)$ *pričakovani čas za izvedbo operacije.*

4.3 `SkiplistList`: Učinkovit naključni dostop `List`

A `SkiplistList` implementira `List` vmesnik s pomočjo(uporabo) preskočnega seznama. V `SkiplistList`, L_0 vsebuje elemente seznama po vrstnem redu pojavljanja elementov. Po drugi strani `SkiplistSSet`, elemente lahko dodajamo, brišemo ali do njih dostopamo v $O(\log n)$ časa.

Za doseganje tega, potrebujemo možnost iskanja poti `i`th elementa v L_0. Najlažji način je definirati notacijo the *length* od roba nekega seznama, L_r. Vsak rob seznama definiramo L_0 kot 1. Dolžina robu, `e`, v L_r, $r > 0$, je definiran kot vsota dolžin robov pod njim `e` v L_{r-1}. Ekvivalenčno, dolžina `e` je število robov v L_0 spodaj `e`. Poglej Figure 4.5 za primer preskočnega seznama z dolžino njegovih robov. Posledica shra-



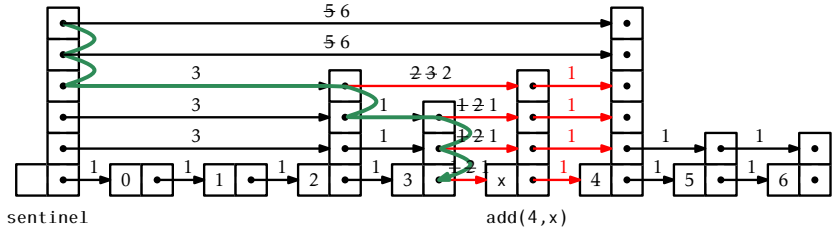
Slika 4.5: The lengths of the edges in a skiplist.

njevanja robov preskočnega seznama v nizih, lahko dolžino shranjujemo na enak način:

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

Povzetek te opredelitve dolžin je da smo trenutno na vozlišču, ki se nahaja na poziciji `j` v L_0 in sledimo robu dolžine ℓ, nato se premaknemo na vozlišče čigar pozicija v L_0, je $\texttt{j} + \ell$. Po takem postopku, medtem ko iščemo iskalno pot lahko ohranjamo vrednost pozicije, `j`, trenutnega vozlišča v L_0. Medtem ko na vozlišču, `u`, v $L_\texttt{r}$, gremo desno če `j` plus dolžina roba `u.next[r]` je manj kakor `i`. V nasprotnem primeru, gremo navzdol v $L_{\texttt{r}-1}$.

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];
    }
```



Slika 4.6: Adding an element to a SkiplistList.

```
    r--;
  }
  return u;
}
```

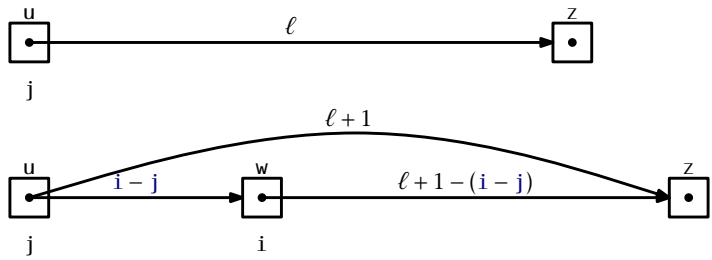
SkiplistList

```
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`th vozlišča v L_0, se operacije izvedejo v $O(\log n)$ časa.

Dodajanje elementa v `SkiplistList` na pozicijo, `i`, je enostavno. Za razliko dodajanje v `SkiplistSSet`, smo prepričani da bo vozlišče Dejansko dodano, zato lahko hkrati dodajamo in iščemo lokacijo za novo vozlišče. Najprej izberemo višino, `k`, novo dodanega vozlišča `w`, nato sledimo iskalni poti `i`. Vsakič ko se iskalna pot premakne navzdol od L_r z $r \le k$, uporabimo spoj `w` v L_r. Dodatno moremo biti pozorni da se dolžina robov pravilno osvežuje. Poglej Figure 4.6.

Pozorni moremo biti, da vsakič ko se iskalna pot premakne za eno vozščišče navzdol, `u`, v L_r, se dolžina roba `u.next[r]` poveča za ena, ker



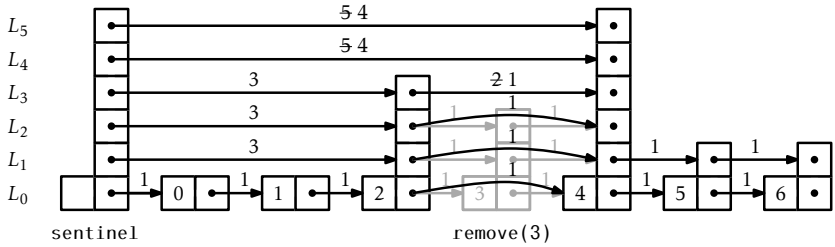
Slika 4.7: Updating the lengths of edges while splicing a node w into a skiplist.

dodajamo element pod rob na poziciji i. Spojimo vozlišče w med vozlišča, u in z, deluje kakor prikazano v Figure 4.7. Medtem Ko sledimo iskalni poti, tudi shranjujemo pozicijo j, od u v L_0. Zato, vemo da je dolžina roba od u do w velikosti i − j. Sklepamo lahko da je razdalja roba od w do z iz dolžine, ℓ, od roba u do z. Potemtakem, lahko spojimo v w in osvežimo dolžine od robov v konstantnem času.

Postopek izgleda veliko bolj zakompliciran 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];
```



Slika 4.8: Removing an element from a SkiplistList.

```
    }
    u->length[r]++; // to account for new node in list 0
    if (r <= k) {
      w->next[r] = u->next[r];
      u->next[r] = w;
      w->length[r] = u->length[r] - (i - j);
      u->length[r] = i - j;
    }
    r--;
  }
  n++;
  return u;
}
```

Do sedaj bi vam morala biti implementacija remove(i) operacije v SkiplistList jasna. Iščemo iskalno pot vozlišča na poziciji i. Vsakič ko se iskalna pot zmanjša za ena od vozlišča u, na ravni r zmanjšamo radaljo od roba, tako da pustimo u na tistem nivoju. Pregledovati 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 Figure 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 Summary

Naslednji teorem povzema učinkovitost podatkovne strukture `SkiplistList`:

Theorem 4.3. *`SkiplistList` implementira `List` -ov vmesnik. `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.

Lemma 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 nav-

zgor. 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:

Lemma 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

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

□

Lemma 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

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

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

Višina, h, preskočnega seznama je

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

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

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

Zato imamo

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

□

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

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

Lemma 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. footnoteBodite 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. Lemma **??** 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 Lemma 4.4. Naj bo S dolžina iskalne poti nekega vozlišča, u, v preskočnem seznamu in naj bo h višina preskočnega seznama. Sledi

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

$$\begin{aligned}
&\leq \mathrm{E}[h] + \sum_{r=0}^{\lfloor \log n \rfloor} 1 + \sum_{r=0}^{\infty} 1/2^r \\
&\leq \mathrm{E}[h] + \log n + 3 \\
&\leq 2 \log n + 5 \ .
\end{aligned}$$

□

Sledeči teorem povzema rezultat sekcije:

Theorem 4.4. *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 Discussion and Exercises

Poglavje 5

Dvojiška drevesa

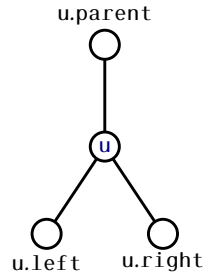
To poglavje predstavlja eno najbolj temeljnih struktur v računalništvu: dvojiška drevesa. Uporaba besede *drevo* prihaja iz dejstva, da ko jih rišemo, je risba podobna drevesom iz gozda. Obstaja veliko načinov definiranja binarnega drevesa. Matematično je *binarno drevo* povezan, neusmerjen, končni graf brez ciklov.

Za večino aplikacij v računalništvu, so binarna 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 . Vsa druga vozlišča, ki mejijo na u imenujemo *otrok* od u. Večina dvojiških dreves, ki nas zanimajo, so *urejena* in tako lahko ločimo med *levi otrok* in *desni otrok* od u.

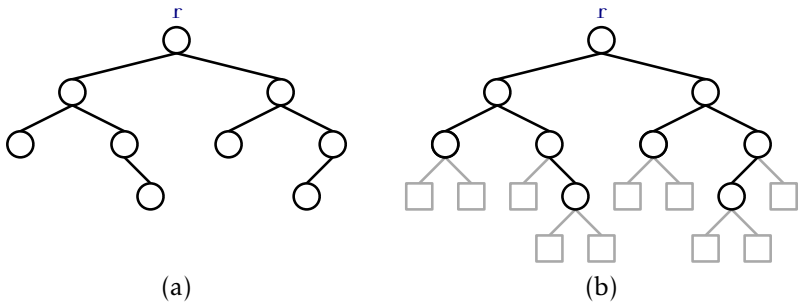
V ilustraciji so dvojiška drevesa običajno sestavljena iz korena navzdol. Koren je na vrhu slike, ki ima levega in desnega otroka. Levi otrok je na levi strani, desni pa na desni strani (Figure 5.1). Na primer Figure 5.2. Kaže binarno drevo z devetimi vozlišči.

Ker so dvojiška drevesa tako pomembna, so za njih razvili določeno terminologijo: *globina* vozliščea, u, je v binarnem 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 *potomec* od w. *poddrevo* od vozlišča u je binarno drevo, ki ima korenine v u in vsebuje vse potomce od u. *višina* vozlišča u, je dolžina najdaljše poti od u do enega od njenih potomcev. *višina* od drevesa je višina njegovega korena. Vozlišče u, je *list* če nima nobenega otroka.

Včasih mislimo, da so drevesa utrjena z *zunanjimi vozlišči*. Vsako



Slika 5.1: Starš, levi otrok, desni otrok vozlišča u v BinaryTree.



Slika 5.2: Binarno drevo (a) devet vozlišč in (b) deset zunanjih vozlišč.

vozlišče, ki nima levega otroka ima zunanje vozlišče kot svojega levega otroka in ustrezno vsako vozlišče, ki nima pravega otroka ima zunanjo vozlišče kot svojega pravega otroka (glejte Figure 5.2.b). Z indukcijo lahko enostavno preverimo, da binarno drevo z $n \geq 1$ pravimi vozlišči ima $n+1$ zunanjih vozlišč.

5.1 BinaryTree: Osnovno Binarno Drevo

Najenostavnejši način, predstavitve vozlišča u, v binarnem drevesu je izrecno shranjevanje (največ treh) sosedeov 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`.

Binarno 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;
}
```

5.1.1 Rekurzivni algoritmi

Z uporabo rekurzivnih algoritmov je izračun o binarnih drevesih enostaven. Na primer, za izračun velikosti (število vozlišč) binarnega 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));
}
```

5.1.2 Obiskovanje Binarnega drevesa

Prejšnja algoritma iz prejšnjega odseka uporabljata rekurzijo, za obisk vseh vozlišč v binarnem drevesu. Vsak od njih obišče vozlišča binarnega 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 rekurzijska je podana z največjo globino vozlišča v dvojiškem drevesu, tj,

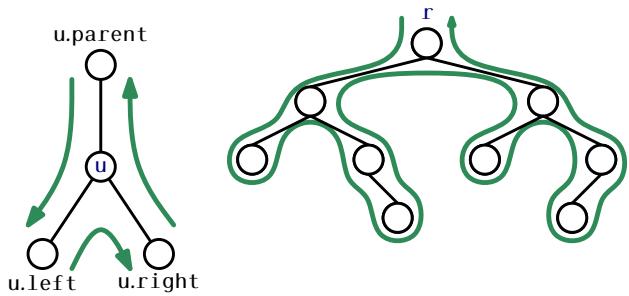
višina drevesa. Če je višina drevesa zelo velika, potem lahko taka rekurzija porabi veliko več pomnilnika na skladu, kot ga je na voljo.

Za obhod binarnega drevesa brez rekurzije, lahko uporabimo algoritem, ki se zanaša na to, da ve iz kje je prišel in kam bo odšel. Glej Figure 5.3. 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 to 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;
```

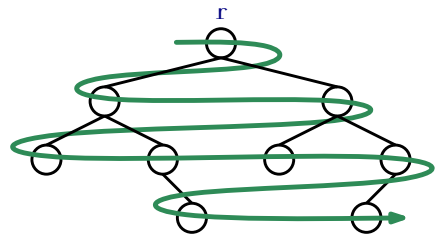


Slika 5.3: Trije primeri, ki se pojavijo na vozlišču u kadar obhodimo binarna drevesa,ki niso rekurzivna

```
      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;
  }
  return n;
}
```

V nekaterih implementacijah binarnih 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 Figure 5.4). 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,



Slika 5.4: Med prvi-v-širino obhodu, so vozlišča v binarnem drevesu obiskana po principu stopnja-po-stopnjo in levo-proti-desni za vsako stopnjo.

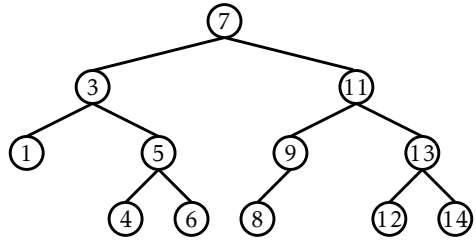
in dodamo `u.left` in `u.right` (če niso `nil`) v `q`:

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);
  }
}
```

5.2 BinarySearchTree: Neuravnoteženo binarno iskalno drevo

`BinarySearchTree` je posebna oblika binarnega drevesa, pri katerem vsako vozlišče `u` hrani tudi podatek `u.x` iz nekega skupnega vrstnega reda. Podatki binarnega iskalnega drevesa upoštevajo *lastnost binarnih 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 Figure 5.5.



Slika 5.5: Binarno iskalno drevo.

5.2.1 Iskanje

Lastnost binarnega iskalnega drevesa je zelo uporabna, ker nam omogoča hitro iskanje vrednosti `x` v binarnem 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` = `nil` (prazen). V prvem primeru smo našli `x`. V drugem pa sklenemo, da `x` ni v binarnem 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;
}
```

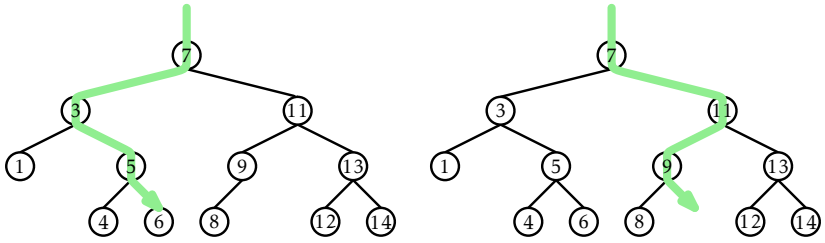
V Figure 5.6 sta prikazana dva primera iskanj v binarnem iskalnem drevesu. Drugi primer prikazuje, da tudi če ne najdemo x v drevesu, vseeno pridobimo nekaj pomembnih informacij. Če pogledamo zadnje vozlišče u pri katerem se je zgodila Možnost 1, vidimo, da je u.x najmanjša vrednost v drevesu, ki je 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;
}
```

5.2.2 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 5.6: Primer (a) uspešnega iskanja (za 6) ter (b) neuspešnega iskanja (za 10) v binarnem iskalnem drevesu.

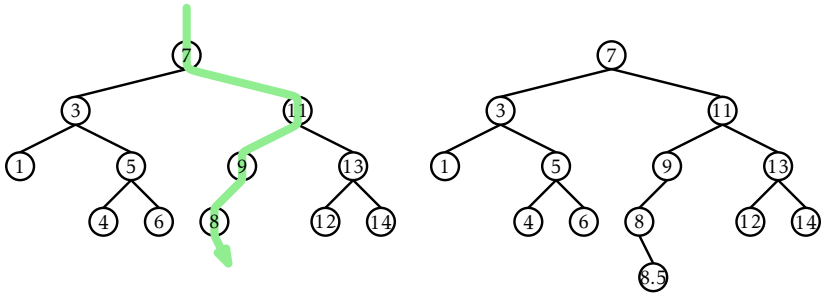
p, je odvisno od rezultata primerjave med x ter p.x.

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 5.7: Vstavljanje vrednosti 8.5 v binarno 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 Figure 5.7. 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`.

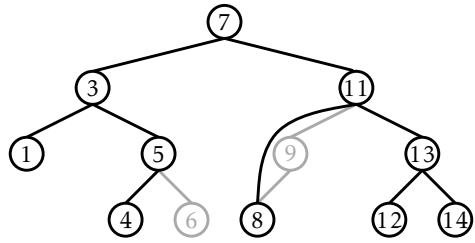
5.2.3 Brisanje

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 Figure 5.8):

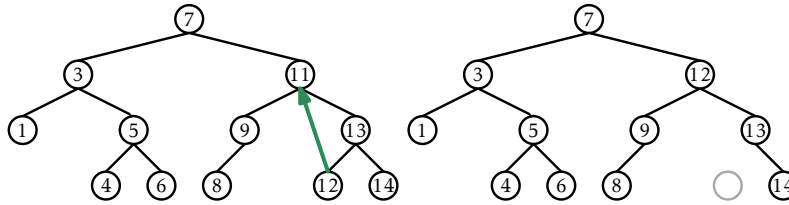
BinarySearchTree

```
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 binarnega bskalnega 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



Slika 5.8: Brisanje lista (6) ali vozlišča z enim otrokom (9) je preprosto.



Slika 5.9: 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.

levega otroka (glej Figure 5.9).

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;
  }
}
```

5.2.4 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`:

Theorem 5.1. *`BinarySearchTree` implementira `SSet` vmesnik ter podpira funkcije* `add(x)`, `remove(x)` *ter* `find(x)` *v* $O(n)$ *časa na operacijo.*

Theorem 5.1 se slabo primerja z Theorem 4.2, 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 Figure 5.5, 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 Chapter 6 pokažemo, kako lahko dosežemo operacije z *pričakovanim* časom $O(\log n)$ s pomočjo naključnosti. V Chapter **??** pokažemo, kako dosežemo operacije z *amortiziranim* časom $O(\log n)$ s pomočjo delnih obnovitvenih operacij. V Chapter 7 pokažemo, kako dosežemo operacije z *najslabšim* časom $O(\log n)$ s pomočjo simulacije dreves, ki niso binarna: eno v katerem imajo vozlišča lahko do štiri otroke.

5.3 `BinaryTree`: Razprava in vaje

TODO :: bostjan TODO :: sandi **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 binarnem drevesu. **Vaja 6.6** Binarno drevo je uravnoteženo, če za vsako vozlišče u... TODO :: ziga

Poglavje 6

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.

6.1 Naključna iskalna binarna drevesa

Premislimo o dveh binarnih iskalnih drevesih, ki sta prikazani na Figure 6.1, 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

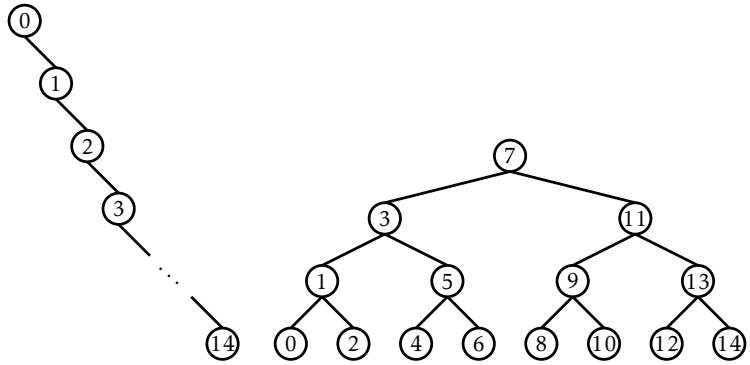
$$\langle 7, 3, 11, 1, 5, 9, 13, 0, 2, 4, 6, 8, 10, 12, 14 \rangle \ .$$

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 6.1: Dva binarna iskana drevesa vsebujeta cela števila $0,\ldots,14$.

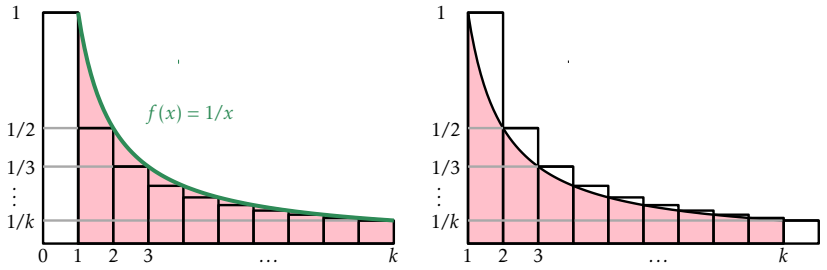
Dejstvo je, da obstaja $21,964,800$ dodatnih zaporedij, ki lahko ustvarijo drevo na desni strani in samo eno zaporedje, ki lahko ustvari drevo na levi strani.

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 Figure 6.1) tako lahko dobimo zelo neuravnoteženo drevo (na levi strani Figure 6.1).

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 6.2: 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 Figure 6.2 za grafično razlago.)

Lemma 6.1. *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 Lemma 6.1 v naslednjem poglavju. Za zdaj, upoštevajmo kaj nam povedo oba dela Lemma 6.1. 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.

6.1.1 Dokaz Lemma 6.1

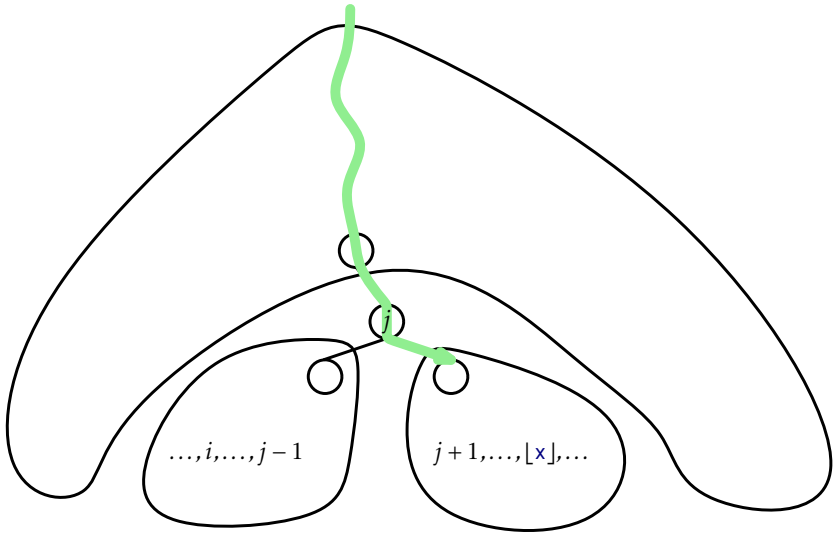
Ključna ugotovitev pri dokazovanju Lemma 6.1 je naslednja: Iskana pot za vrednost x v odprtem intervalu $(-1, n)$ v naključnem binarnem iskalnem drevesu, T, vsebuje vozlišče s ključem $i < x$ če, in samo če je naključna permutacija uporabljena za ustvarjanje T, i preden se pojavi katerakoli od $\{i+1, i+2, \ldots, \lfloor x \rfloor\}$.

Da bi to videli, se nanašamo Figure 6.3 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 6.3: 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.

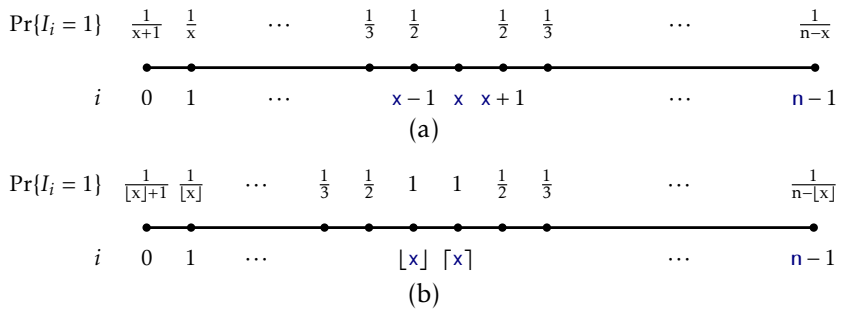
S tem opazovanjem, dokaz za Lemma 6.1 vključuje nekaj preprostih izračunov z harmonskimi števili:

Dokaz Lemma 6.1. 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 Figure 6.4.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 6.4: 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 Figure 6.4.b). □

6.1.2 Povzetek

Spodnji teorem povzame učinkovitost naključnega binarnega iskalnega drevesa:

Theorem 6.1. *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 Theorem 6.1 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.

6.2 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 Lemma 6.1 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*:

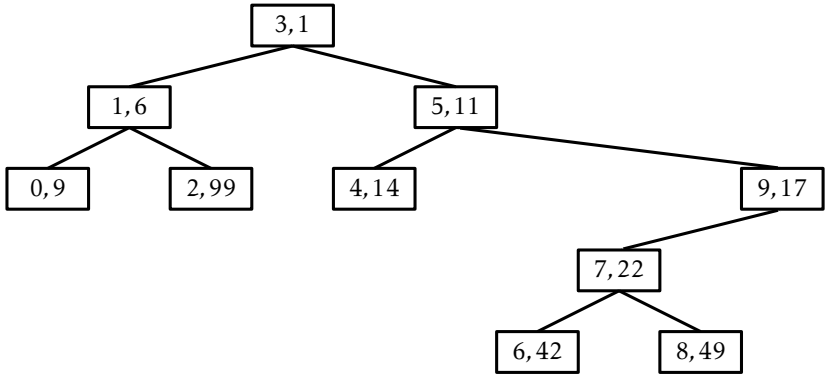
- (Lastnosti kopice) Pri vsakem vozlišču u, razen pri korenu, u.parent.p < u.p.

Z drugimi besedami, vsako vozlišče ima prioriteto manjšo od svojih dveh otrok. Primer je prikazan na Figure 6.5.

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) (Section 5.2) in kopice(h**eap**)(Chapter **??**).



Slika 6.5: 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 Figure 6.5 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.

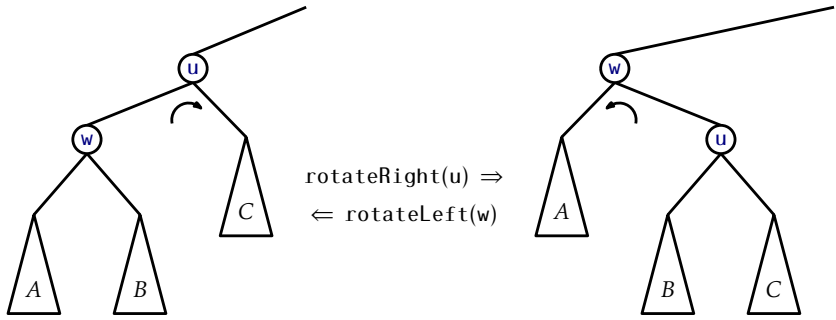
Ker so prioritete izbrane naključno, je to enako, če vzamemo naključno permutacijo ključev—v tem primeru permutacija je

$$\langle 3,1,0,5,9,4,7,6,8,2\rangle$$

—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 Lemma 6.1. Preračunavanju lemref rbs glede na drevesa Treap, imamo:

Lemma 6.2. *V drevesu Treap, ki shranjuje niz S z n ključi, naslednje izjave držijo:*

[3]Rang elementa x v nizu S elementov je število elementov v S, ki so manjši kot x.



Slika 6.6: Leva in desna rotacija v binarno iskalnem drevesu.

1. *Za vsak* $x \in S$, *pričakovana dolžina iskanja poti za* x *je* $H_{r(x)+1} + H_{n-r(x)} - O(1)$.

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 Lemma 6.2 prevzemajo preko naključne izbire prioritet za vsako vozlišče. To ne potrebuje nobene predpostavke o naključju ključev.

Lemma 6.2 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 Figure 6.6 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,

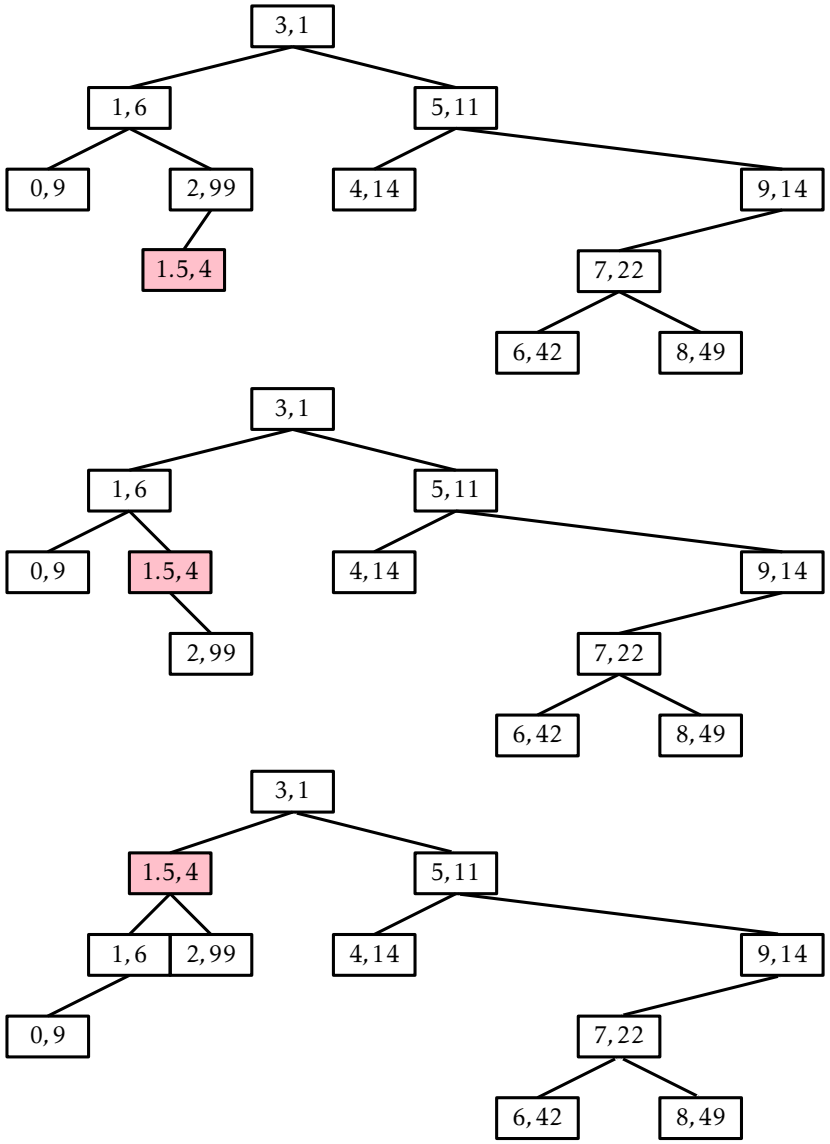
naše Treap drevo izpolnjuje lastnosti binarno iskalnega drevesa, vendar pa ni nujno, da izpolnjuje lastnosti kopice. Zlasti se lahko zgodi, da u.parent.p > u.p. Če se to zgodi, moramo izvesti rotacijo na vozlišču w=u.parent, tako da u postane starš w. Če u še naprej krši lastnosti kopice, bomo morali 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 Figure 6.7.

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 Lemma 6.2 je pričakovano trajanje iskalne poti maksimalno $2 \ln n + O(1)$. 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, $O(\log n)$. (Exercise **??** sprašuje po dokazu, da je



Slika 6.7: Dodajamo vrednost 1.5 v Treap drevo iz Figure 6.5.

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 `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 Figure 6.8.

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, Figure 6.8 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) $O(\log n)$

6.2.1 Povzetek

Naslednji izrek povzema zmogljivosti podatkovne strukture Treap:

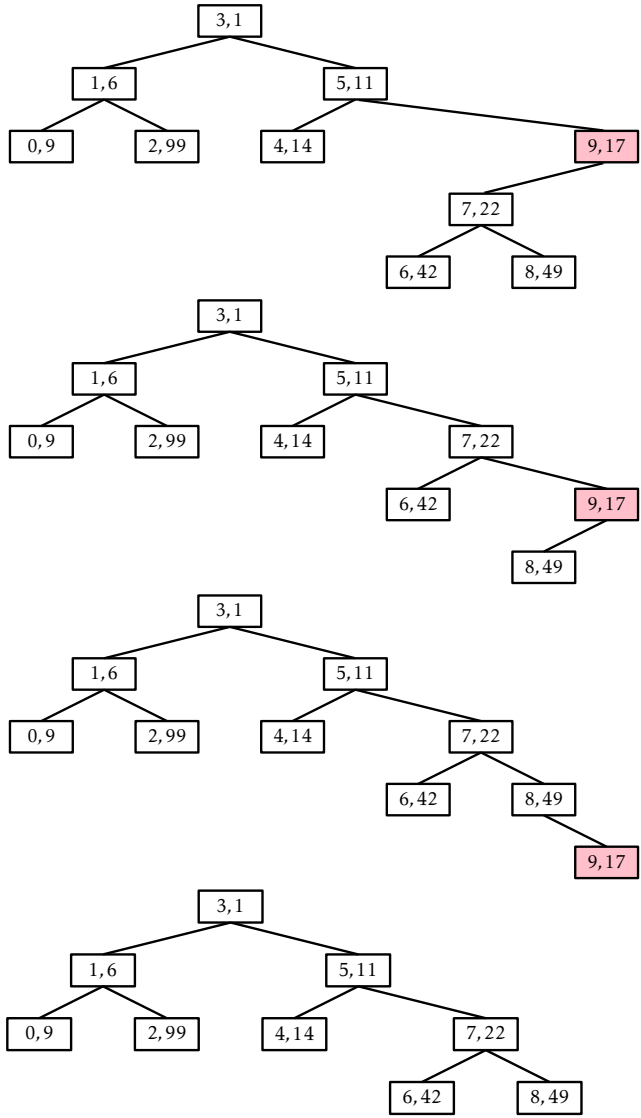
Theorem 6.2. *Treap implementira vmesnik SSet. Treap podpira operacije* add(x), remove(x) *in* find(x) *v pričakovanem času* $O(\log n)$ *za vsako operacijo.*

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 Exercise **??** 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

$$2\ln n + O(1) \approx 1.386\log n + O(1) \ .$$



Slika 6.8: Brišemo vrednost 9 iz drevesa Treap na Figure 6.5.

Tako je iskanje poti v `Treap` precej krajše in to se prevede v občutno hitrejše operacije nad `Treap` drevesih kot nad `Skiplist`. Exercise **??** v Chapter 4 prikazuje, kako se lahko pričakovana dolžina iskalne poti v `Skiplist` zmanjša na

$$e \ln n + O(1) \approx 1.884 \log n + O(1)$$

z uporabo pristranskega meta kovanca. Tudi s to optimizacijo, pričakovana trajanje iskanja poti v `SkiplistSSet` je občutno daljše kot v `Treap`.

Poglavje 7

Rdeče-Črna Drevesa

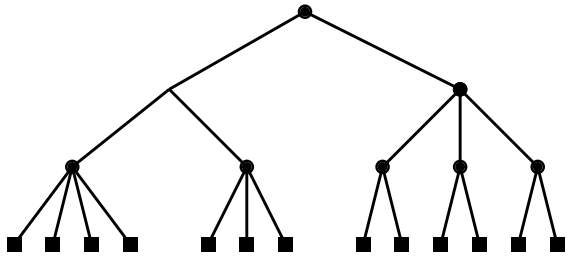
V tem poglavju so predstavljena rdeče-črna drevesa. So delno poravnana iskalna dvojiška drevesa z logaritemsko višino. Rdeče-črna drevesa so najbolj razširjena uporaba podatkovnih struktur. Pojavljajo se kot primarne iskalne strukture v mnogih knjižnicah, kot je naprimer Java Collections Framework in v posameznih implementacijah C++ Standard Template Library. Prav tako se uporabljajo znotraj jedra operacijskega sistema Linux. Nekaj 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 $O(\log n)$ *worst-case*.

3. Amortizirano število rotacij, ki nastopijo med izvajanjem operacij `add(x)` ali `remove(x)` je konstanta.

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 garantirano povezavo na njihovi višini, 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 vstaviti ali



Slika 7.1: 2-4 drevo višine 3.

izločiti element x manjši od časa, ki ga porabimo za iskanje elementa 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.

7.1 2-4 Trees

2-4 Drevo je korensko drevo, ki ima naslednje lastnosti:

Property 7.1 (height)**.** Vsi listi imajo enako globino.

Property 7.2 (degree)**.** Vsako notranje vozlišče ima 2, 3 ali 4 otrok.

Primer 2-4 drevesa je prikazan v Figure 7.1. Lastnost 2-4 dreves je logaritemska višina v številu listov:

Lemma 7.1. *Najvišja višina 2-4 drevesa z* n *je* $\log n$.

[1] Naključna iskalna binarna drevesa in samo uravnotežena binarna drevesa imajo enako lastnost. Glej vaje **??** in **??**.

Dokaz. Omejenost vsakega notranje vozlišče 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 $h \leq \log n$. □

7.1.1 Dodajanje lista

Dodajanje lista v 2-4 drevo je preprosto (Glej Figure 7.2). Če hželimo dodati na predzadnje mesto list u v vozlišče w, potem preprosto u postane otrok 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. Toda w' sedaj nima staršev, zato smo rekurzivno dali w' 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.

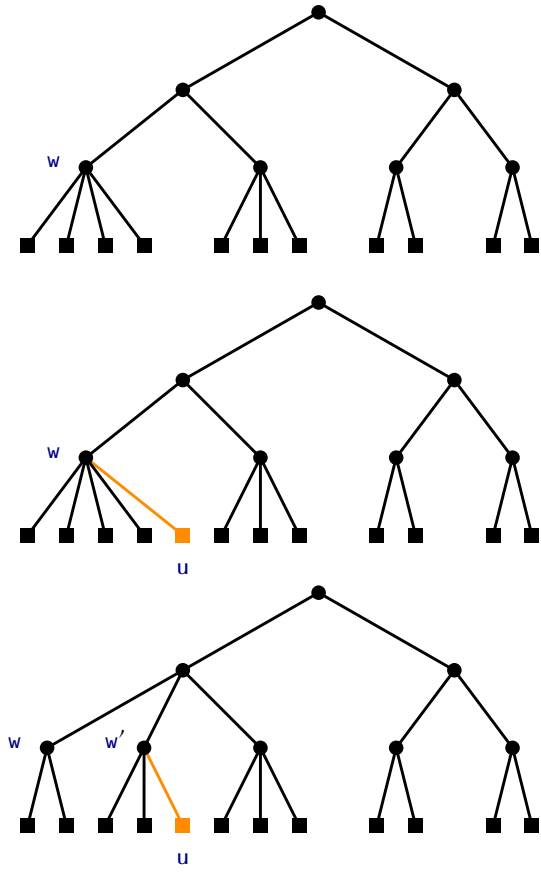
Ker višina 2-4 drevesa nikoli več kot $\log n$, se proces dodajanja listov konča po največ $\log n$ korakih.

7.1.2 Odstranjevanje lista

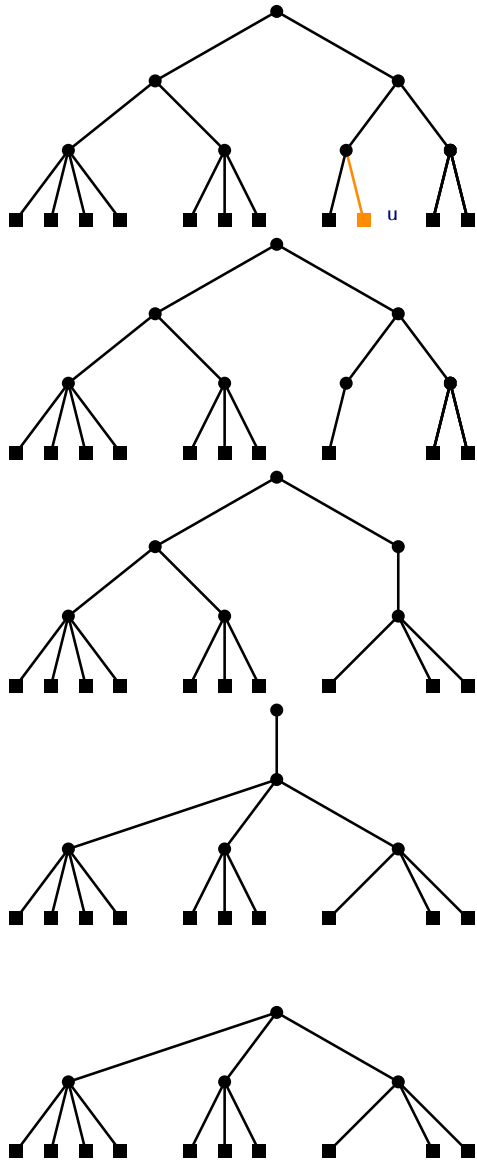
Odstranjevanje lista 2-4 drevesa je lahko rahlo bolj komplicirano kot dodajanje(Glej Figure 7.3). Da ločimo list u od njegovega starša w, ga samo odstranimo. Če ima w samo dva lista in mu mi enega izmed njih odstranimo, moramo drevo ustrezno popraviti, saj krši pravilo.

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* 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



Slika 7.2: Dodajanje lista v 2-4 drevo. Ta proces se konča po enemu razdeljevanju, ker ima `w.parent` stopnjo manj kot 4 pred dodajanjem.



Slika 7.3: Odstranjevanje lista z 2-4 drevesa. Ta proces sega vse do korena, saj ima vsak prednik in bratje vozlišča u samo dva otroka.

ali ne dosežemo 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 nikoli več kot $\log n$, se proces odstranjevanja listov konča po največ $\log n$ korakih.

7.2 RedBlackTree: A Simulated 2-4 Tree

Rdeče črno drevo je binarno iskalno drevo, katerega vsako vozlišče, u, je *rdeče* ali *črno*. Rdeče predstavlja vrednost 0, črno pa vrednost 1.

RedBlackTree

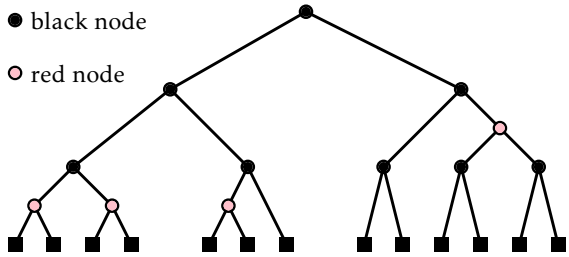
```
class RedBlackNode : public BSTNode<Node, T> {
  friend class RedBlackTree<Node, T>;
  char colour;
};
int red = 0;
int black = 1;
```

Pred in po spreminjanju rdeče-črnega drevesa, morata veljati naslednji dve lastnosti. Each property is defined both in terms of the colours red and black, and in terms of the numeric values 0 and 1.

Property 7.3 (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.)

Property 7.4 (no-red-edge)**.** Dve rdeči vozlišči nista med seboj nikoli sosednji. (Velja za vsako vozlišče u, razen korena, u.barva+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 Figure 7.4.



Slika 7.4: Primer rdeče-črnega drevesa, kjer je višina črnih 3. Zunanja (nil) vozlišča so v obliki kvadrata.

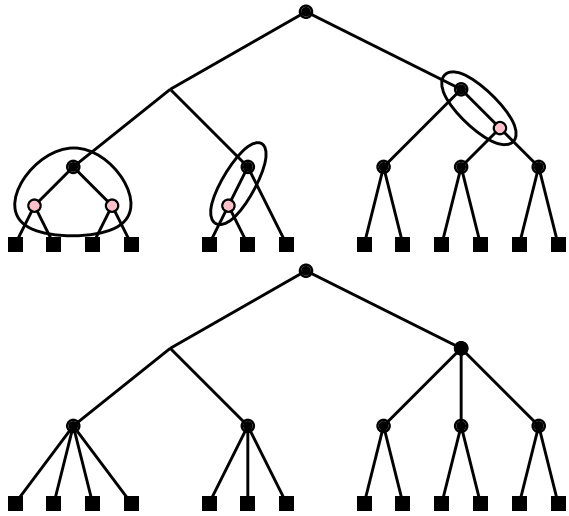
7.2.1 Rdeče-Črna drevesa in 2-4 Drevesa

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 Figure 7.5. 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 teji spremembi imamo drevo T' z 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 garantira, da vsaka pot od korena do lista v T' je 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 v teji poti sta črna in največ eden na vsaka dva notranja vozlišča je rdeč, tako, da ima ta pot največ $\log(n+1)$ črnih in največ $\log(n+1)-1$ rdečih vozlišč. Torej,



Slika 7.5: Vsako rdeče-črno drevo ima ustrezno 2-4 drevo.

najdaljša pot od korena do kateregakoli *notranjega* vozlišča v T je največ

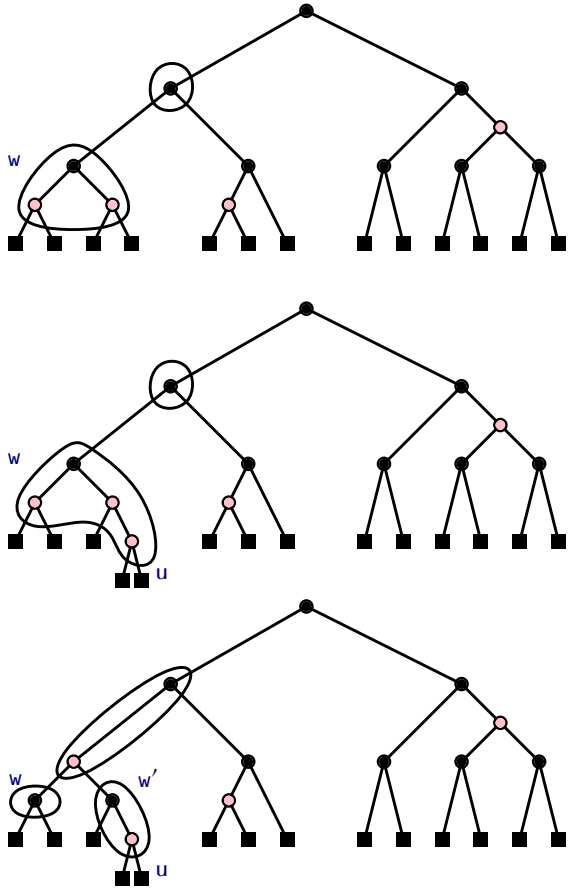
$$2\log(n+1) - 2 \leq 2\log n \ ,$$

za kateregakoli $n \geq 1$. To dokaže najpomembnejšo lastnost rdeče-črnih dreves:

Lemma 7.2. *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 otroci v 2-4 drevesu. Vozlišče v 2-4 drevesu s petimi otroci 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 Figure 7.6.



Slika 7.6: Simuliranje operacije deljenja 2-4 drevesa med dodajanjem v rdeče-črno drevo. (To simulira dodajanje v 2-4 drevo prikazano na Figure 7.2.)

Podobno, implementacija `remove(x)` zahteva metodo za združevanje dveh vozlišč in izposojo sorodnikovega otroka. Združitev dveh vozlišč je inverz deljenja vozlišč (prikazano na Figure 7.6) 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 no-red-edge in lastnost black-height. Medtem ni več presenetljivo, da je to lahko izvedljivo, veliko je število primerov, ki jih moramo upoštevati, če poikušamo narediti ditektno simulacijo 2-4 drevesa z rdeče-črnim drevesom. Na neki točki, postane lažje če neupoštevamo osnovnih 2-4 dreves in delamo neposredno k ohranjanju lastnosti rdeče-črnih dreves.

7.2.2 Levo-viseca Rdece-Crna Drevesa

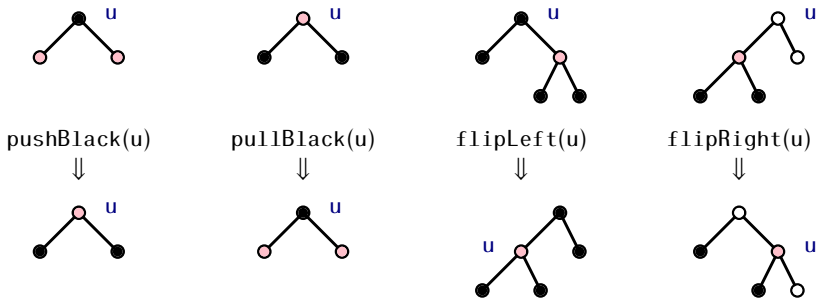
Ne obstaja nobena definicija rdeče-črnega drevesa. Namesto tega, je družina struktur, ki uspe ohranjati lastnosti black-height in no-red-edge med operacijama `add(x)` in `remove(x)`. Drugačne strukture to delajo na drugačne načine. Tukaj mi implementiramo podatkovno strukturo, ki jo kličemo `RdeceCrnoDrevo`. Ta struktura implementira posebno obliko rdeče-črnega drevesa, ki zadovoljuje dodatni lastnosti.

Property 7.5 (left-leaning)**.** Na kateremkoli vozlišču `u`, če je `u.levo` črno, potem `u.desno` je črno.

Opomnimo, da rdeče-črno drevo prikazano na Figure 7.4 ne zadošča levo-viseči lastnosti; krši jo starš rdečega vozlišča na najbolj desni poti od korena proti listu.

Razlog za ohranjanje levo-viseče lastnosti je, da zmanjšuje število primerov soočenih pri posodabljanju drevesa med operacijama `add(x)` in `remove(x)`. V smuslu 2-4 dreves, to pomeni, da vsako 2-4 drevo ima edinstveno zastopanje: Vozlišče stopnje dva postane črno vozlišče z dvemi črnimi otroci. 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 opišemo implementacijo operacij `add(x)` in `remove(x)` v podrobnosti, prvo predstavimo nekaj osnovnih podoperacij uporabljenih v



Slika 7.7: Flips, pulls and pushes

teh metodah prikazanih v Figure 7.7. Prvi dve podoperaciji so za manipulacijo barv med ohranjanjem lastnosti black-height. Operacija `pushBlack(u)` metoda vzame za vhod črno 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.desno` in izvede levo rotacijo nad vozliščem `u`. Ta metoda obrne barve teh dveh vozlišč tako kot tudi relacijo njihovih staršev-otrok:

RedBlackTree
```
void flipLeft(Node *u) {
  swapcolours(u, u->right);
  rotateLeft(u);
}
```

Operacija `flipLeft(u)` je pposebno uporabna pri povrnitvi levo-viseče la-

stnosti na vozlišču u, katero krši to lastnost (ker je u.left črno in u.right rdeče). V tem posebnem primeru, smo lahko zagotovi, da ta operacija ohranja obe lastnosti black-height in no-red-edge. Relacija flipRight(u) je simetrična z fliPLeft(u), ko so vloge levega in desnega obrnjene.

RedBlackTree
```
void flipRight(Node *u) {
  swapcolours(u, u->left);
  rotateRight(u);
}
```

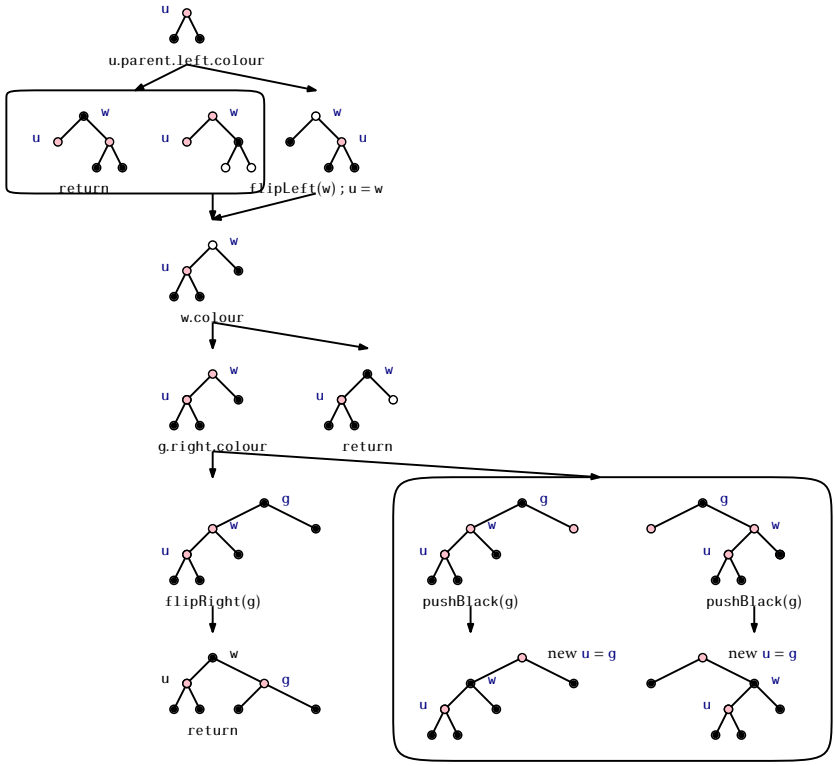
7.2.3 Dodajanje

Za implementacijo add(x) v RdeceCrnoDrevo, izvedemo standardno BinarnoIskalnoDrevo vstavljanje za dodajanje novega lista, u, z u.x = x in nastavimo u.colour = red. Opomnimo, da to ne spremeni črne višine kateremukoli vozlišču, torej ne krši lastnosti black-height. To pa lahko krši levo-visečo lastnost (če je u desni otrok svojega starša), in lahko krši no-red-edge lastnost (če je ujev starš rdec). 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 Figure 7.8, metoda addFixup(u) vzame za vhod vozlišče u katerega barva je rdeča in katero bi lahko kršilo lastnost no-red-edge in/ali levo-ležečo lastnost. Slednja razprava je verjetno nemogoča za sledelje brez sklicevanja na Figure 7.8 ali ponovnega ustvarjanja na kosu papirja. Dejansko, bralec bi si moral preučiti to sliko preden nadaljuje.

Če je u koren drevesa, potem lahko pobarvamo u v črno za pridobitev nazaj obeh lastnosti. Če je tudi ujev sorodnik rdeč, potem mora biti ujev



Slika 7.8: Enotni postopek v procesu popravljanja Property 2 po vstavljanju.

starš črn, torej oba levo-viseča in no-red-edge lastnost že držita.

Če ne, najprej določimo, če je ujev starš, w, kršil levo-visečo lastnost in, če je tako, izvedemo operacijo `flipLeft(w)` in nastavimo u = w. To nas pusti v lepo definiranem stanju: u je levi otrok starša, w, torej w sedaj zadošča levi-viseči lastnosti. Vse kar nam ostane je, da zagotovimo no-red-edge lastnost na u. Moramo samo še skrbeti za primer v katerem je w rdeč, sicer v nasprotnem primeru u že zadošča lastnosti no-red-edge.

Glede na to, da še nismo končali, u je rdeč in w je rdeč. Lastnost no-red-edge (katero krši u in ne w) implicira, da ujev stari starš g obstaja in je črn. Če je gjev desni otrok rdeč, potem levo-viseča lastnost zagotavlja, da oba gjev otrok je rdeč in klic na `pushBlack(g)` naredi g rdečega in w črnega. To povrne no-red-edge lastnost na u, ampak lahko povzroči, da jo krši na g vozlišču, tako, da celoten proces začne z u = g.

Če je gjev otrok črn, potem klic na `flipRight(g)` nardi w črnega starča od g in naredi wju dva rdeča otroka, u in g. To zagotovi, da u zadošča no-red-edge lastnosti in g zadošča levo-viseči lastnosti. V tem primeru se lahko ustavimo.

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.

7.2.4 Odstranitev

`odstrani(x)` operacija v `RdeceCrnemDrevesu` je najbolj zahtevna za implementacijo in to velja za vse različice rdeče-črnega drevesa. Tako kot `odstrani(x)` operacija v `BinarnemIskalnemDrevesu`, išče vozlišče `w` z enim otrokom, `u` in preplete `w` iz drevesa tako, da `w.parent` sprejme `u`.

Težava lahko nastane takrat, ko je `w` črn, saj s tem kršimo lastnost višine črnih v `w.parent`. Temu se lahko začasno izognemo z dodajanem `w.barva` do `u.barva`. To predstavlja dve težavi: (1) če se `u` in `w` obe začneta z črno, potem `u.barva` + `w.barva` = 2 (dvojna črna), ki pa ni veljavna. Če je bil `w` rdeč, se ga nadomesti s črnim vozliščem `u`, kateri lahko krši levo usmerjeno lastnost pri `u.parent`. Obe težave lahko rešimo tako, da pokličemo metodo `removeFixup(u)`.

`removeFixup(u)` metoda 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 odpravljen. The `removeFixup(u)` method takes as its input a node `u` whose colour is black (1) or double-black (2). If `u` is double-black, then `removeFixup(u)` performs a series of rotations and recolouring operations that move the double-black node up the tree until it can be eliminated. Skozi ta postopek se vozlišče `u` spreminja, dokler ne pride do konca postopka, `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, `removeFixup(u)` metoda na koncu preverja, če `u`-jev starš krši levo usmerjeno lastnost in če jo to popravi.

RedBlackTree
```
void removeFixup(Node *u) {
  while (u->colour > black) {
    if (u == r) {
```

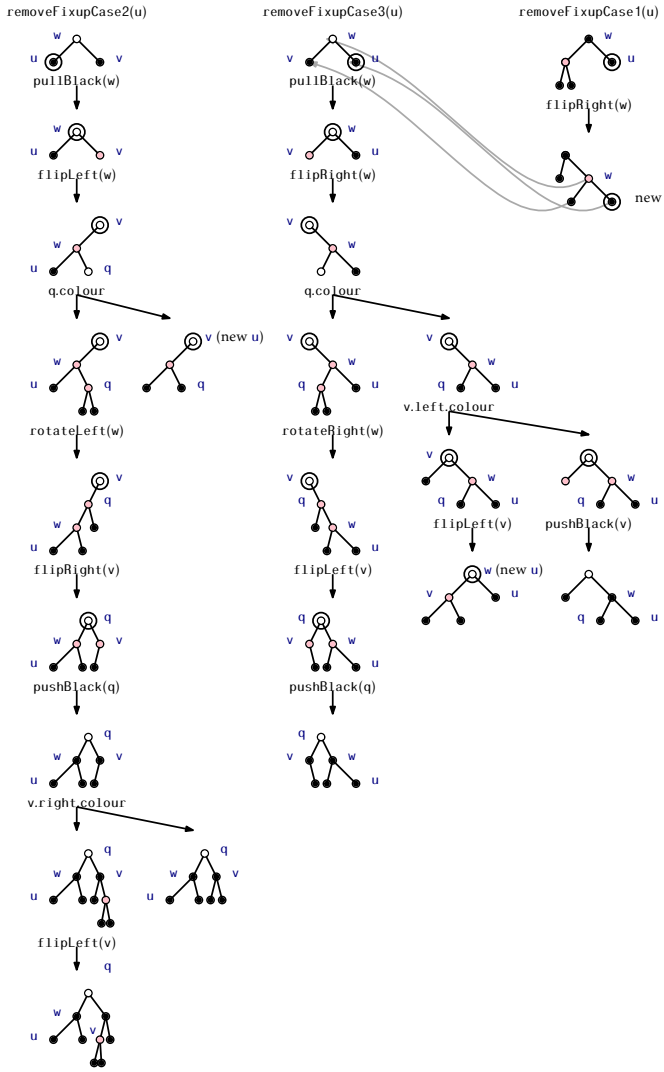
```
      u->colour = black;
    } else if (u->parent->left->colour == red) {
      u = removeFixupCase1(u);
    } else if (u == u->parent->left) {
      u = removeFixupCase2(u);
    } else {
      u = removeFixupCase3(u);
    }
  }
  if (u != r) { // restore left-leaning property, if needed
    Node *w = u->parent;
    if (w->right->colour == red && w->left->colour == black) {
      flipLeft(w);
    }
  }
}
```

`removeFixup(u)` metoda je predstavljena na figrefrb-removefix Naslednjemu besedilu bo težko, če ne kar nemogoče slediti brez sklicevanja na Figure 7.9. Vsaka ponovitev zanke v `removeFixup(u)` postopku 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: `u` je sorodstven, `v`, je rdeč. V tem primeru, je `u` sorodstven levemu otroku svojega starša, `w` (z lastnostjo levo-visečega). Opravimo desno rotacijo na `w` in nadaljujemo z naslednjo ponovitvijo. left child of its parent, `w` (by the left-leaning property). We perform a right-flip at `w` and then proceed to the next iteration. Upoštevamo, da ta ukrep povzroči, da `w`-jev starš krši lasnost levo-visečega 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: `u` je sorodstven, `v`, je črn , `u` je levi otrok njegovega starša, `w`. V



Slika 7.9: A single round in the process of eliminating a double-black node after a removal.

tem primeru pokličemo funkcijo `pullBlack(w)`, ki obarvai `u` črno, `v` rdeče in potemni barvo `w` v črno ali dvojno-črno. V tem primeru `w` ne izpolnjuje lastnost levo-visečega, zato to uredimo tako, da pokličemo `flipLeft(w)`.

V tem primeru je `w` rdeč, `v` pa je koren podrevesa v katerem smo začeli. Preveriti moram še, če `w` ne povzroča kršitve lastnosti no-red-egde. To naredimo tako, da preverimo `w`-jevega desnega otroka `q`. Če je `q` črn, potem `w` izpolnjuje lastnost no-red-edge in nadaljujemo z naslednjo ponovitvijo z $u = v$.

V nasprotnem primeru (`q` je rdeč) sta obe lastnosti, nobenega-rdečega-lista – rdeče pravilo in levo-visečega, kršeni pri `q` in `w`. Levo-visečega popravimo s klicem `rotateLeft(w)`, sedaj nam ostane le še lastnost nobenega-rdečega-lista, 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 pa 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 nobenega-rdečega-lista in črna-višina. Ostane nam pa samo še ena težava: če ima `v` desnega sina, ki je rdeč, kršimo lastnost levo-visečega. 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;
  }
```

```
}
```

Case 3: u's sibling is black and u is the right child of its parent, w. This case is symmetric to Case 2 and is handled mostly the same way. The only differences come from the fact that the left-leaning property is asymmetric, so it requires different handling.

As before, we begin with a call to `pullBlack(w)`, which makes v red and u black. A call to `flipRight(w)` promotes v to the root of the subtree. At this point w is red, and the code branches two ways depending on the colour of w's left child, q.

If q is red, then the code finishes up exactly the same way as Case 2 does, but is even simpler since there is no danger of v not satisfying the left-leaning property.

The more complicated case occurs when q is black. In this case, we examine the colour of v's left child. If it is red, then v has two red children and its extra black can be pushed down with a call to `pushBlack(v)`. At this point, v now has w's original colour, and we are done.

If v's left child is black, then v violates the left-leaning property, and we restore this with a call to `flipLeft(v)`. We then return the node v so that the next iteration of `removeFixup(u)` then continues with 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;
```

```
      }
    }
}
```

Each iteration of `removeFixup(u)` takes constant time. Cases 2 and 3 either finish or move `u` closer to the root of the tree. Case 0 (where `u` is the root) always terminates and Case 1 leads immediately to Case 3, which also terminates. Since the height of the tree is at most $2\log n$, we conclude that there are at most $O(\log n)$ iterations of `removeFixup(u)`, so `removeFixup(u)` runs in $O(\log n)$ time.

7.3 Summary

Naslednji izrek povzema učinkovitost podatkovne strukture `RdeceCrno-Drevo`:

Theorem 7.1. *`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:

Theorem 7.2. *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.*

Skicirali smo le dokaz za Theorem 7.2. 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 to pomeni Theorem 7.2.

Dokaz tega izreka za 2-4 drevo uporablja potencial odplačne analize.[2] Definiraj potencial za notranje vozlišče `u` v 2-4 drevesu kot

$$\Phi(u) = \begin{cases} 1 & \text{če ima } u \text{ 2 otroka} \\ 0 & \text{če ima } u \text{ 3 otroke} \\ 3 & \text{če ima } u \text{ 4 otroke} \end{cases}$$

[2] Oglej si Lemma 2.2 in Lemma 3.1 dokaze za potencialno metodo v ostalih aplikacijah.

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.

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 $3m/2$. Theorem 7.2 izhaja iz te analize in povezav med 2-4 drevesi in rdečečrnimi drevesi.

7.4 Discussion and Exercises

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 left-leaning uravnano drevo, ki je podobno rdeče-črnim drevesom, vendar pa ima left-leaning dodatno omejitev; vsako vozlišče ima 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` predstavljena v tem poglavju.

Sedgewick [**?**] pa je predstavil dve verziji left-leaning rdeče-črnih dreves. These use recursion along with a simulation of top-down splitting and združevanj v 2-4 drevesih. Kombinacija obeh tehnih nam omogoča zelo kratko in elegantno kodo.

Povezana, a starejša, podatkovna struktura je *AVL tree* [**?**]. AVL drevesa so *height-balanced*: V vsakem vozlišče 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 $\varphi^h/\sqrt{5}$, where $\varphi = (1+\sqrt{5})/2 \approx 1.61803399$ is the *golden ratio*. (Bolj natančno $|\varphi^h/\sqrt{5} - F(h)| \leq 1/2$.) Arguing as in the proof of Lemma 7.1, this implies

$$h \leq \log_\varphi \mathtt{n} \approx 1.440420088 \log \mathtt{n} \ ,$$

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 Figure 7.10.

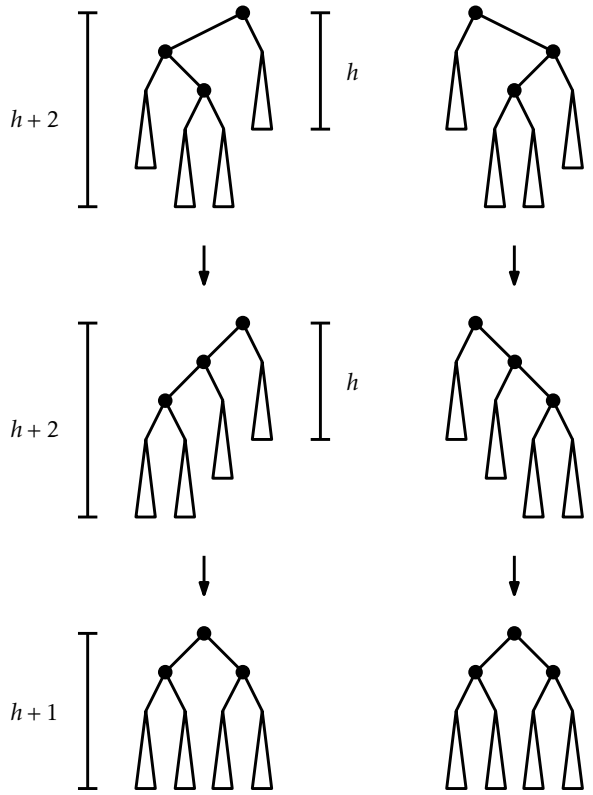
Uporaba Anderssonove variante in Sadgewickove variante rdeče-črnih dreves in uporaba AVL dreves je enostavnejša kot uporaba `RedBlackTree` strukture, kot je definirana tukaj. Žal pa ne more nobena od njih zagotavljati, da bi bil amortizacijski čas $O(1)$, za vsako posodobitev uravnovešen. In particular, there is no analogue of Theorem 7.2 for those structures.

Exercise 7.1. Nariši 2-4 drevo, ki ustreza `RedBlackTree` iz Figure 7.11.

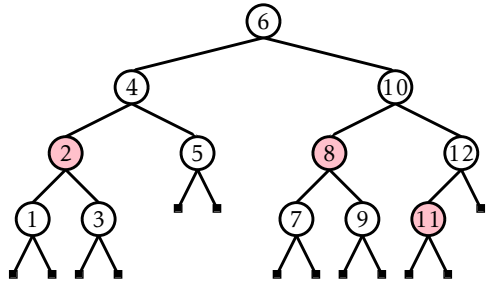
Exercise 7.2. Nariši dodajanje elementov 13, 3.5 in 3.3 na `RedBlackTree` iz Figure 7.11.

Exercise 7.3. Nariši odstranjevanje elementov 11, 9, ter 5 na `RedBlackTree` iz Figure 7.11.

Exercise 7.4. Pokaži, da za poljubno velike vrednosti `n`, obstaja rdeče-črno drevo z `n` vozlišči, ki imajo višino $2\log \mathtt{n} - O(1)$.



Slika 7.10: Uravnoteženje v AVL drevesih. Največ dve rotaciji sta potrebni, da vozlišče z poddrevesoma višine h in $h+2$ spremenimo v vozlišče z poddrevesoma višine $h+1$.



Slika 7.11: A red-black tree on which to practice.

Exercise 7.5. Preuči operaciji `pushBlack(u)` and `pullBlack(u)`. Kaj naredijo ti dve operaciji na 2-4 drevesu, ki temelji na simulaciji z rdeče-črnim drevesom.

Exercise 7.6. 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 $2\log n - O(1)$.

Exercise 7.7. Zakaj metoda `odstrani(x)` v `RedBlackTree` izvede operacijo `u.parent = w.parent`? Naj nebi bilo to storjeno že z klicem metode `splice(w)`?

Exercise 7.8. 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?

Exercise 7.9. Predpostavimo, da imamo binarno iskalno drevo z n vozlišči in višini največ $2\log n - 2$. Is it always possible to colour the nodes red and black so that the tree satisfies the black-height and no-red-edge properties? If so, can it also be made to satisfy the left-leaning property?

Exercise 7.10. 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 $O(h)$.

Exercise 7.11. Nadgradi rešitev iz Exercise 7.10, 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 $O(\max\{h_1, h_2\})$.

Exercise 7.12. Dokaži, da mora AVL drevo pri izvajanju `add(x)` metode, izvesti največ eno operacijo uravnoteženja (vključuje največ dve rotaciji; glej Figure 7.10). Podaj primer AVL drevesa in klica metode `remove(x)` na tem drevesu, ki zahteva $\log n$ operacij uravnoteženja.

Exercise 7.13. Napiši razred `AVLTree`, ki uporablja AVL drevo kot je opisano zgoraj. Primerjaj hitrost izvajanja s hitrostjo `RedBlackTree`. Katera izvedba drevesa ima hitrejšo operacijo `find(x)`?

Exercise 7.14. 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 8

Kopice

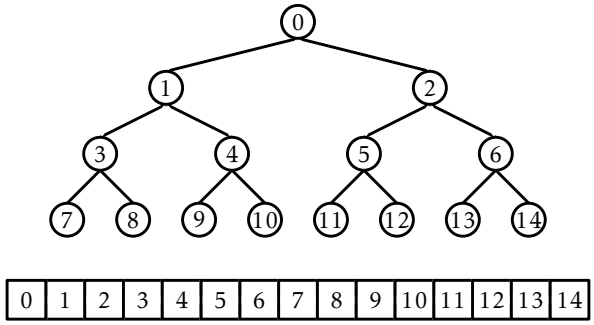
V tem poglavju si bomo pogledali 2 implementacije zelo uporabne podatkovne strukture `Polje` s prednostjo. Obe od teh dveh struktur sta posebne oblike Binarnega drevesa imenovani *Kopica*, kar pomeni "neorganizirana kopica". To je v nasprotju z binarnimi iskalnimi drevesi pri katerih pomislimo na zelo urejeno kopico.

Prva izvedba kup uporablja polje, da simuliramo popolno binarno drevo. To zelo hitra implementacija je osnova za enega izmed najhitrejsih znanih sortirnih algoritmov, in sicer Kopicno urejanje. (see Section **??**).

Druga implementacija je bazirana na bolj fleksiblinih binarnih drevesih, ki podpira `meld`(h) operacijo, ki omogoca vrsti s prednostjo da obsorbira elemente druge vrste s prednostjo h.

8.1 `BinarnaKopica`: inplicitno binarno drevo

Nasa prva implementacije `Vrste` (s prednostjo) temelji na tehniki, ki je stara preko 400 let. *Eytzingerjeva metoda* Our first implementation of a (priority) `Queue` is based on a technique that is over four hundred years old. *Eytzinger's method* nam omogoca da predstavimp popolno binarno drevo kot polje, v katerem imamo vozlisca postavlna v vrsto iz leve proti desni. (glej Section 5.1.2). Na ta nacin je koren drevesa spravljen na pozociji 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 Figure 8.1.



Slika 8.1: Eytzingerjeva meroda predstavlja poplno binarno drevo kot polje.

ce uporabimo Eytzingerjevo metodo na zadosti velikih drevesih se zacnejo pojavljati vzroci. Levi otrok vozlica pri indexu `i` je na indexu `left(i)` = 2`i`+1 in desni otrok vozlisca pri indexu `i` je na indexu `right(i)` = 2`i` + 2 Stars vozlisca 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;
}
```

`BinarnaKopica` uporablja to tehniko da implicitno predstavi popolno binarno drevo v katerem so elementi *Kopicno urejeni*: Vrednost shranjena na kateremkoli indexu `i` ni manjpa kot vrednost shranjena na kateremkoli indexu `parent(i)` razen izjeme vrednosti korena `i` = 0. To nam omogoca da je najmanjsa vrednost `vrste` s prednostjo tako na shranjena na poziciji 0(koren).

V `BinarnaKopici`, je `n` elementov shranjenih v vrsti `a`:

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

Implementacija operacije Doda j(x) je preprosta. Kot vse strukture bazirane na polju najprej pogledamo ce je a pol (preverimo a.length = n) in ce le povecamo a a[n] in increment n. Na tej poti je samo se kar nam ostane, da zagotovimo lastnost kopice. To delamo tako da premescamo x z njegovim starsem dokler ni x manjsi od svojega starsa. Implementing the add(x) operation is fairly straightforward. As with See Figure **??**.

BinaryHeap

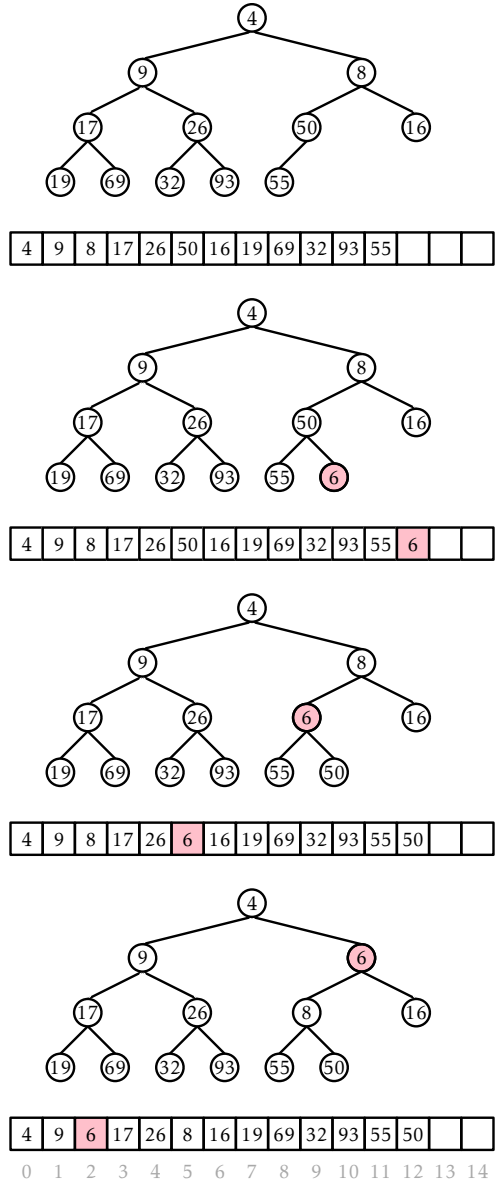
```
bool add(T x) {
  if (n + 1 > a.length) resize();
  a[n++] = x;
  bubbleUp(n-1);
  return true;
}
void bubbleUp(int i) {
  int p = parent(i);
  while (i > 0 && compare(a[i], a[p]) < 0) {
    a.swap(i,p);
    i = p;
    p = parent(i);
  }
}
```

Implementavija remove() operacije, katera odstarani najmanjso vrednost v kopici, je malo tesje. Vemo kje je najmanjsi element (v korenu), ampak ga moramo nademstiti potem ko ga odstranimi in zagotoviti to ohranjamo lastnosti kopice.

Najlasji nacin da to naredimo da koren nadomestimo v vrednostjo a[n – 1], zbrisemo vrednost in decrement ”n”. Na salost novi koren najverjetneje ni najmanjsi element, zato ga moramo prestaviti dol v kopici. To naredimo tako da ponavljajoce primerjamo ta element z njegovimi otroki. ce je najmanjsi v kopici smo kocali, v nasprotnem primeru ga zamenjamo z najmanjsim od njegovih otrok in nadaljujemo z primerjavo.

BinaryHeap

```
T remove() {
  T x = a[0];
  a[0] = a[--n];
  trickleDown(0);
  if (3*n < a.length) resize();
  return x;
```



Slika 8.2: Adding the value 6 to a `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 ostali iz polja implementirane strukture, bomo mi ignorirali cas porabljen v celicah za poveca j(), ker se to lahko obracunava na amortizacijskem argumentu iz Lemma 2.1. Pretekli cas za Doda j() in remove() je odvisen od visine (implicitnega) binarnega drevesa. Na sreco je to *popolno* Binarno drevo; vsako nivo razen zadnje ima maximalno stevilo vozlisc. Tako, je visina drevesa enaka h in ima najmanj 2^h vizlisc.

zacnimo na ta nacin:

$$n \geq 2^h \ .$$

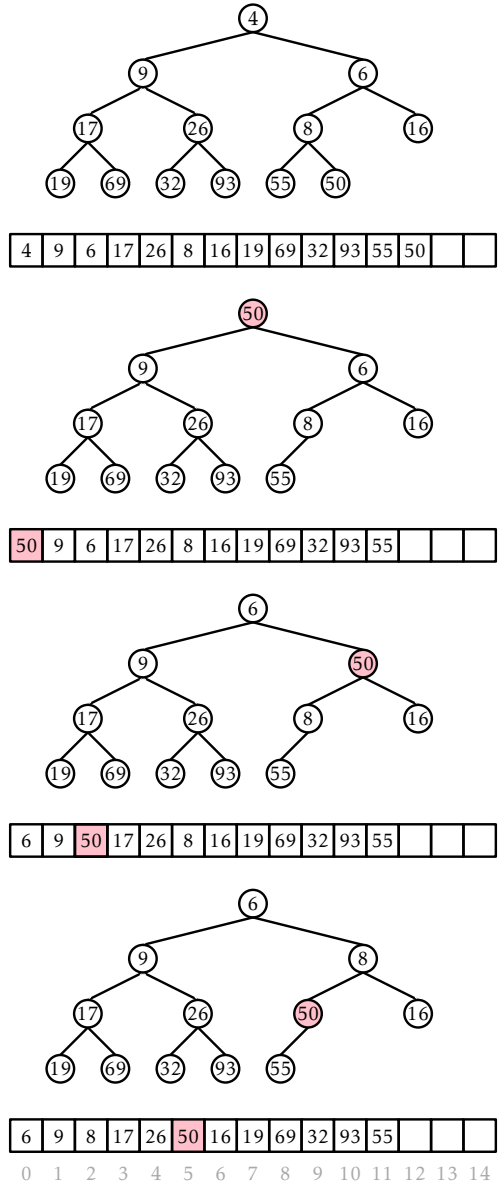
Algoritem da na obeh straneh enacbe

$$h \leq \log n \ .$$

teko obe add(x) in remove() operaciji teceta v $O(\log n)$ casu.

8.1.1 Summary

Naslednji teorem povzame uspesnost Binarnekopice



Slika 8.3: Odstranjevanje 4 najmanjsega elementa, iz `BinarneKopice`.

Theorem 8.1. BinarnaKopica *implementira* Polje *(s prednostjo). Ignoriramo ceno polja da se povecа*resize(), *Binarna Kopica podpira oberaciji* add(x) *in* remove() *v casu* $O(\log n)$ *na operacijo.*

Tako naprej, zacetek z prazno Binarnokopico, *katerokoli zapredje* m add(x) *in* remove() *opraciji je rezultat skupaj* $O(m)$ *cas enak porabljen za povecanje* resize().

8.2 MeldableHeap: A Randomized Meldable Heap

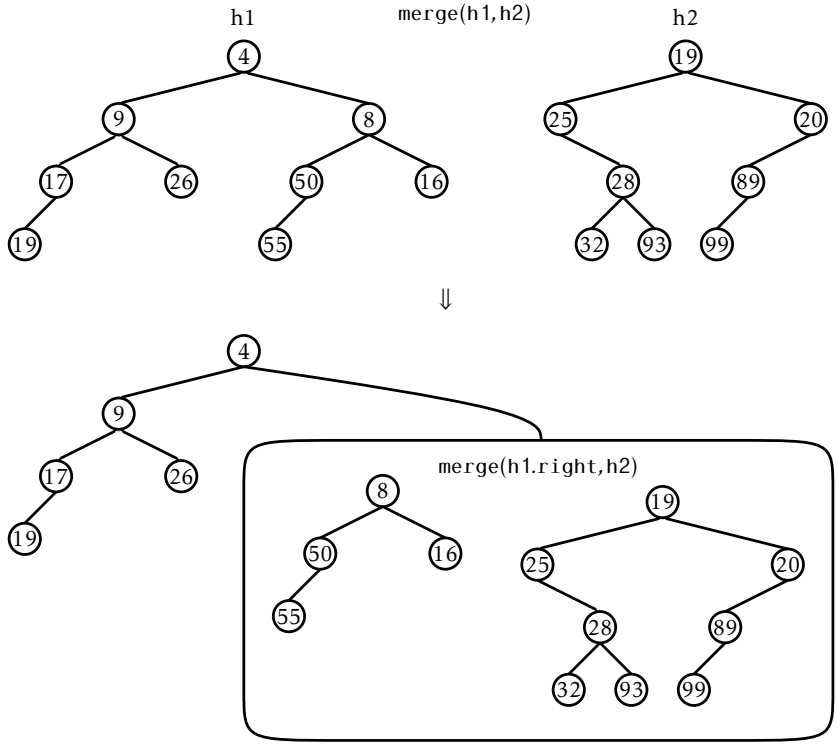
In this section, we describe the MeldableHeap, a priority Queue implementation in which the underlying structure is also a heap-ordered binary tree. However, unlike a BinaryHeap in which the underlying binary tree is completely defined by the number of elements, there are no restrictions on the shape of the binary tree that underlies a MeldableHeap; anything goes.

The add(x) and remove() operations in a MeldableHeap are implemented in terms of the merge(h1,h2) operation. This operation takes two heap nodes h1 and h2 and merges them, returning a heap node that is the root of a heap that contains all elements in the subtree rooted at h1 and all elements in the subtree rooted at h2.

The nice thing about a merge(h1,h2) operation is that it can be defined recursively. See Figure 8.4. If either h1 or h2 is nil, then we are merging with an empty set, so we return h2 or h1, respectively. Otherwise, assume $h1.x \leq h2.x$ since, if $h1.x > h2.x$, then we can reverse the roles of h1 and h2. Then we know that the root of the merged heap will contain h1.x, and we can recursively merge h2 with h1.left or h1.right, as we wish. This is where randomization comes in, and we toss a coin to decide whether to merge h2 with h1.left or h1.right:

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);
```



Slika 8.4: Merging `h1` and `h2` is done by merging `h2` with one of `h1.left` or `h1.right`.

```
    if (h1->left != nil) h1->left->parent = h1;
  } else {
    h1->right = merge(h1->right, h2);
    if (h1->right != nil) h1->right->parent = h1;
  }
  return h1;
}
```

In the next section, we show that `merge(h1,h2)` runs in $O(\log n)$ expected time, where `n` is the total number of elements in `h1` and `h2`.

With access to a `merge(h1,h2)` operation, the `add(x)` operation is easy. We create a new node `u` containing `x` and then merge `u` with the root of our heap:

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;
}
```

This takes $O(\log(n+1)) = O(\log n)$ expected time.

The `remove()` operation is similarly easy. The node we want to remove is the root, so we just merge its two children and make the result the root:

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;
}
```

Again, this takes $O(\log n)$ expected time.

Additionally, a `MeldableHeap` can implement many other operations in $O(\log n)$ expected time, including:

- `remove(u)`: remove the node `u` (and its key `u.x`) from the heap.
- `absorb(h)`: add all the elements of the `MeldableHeap` `h` to this heap, emptying `h` in the process.

Each of these operations can be implemented using a constant number of `merge(h1,h2)` operations that each take $O(\log n)$ expected time.

8.2.1 Analysis of `merge(h1,h2)`

The analysis of `merge(h1,h2)` is based on the analysis of a random walk in a binary tree. A *random walk* in a binary tree starts at the root of the tree. At each step in the random walk, a coin is tossed and, depending on the result of this coin toss, the walk proceeds to the left or to the right child of the current node. The walk ends when it falls off the tree (the current node becomes `nil`).

The following lemma is somewhat remarkable because it does not depend at all on the shape of the binary tree:

Lemma 8.1. *The expected length of a random walk in a binary tree with* n *nodes is at most* $\log(n+1)$.

Dokaz. The proof is by induction on n. In the base case, $n = 0$ and the walk has length $0 = \log(n+1)$. Suppose now that the result is true for all non-negative integers $n' < n$.

Let n_1 denote the size of the root's left subtree, so that $n_2 = n - n_1 - 1$ is the size of the root's right subtree. Starting at the root, the walk takes one step and then continues in a subtree of size n_1 or n_2. By our inductive hypothesis, the expected length of the walk is then

$$\mathrm{E}[W] = 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \ ,$$

since each of n_1 and n_2 are less than n. Since log is a concave function, $\mathrm{E}[W]$ is maximized when $n_1 = n_2 = (n-1)/2$. Therefore, the expected number of steps taken by the random walk is

$$\begin{aligned}\mathrm{E}[W] &= 1 + \frac{1}{2}\log(n_1+1) + \frac{1}{2}\log(n_2+1) \\ &\le 1 + \log((n-1)/2+1) \\ &= 1 + \log((n+1)/2) \\ &= \log(n+1) \ .\end{aligned}$$

□

We make a quick digression to note that, for readers who know a little about information theory, the proof of Lemma 8.1 can be stated in terms of entropy.

Information Theoretic Proof of Lemma 8.1. Let d_i denote the depth of the ith external node and recall that a binary tree with n nodes has $n+1$ external nodes. The probability of the random walk reaching the ith external node is exactly $p_i = 1/2^{d_i}$, so the expected length of the random walk is given by

$$H = \sum_{i=0}^{n} p_i d_i = \sum_{i=0}^{n} p_i \log\left(2^{d_i}\right) = \sum_{i=0}^{n} p_i \log(1/p_i)$$

The right hand side of this equation is easily recognizable as the entropy of a probability distribution over $n+1$ elements. A basic fact about the entropy of a distribution over $n+1$ elements is that it does not exceed $\log(n+1)$, which proves the lemma. □

With this result on random walks, we can now easily prove that the running time of the `merge(h1,h2)` operation is $O(\log n)$.

Lemma 8.2. *If* `h1` *and* `h2` *are the roots of two heaps containing* n_1 *and* n_2 *nodes, respectively, then the expected running time of* `merge(h1,h2)` *is at most* $O(\log n)$*, where* $n = n_1 + n_2$.

Dokaz. Each step of the merge algorithm takes one step of a random walk, either in the heap rooted at `h1` or the heap rooted at `h2`. The algorithm terminates when either of these two random walks fall out of its corresponding tree (when `h1` = `null` or `h2` = `null`). Therefore, the expected number of steps performed by the merge algorithm is at most

$$\log(n_1+1) + \log(n_2+1) \leq 2\log n \ .$$ □

8.2.2 Summary

The following theorem summarizes the performance of a `MeldableHeap`:

Theorem 8.2. *A* `MeldableHeap` *implements the (priority)* `Queue` *interface. A* `MeldableHeap` *supports the operations* `add(x)` *and* `remove()` *in* $O(\log n)$ *expected time per operation.*

8.3 Discussion and Exercises

The implicit representation of a complete binary tree as an array, or list, seems to have been first proposed by Eytzinger [**?**]. He used this representation in books containing pedigree family trees of noble families. The `BinaryHeap` data structure described here was first introduced by Williams [**?**].

The randomized `MeldableHeap` data structure described here appears to have first been proposed by Gambin and Malinowski [**?**]. Other meldable heap implementations exist, including leftist heaps [**?**, **?**, Section 5.3.2], binomial heaps [**?**], Fibonacci heaps [**?**], pairing heaps [**?**], and skew heaps [**?**], although none of these are as simple as the `MeldableHeap` structure.

Some of the above structures also support a `decreaseKey(u,y)` operation in which the value stored at node `u` is decreased to `y`. (It is a precondition that $y \leq u.x$.) In most of the preceding structures, this operation can be supported in $O(\log n)$ time by removing node `u` and adding `y`. However, some of these structures can implement `decreaseKey(u,y)` more efficiently. In particular, `decreaseKey(u,y)` takes $O(1)$ amortized time in Fibonacci heaps and $O(\log\log n)$ amortized time in a special version of pairing heaps [**?**]. This more efficient `decreaseKey(u,y)` operation has applications in speeding up several graph algorithms, including Dijkstra's shortest path algorithm [**?**].

Exercise 8.1. Illustrate the addition of the values 7 and then 3 to the `BinaryHeap` shown at the end of Figure 8.2.

Exercise 8.2. Illustrate the removal of the next two values (6 and 8) on the `BinaryHeap` shown at the end of Figure 8.3.

Exercise 8.3. Implement the `remove(i)` method, that removes the value stored in `a[i]` in a `BinaryHeap`. This method should run in $O(\log n)$ time. Next, explain why this method is not likely to be useful.

Exercise 8.4. A d-ary tree is a generalization of a binary tree in which each internal node has d children. Using Eytzinger's method it is also possible to represent complete d-ary trees using arrays. Work out the

equations that, given an index `i`, determine the index of `i`'s parent and each of `i`'s d children in this representation.

Exercise 8.5. Using what you learned in Exercise 8.4, design and implement a *DaryHeap*, the d-ary generalization of a `BinaryHeap`. Analyze the running times of operations on a `DaryHeap` and test the performance of your `DaryHeap` implementation against that of the `BinaryHeap` implementation given here.

Exercise 8.6. Illustrate the addition of the values 17 and then 82 in the `MeldableHeap` `h1` shown in Figure 8.4. Use a coin to simulate a random bit when needed.

Exercise 8.7. Illustrate the removal of the next two values (4 and 8) in the `MeldableHeap` `h1` shown in Figure 8.4. Use a coin to simulate a random bit when needed.

Exercise 8.8. Implement the `remove(u)` method, that removes the node `u` from a `MeldableHeap`. This method should run in $O(\log n)$ expected time.

Exercise 8.9. Show how to find the second smallest value in a `BinaryHeap` or `MeldableHeap` in constant time.

Exercise 8.10. Show how to find the kth smallest value in a `BinaryHeap` or `MeldableHeap` in $O(k \log k)$ time. (Hint: Using another heap might help.)

Exercise 8.11. Suppose you are given `k` sorted lists, of total length `n`. Using a heap, show how to merge these into a single sorted list in $O(n \log k)$ time. (Hint: Starting with the case $k = 2$ can be instructive.)

Hitro urejanje ali*quicksort* algoritem je še en klasični "deli in vladaj" algoritem. V nasprotju z algoritmom zlivanja (mergesort), kateri združuje po 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. Dobimo ga iz a; Particija a je sestavljena iz sklopa elementov manjših kot x, sklopa elementov enakih kot x in niz elementov večjih od x; na koncu pa rekurzivno razvrstimo prvi in tretji sklop števil v tej particiji. Primer je prikazan na sliki 11.3.

Slika 11.3: Primer izvedbe algoritma hitrega urejanja (a, 0, 14, c)

Vse to je narejeno v enem koraku, tako da namesto ustvarjanja kopij urejenih podseznamov, quickSort(a, i, n, c) metoda razvršča samo podseznam a[i],..., a[i + n - 1]. Prvotno kli čemo to metodo kot quickSort(a, 0, a.length, c).

V središču quicksort algoritma je algoritem delitve na mestu. Ta algoritem, brez uporabe dodatnega prostora, zamenja elemente v a in izračuna indekse p in q tako da:

$$a[i]\{$$

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). V vsakem koraku element na položaju j je premaknjen levo na prvo mesto, ali pa je premaknjen na zadnje mesto. V prvih dveh primerih, je j povečan, v zadnjem primeru j ni povečan, zato ker nov element na položaju j še ni bil obdelan.

Quicksort algoritem je zelo tesno povezan z naključnim binarnim iskalnim drevesom, opisan v poglavju 7.1. Dejansko, če poženemo quicksort algoritem nad n različnimi elementi, potem je quicksort rekurzivno drevo naključno iskalno drevo. Da bi to videli, se moramo spomniti, kako gradimo naključno binarno iskalno drevo. Najprej naključno izberemo element x in ga postavimo za koren drevesa. Takoj

za tem, vsak naslednji element primerjamo z x-om. Manjše elemente postavljamo v levo stran poddrevesa in večje elemente v desno stran poddrevesa.

S tem algoritmom, izberemo nakjučni element x in takoj za tem primerjamo vse s tem x-om. Najmanjše elemente postavimo na začetek polja in večje elemente postavimo na konec polja. Quicksort algoritem nato rekurzivno uredi začetek in konec polja, medtem ko naključno iskalno drevo rekurzivno vstavi manjše elemente v levo poddrevo korena in večje elemente v desno poddrevo korena.

Zgornje ujemanje med naključnim binarnim iskalnim drevesom in algoritmom hitrega urejanja, lahko uporabimo za Lemo 7.1

Lemma 11.1. *Ko kličemo algoritem quicksort za urejanje polja, ki vsebuje cela števila* $0, ..., n-1$ *pričakovano število primerjav elementa s pivot elementom je* $H_{i+1} + H^{n-i}$.

Malo seštevanja harmoničnih števil nam daje 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* 2nlnn +O(n).

Proof. Naj bo T število primerjav opravljenih z algoritmom quicksort, ko razvrš ča n razlčne elemente. Z uporabo Leme 11,1, imamo:

Izrek 11.3 opisuje primer, kjer so razvrš čeni elementi vsi različni. Ko vhodna matrika, 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 element pivot a, vse pojavitve x-a se združijo in jih kasneje ne vključimo v enem od dveh podproblemov.

Izrek 11.3. *Quicksort(a, c) metoda ima pričakovani čas izvedbe O (nlogn) in pričakovano število primerjav, ki jih opravi, je v večini* $2n \ln n + O$ (n).

Poglavje 9

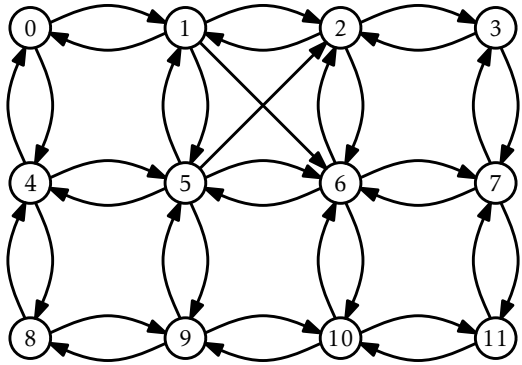
Grafi

V tem poglavju se učimo dva načina za predstavitve grafov in algoritmov, ki uporabljajo te predstavitve.

Matematično, *(usmerjen) graf* je par $G = (V, E)$ kjer V množica *vozlišč* in E je množica urejenih parov vozlišč imenovanih *robovi*. Rob (`i`, `j`) je *usmerjen* od `i` do `j`; `i` se imenuje *vir* množice in `j` se imenuje *tarča*. *Pot* v G je zaporedje vozlišč $v_0, \ldots, v_k$ tako, da za vsak $i \in \{1, \ldots, k\}$, roba (v_{i-1}, v_i) je v E. Pot $v_0, \ldots, v_k$ je *cikel* če, dodatno še imamo pot (v_k, v_0), ki je v E. Pot (ali cikel) je *edinstven* če tudi njegova vozlišča so edinstvena. Če je pot iz neke točke v_i do neke točke v_j, potem pravimo, da v_j je *dosegljiva* iz v_i. Primer grafa je prikazan na sliki Figure 9.1.

Zaradi svoje zmogljivosti pri izdelavi modela raznih pojavov, grafi imajo ogromno število aplikacij. Obstajajo številni primeri. Računalniška omrežja lahko modeliramo v nek graf, kjer vozlišča (točke) predstavljajo računalnike in robovi predstavljajo (direktno) komunikacijsko pot med dvema računalnikoma. Tudi ceste v nekem mestu lahko predstavimo kot neki graf, kjer vozlišča predstavljajo križišča in robovi 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 lahko imamo *konfliktni graf* koledarja kjer vozlišča predstavljajo predavanja v univerzah in rob (`i`, `j`) obstaja samo v primeru če je prisoten vsaj en študent, ki je prevzel razred `i` in razred `j`. Tako en rob prikaže, da izpit za razred `i` ne more na noben način biti načrtovan ob istem času tudi za razred `j`.



Slika 9.1: A graph with twelve vertices. Vertices are drawn as numbered circles and edges are drawn as pointed curves pointing from source to target.

V tem poglavju nam `n` predstavlja število vozlišč v množici G in `m` število robov v množici G. To pomeni, da $\texttt{n} = |V|$ in $\texttt{m} = |E|$. Poleg vsega tega pa predpostavljamo, da je $V = \{0, \ldots, \texttt{n} - 1\}$. Za katerekoli druge podatke, ki bi radi povezali z elementi, ki se nahajajo v množici V, lahko le-te shranimo v neko tabelo dolžine `n`.
Značilne operacije, ki opravljamo nad grafe so:

- `addEdge(i, j)`: Dodaj rob `(i, j)` v E.
- `removeEdge(i, j)`: Zbriši rob `(i, j)` iz E.
- `hasEdge(i, j)`: Poišči rob $\texttt{(i, j)} \in E$
- `outEdges(i)`: Vrne `List` (seznam) celih števil `j` od $\texttt{(i, j)} \in E$
- `inEdges(i)`: Vrne `List` (seznam) celih števil `j` od $\texttt{(j, i)} \in E$

Vedeti je treba, da take operacije ni težko implementirati na unčikovit način. Na primer, prve tri operacije lahko so uporabljene direktno z uporabo `USet`, na tak način, da se lahko izvajajo v konstantnem pričakovanem času z uporabo razpršenih tabel (predstavljeni v poglavju Chapter **??**). Zadnje dve operaciji so lahko implementirane v konstantnem času s shranjevanjem, za vsako vozlišče shranjujemo še seznam sosednjih vozlišč.

Vendar, različne aplikacije grafov zahtevajo različna delovanja teh operacij in v idealnem primeru, lahko uporabljamo aplikacijo, ki je najbolj enostavna in lahko zadovoljuje vse rekvizite aplikacije. Zaradi tega razpravljamo o dveh velikih kategorij za predstavljanje grafov.

9.1 AdjacencyMatrix: Predstavitev grafov z uporabo matrik

Matrika sosednosti je način za predstaviti n vozlišč grafa $G = (V, E)$ iz ene matrike $n \times n$, a, kjer notranji elementi imajo vrednosti tipa boolean.

AdjacencyMatrix

```
int n;
bool **a;
```

Vnos elemnta matrike a[i][j] je definiran kot

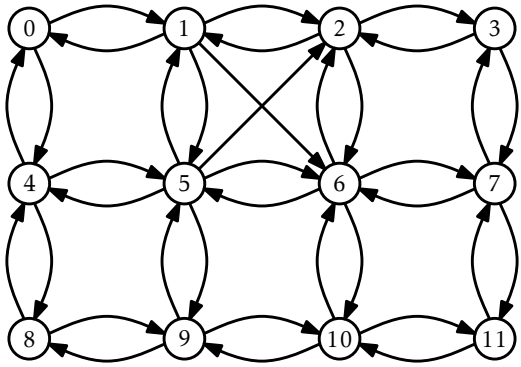
$$\texttt{a[i][j]} = \begin{cases} \texttt{true} & \text{if } (\texttt{i}, \texttt{j}) \in E \\ \texttt{false} & \text{otherwise} \end{cases}$$

Matrika sosednosti za graf iz slike Figure 9.1 je prikazana na sliki Figure 9.2.
Tu je prikazana operacija addEdge(i, j), removeEdge(i, j) in hasEdge(i, j), ki samo 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];
}
```

Te operacije nedvoumno vzamejo konstanten čas po operaciji.
Where the adjacency matrix performs poorly is with the outEdges(i) and inEdges(i) operations. Da bi implementirali le teh, je treba preveriti



	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 9.2: A graph and its adjacency matrix.

vse n vnose v ustrezno vrstico ali stolpec iz a in je treba zbrati vse indeks j, kjer a[i][j] oziroma a[j][i] je vrednost TRUE.

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);
}
```

Take operacije očitno nam vzamejo $O(n)$ časa po operaciji.
Druga slaba lastnost matrike sosednoste je ta, da je velika. V matriki je shranjeno $n \times n$ boolean vrednosti, kar pomeni, da nam rabi najmanj n^2 bitov prostora v pomnilniku. Implementacija tu uporablja dejansko eno matriko z vrednosti in to na tak način, da uporablja efektivno vrednosti n^2 zlogov pomnilnika. Za bolj previdno implementacijo, katera zapakira w boolean vrednosti v vsako besedo pomnilnik. Tako bi zmanjšali porabo prostora in tako dobili $O(n^2/w)$.

Theorem 9.1. *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)` *in constant time per operation; and*
- `inEdges(i)`, *and* `outEdges(i)` *in* $O(n)$ *time per operation.*

The space used by an `AdjacencyMatrix` *is* $O(n^2)$.

Kljub visoke zahteve po prosturu in za ne unčikovitega delovanja vhoda `inEdges(i)` in izhoda `outEdges(i)` operacije, `AdjacencyMatrix` je lahko še vedno uporabna za nekatere operacije. Še posebno, ko graf G je gost (*dense*), kar pomeni, da ima nekje okoli n^2 robov, potem zavzeti n^2 prostora je sprejemljivo.
Podatkovna struktura `AdjacencyMatrix` se pogosto uporavlja, saj operacije nad matriko a se lahko uporabljajo za definirati lastnosti grafa

G. To je argument, ki se predela na tečaju za algoritme, ampak si oglejmo vsaj eno lastnost: če obravnavamo vhod kot neko celo število a (integer: 1 za true in 0 za false) in pomnožimo matriko a s samo seboj z uporabo operacije množenje matrik, potem kot rezultat bomo dobili matriko a^2. Po definiciji za množenje matrik

$$a^2[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot a[k][j] \ .$$

Po razlagi te vsote glede na graf G, ta formula prešteje število vozlišč, k, tako, da G vsebuje oba robova (i,k) in (k, j). Bolj natančno povedano, se šteje število poti od i do j (preko vmestnih vozlišč k) kjer dolžina je natanko dve. Taka ugotovitev je fondamentalna za algoritme, ki izračunavajo najkrajšo pot med vsemi pari vozlišč v G, ki uporablja samo $O(\log n)$ množenja matrik.

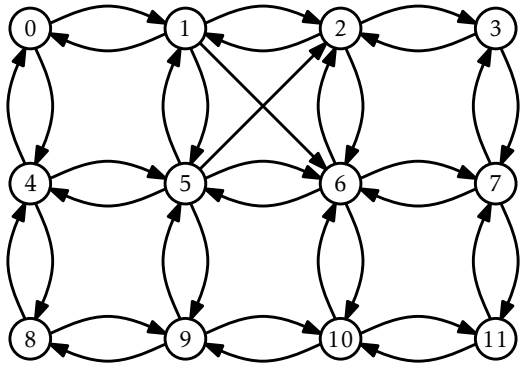
9.2 AdjacencyLists: A Graph as a Collection of Lists

Seznam sosednosti - ponazoritev grafov vzame pristop bolj usmerjen v vozlišča. Obstaja veliko možnih izvedb seznamov sosednosti. V tem poglavju predstavljamo preprosto izvedbo. Na koncu odseka, razpravljamo o različnih možnostih. V seznamu sosednosti je graf $G = (V, E)$ predstavljen kot polje, adj, seznamov. Seznam adj[i] vsebuje seznam vseh vozlišč sosednjih vozlišču i. Vsebuje vsak j tako, da $(i, j) \in E$.

AdjacencyLists
```
int n;
List *adj;
```

(Primer je pokazan v Figure 9.3.) V tej specifični implementaciji, pokažemo vsak seznam adj kot a subclass of ArrayStack, ker želimo doseči konstanten čas dostopov do pozicij. Mogoče so tudi drugačne opcije. Ena opcija je implementiranje adj kot DLList.
Operacija addEdge(i, j) doda vrednost j seznamu 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 9.3: 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 $O(\deg(\texttt{i}))$ času, kjer $\deg(\texttt{i})$ (*stopnja* `i` -ja) prešteje število robov v E, ki imajo `i` za njihov vir.
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 $O(\deg(\texttt{i}))$ 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 $O(\deg(\texttt{i}))$ času.
Operacija `inEdges(i)` je veliko več dela. Operacija pogleda vsako vozlišče j če obstaja `(i, j)` in, če tako, doda `j` v 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 $O(n+m)$ času.
Naslednji izrek povzema delovanje zgornje podatkovne strukture:

Theorem 9.2. *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 array-based list, linked-list, ali celo hashtable.
- Lahko bi uporabili drug seznam sosednosti, `inadj`, ki hrani za vsak `i`, seznam vozlišč `j`, tako da $(j,i) \in E$ Zo lahko močno poveča učinkovitost operacije `inEdges(i)`, ampak rahlo zmanjša učinkovitost dodajanja in brisanja robov.
- Lahko bi vpis za rob `(i, j)` v `adj[i]` bil povezan z referenco na ustrezni vpis v `inadj[j]`
- Lahko bi robovi bili prvorazredni objekti z njihovimi asociativnimi podatki Tako bi `adj` vseboval seznam robov namesto seznama vozlišč (integers).

Pri večini gornjih vprašanj pride do kompromisa med kompleksnostjo (in prosotorom) implementacije in uspešnostjo funkcij implementacije.

9.3 Graph Traversal

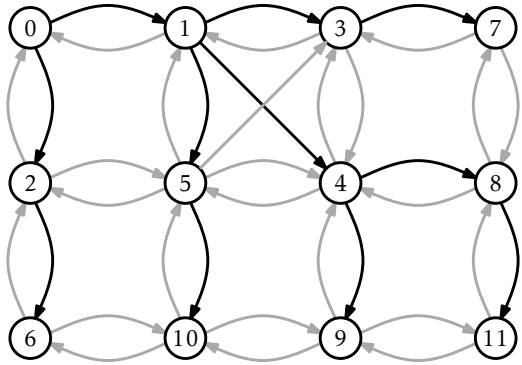
In this section we present two algorithms for exploring a graph, starting at one of its vertices, `i`, and finding all vertices that are reachable from `i`. Both of these algorithms are best suited to graphs represented using an adjacency list representation. Therefore, when analyzing these algorithms we will assume that the underlying representation is an `AdjacencyLists`.

9.3.1 Breadth-First Search

The *bread-first-search* algorithm starts at a vertex `i` and visits, first the neighbours of `i`, then the neighbours of the neighbours of `i`, then the neighbours of the neighbours of the neighbours of `i`, and so on.
This algorithm is a generalization of the breadth-first traversal algorithm for binary trees (Section 5.1.2), and is very similar; it uses a queue, `q`, that initially contains only `i`. It then repeatedly extracts an element from `q` and adds its neighbours to `q`, provided that these neighbours have never been in `q` before. The only major difference between the breadth-first-search algorithm for graphs and the one for trees is that the algorithm for graphs has to ensure that it does not add the same vertex to `q` more than once. It does this by using an auxiliary boolean array, `seen`, that tracks which vertices have already been discovered.

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 9.4: An example of bread-first-search starting at node 0. Nodes are labelled with the order in which they are added to q. Edges that result in nodes being added to q are drawn in black, other edges are drawn in grey.

```
    }
  }
}
delete[] seen;
```

An example of running bfs(g,0) on the graph from Figure 9.1 is shown in Figure 9.4. Different executions are possible, depending on the ordering of the adjacency lists; Figure 9.4 uses the adjacency lists in Figure 9.3.

Analyzing the running-time of the bfs(g, i) routine is fairly straightforward. The use of the seen array ensures that no vertex is added to q more than once. Adding (and later removing) each vertex from q takes constant time per vertex for a total of $O(n)$ time. Since each vertex is processed by the inner loop at most once, each adjacency list is processed at most once, so each edge of G is processed at most once. This processing, which is done in the inner loop takes constant time per iteration, for a total of $O(m)$ time. Therefore, the entire algorithm runs in $O(n+m)$ time.

The following theorem summarizes the performance of the bfs(g, r) algorithm.

Theorem 9.3. *When given as input a* `Graph`, `g`, *that is implemented using the* `AdjacencyLists` *data structure, the* `bfs(g,r)` *algorithm runs in* $O(n+m)$ *time.*

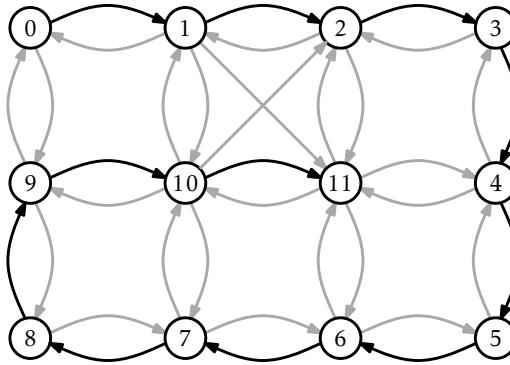
A breadth-first traversal has some very special properties. Calling `bfs(g,r)` will eventually enqueue (and eventually dequeue) every vertex `j` such that there is a directed path from `r` to `j`. Moreover, the vertices at distance 0 from `r` (`r` itself) will enter `q` before the vertices at distance 1, which will enter `q` before the vertices at distance 2, and so on. Thus, the `bfs(g,r)` method visits vertices in increasing order of distance from `r` and vertices that cannot be reached from `r` are never visited at all.
A particularly useful application of the breadth-first-search algorithm is, therefore, in computing shortest paths. To compute the shortest path from `r` to every other vertex, we use a variant of `bfs(g,r)` that uses an auxilliary array, `p`, of length `n`. When a new vertex `j` is added to `q`, we set `p[j] = i`. In this way, `p[j]` becomes the second last node on a shortest path from `r` to `j`. Repeating this, by taking `p[p[j]`, `p[p[p[j]]]`, and so on we can reconstruct the (reversal of) a shortest path from `r` to `j`.

9.3.2 Depth-First Search

The *depth-first-search* algorithm is similar to the standard algorithm for traversing binary trees; it first fully explores one subtree before returning to the current node and then exploring the other subtree. Another way to think of depth-first-search is by saying that it is similar to breadth-first search except that it uses a stack instead of a queue. During the execution of the depth-first-search algorithm, each vertex, `i`, is assigned a colour, `c[i]`: `white` if we have never seen the vertex before, `grey` if we are currently visiting that vertex, and `black` if we are done visiting that vertex. The easiest way to think of depth-first-search is as a recursive algorithm. It starts by visiting `r`. When visiting a vertex `i`, we first mark `i` as `grey`. Next, we scan `i`'s adjacency list and recursively visit any white vertex we find in this list. Finally, we are done processing `i`, so we colour `i` black and return.

Algorithms

```
 dfs(Graph &g, int i, char *c) {
c[i] = grey;  // currently visiting i
```



Slika 9.5: An example of depth-first-search starting at node 0. Nodes are labelled with the order in which they are processed. Edges that result in a recursive call are drawn in black, other edges are drawn in grey.

```
ArrayStack<int> edges;
g.outEdges(i, edges);
for (int k = 0; k < edges.size(); k++) {
  int j = edges.get(k);
  if (c[j] == white) {
    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;
```

An example of the execution of this algorithm is shown in Figure 9.5. Although depth-first-search may best be thought of as a recursive algorithm, recursion is not the best way to implement it. Indeed, the code given above will fail for many large graphs by causing a stack overflow. An alternative implementation is to replace the recursion stack with an explicit stack, s. The following implementation does just that:

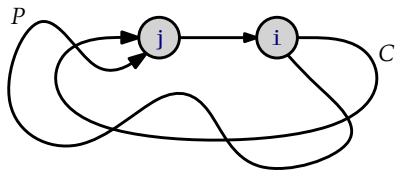
Algorithms

```
 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;
```

In the preceding code, when the next vertex, `i`, is processed, `i` is coloured `grey` and then replaced, on the stack, with its adjacent vertices. During the next iteration, one of these vertices will be visited.
Not surprisingly, the running times of `dfs(g,r)` and `dfs2(g,r)` are the same as that of `bfs(g,r)`:

Theorem 9.4. *When given as input a `Graph`, `g`, that is implemented using the `AdjacencyLists` data structure, the `dfs(g,r)` and `dfs2(g,r)` algorithms each run in $O(n+m)$ time.*

As with the breadth-first-search algorithm, there is an underlying tree associated with each execution of depth-first-search. When a node $i \neq r$ goes from `white` to `grey`, this is because `dfs(g,i,c)` was called recursively while processing some node `i'`. (In the case of `dfs2(g,r)` algorithm, `i` is one of the nodes that replaced `i'` on the stack.) If we think of `i'` as the parent of `i`, then we obtain a tree rooted at `r`. In Figure 9.5, this tree is a path from vertex 0 to vertex 11.
An important property of the depth-first-search algorithm is the following: Suppose that when node `i` is coloured `grey`, there exists a path from `i` to some other node `j` that uses only white vertices. Then `j` will be coloured first `grey` then `black` before `i` is coloured `black`. (This



Slika 9.6: The depth-first-search algorithm can be used to detect cycles in G. The node `j` is coloured `grey` while `i` is still `grey`. This implies that there is a path, P, from `i` to `j` in the depth-first-search tree, and the edge `(j,i)` implies that P is also a cycle.

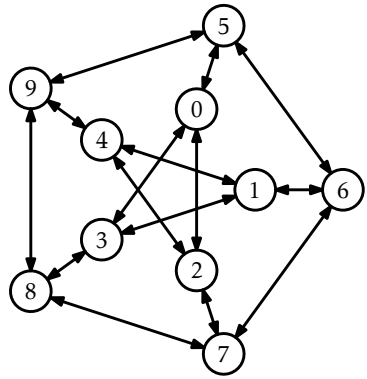
can be proven by contradiction, by considering any path P from `i` to `j`.) One application of this property is the detection of cycles. Refer to Figure 9.6. Consider some cycle, C, that can be reached from `r`. Let `i` be the first node of C that is coloured `grey`, and let `j` be the node that precedes `i` on the cycle C. Then, by the above property, `j` will be coloured `grey` and the edge `(j,i)` will be considered by the algorithm while `i` is still `grey`. Thus, the algorithm can conclude that there is a path, P, from `i` to `j` in the depth-first-search tree and the edge `(j,i)` exists. Therefore, P is also a cycle.

9.4 Discussion and Exercises

The running times of the depth-first-search and breadth-first-search algorithms are somewhat overstated by the Theorems 9.3 and 9.4. Define n_r as the number of vertices, `i`, of G, for which there exists a path from `r` to `i`. Define m_r as the number of edges that have these vertices as their sources. Then the following theorem is a more precise statement of the running times of the breadth-first-search and depth-first-search algorithms. (This more refined statement of the running time is useful in some of the applications of these algorithms outlined in the exercises.)

Theorem 9.5. *When given as input a* `Graph`, `g`, *that is implemented using the* `AdjacencyLists` *data structure, the* `bfs(g,r)`, `dfs(g,r)` *and* `dfs2(g,r)` *algorithms each run in* $O(n_r + m_r)$ *time.*

Breadth-first search seems to have been discovered independently by



Slika 9.7: An example graph.

Moore [**?**] and Lee [**?**] in the contexts of maze exploration and circuit routing, respectively.
Adjacency-list representations of graphs were presented by Hopcroft and Tarjan [**?**] as an alternative to the (then more common) adjacency-matrix representation. This representation, as well as depth-first-search, played a major part in the celebrated Hopcroft-Tarjan planarity testing algorithm that can determine, in $O(\mathtt{n})$ time, if a graph can be drawn, in the plane, and in such a way that no pair of edges cross each other [**?**].
In the following exercises, an undirected graph is one in which, for every `i` and `j`, the edge (`i`, `j`) is present if and only if the edge (`j`, `i`) is present.

Exercise 9.1. Draw an adjacency list representation and an adjacency matrix representation of the graph in Figure 9.7.

Exercise 9.2. The *incidence matrix* representation of a graph, G, is an $\mathtt{n} \times \mathtt{m}$ matrix, A, where

$$A_{i,j} = \begin{cases} -1 & \text{if vertex } i \text{ the source of edge } j \\ +1 & \text{if vertex } i \text{ the target of edge } j \\ 0 & \text{otherwise.} \end{cases}$$

1. Draw the incident matrix representation of the graph in Figure 9.7.

2. Design, analyze and implement an incidence matrix representation of a graph. Be sure to analyze the space, the cost of `addEdge(i, j)`, `removeEdge(i, j)`, `hasEdge(i, j)`, `inEdges(i)`, and `outEdges(i)`.

Exercise 9.3. Illustrate an execution of the `bfs(G, 0)` and `dfs(G, 0)` on the graph, `G`, in Figure 9.7.

Exercise 9.4. Let G be an undirected graph. We say G is *connected* if, for every pair of vertices `i` and `j` in G, there is a path from `i` to `j` (since G is undirected, there is also a path from `j` to `i`). Show how to test if G is connected in $O(n+m)$ time.

Exercise 9.5. Let G be an undirected graph. A *connected-component labelling* of G partitions the vertices of G into maximal sets, each of which forms a connected subgraph. Show how to compute a connected component labelling of G in $O(n+m)$ time.

Exercise 9.6. Let G be an undirected graph. A *spanning forest* of G is a collection of trees, one per component, whose edges are edges of G and whose vertices contain all vertices of G. Show how to compute a spanning forest of of G in $O(n+m)$ time.

Exercise 9.7. We say that a graph G is *strongly-connected* if, for every pair of vertices `i` and `j` in G, there is a path from `i` to `j`. Show how to test if G is strongly-connected in $O(n+m)$ time.

Exercise 9.8. Given a graph $G = (V, E)$ and some special vertex $r \in V$, show how to compute the length of the shortest path from `r` to `i` for every vertex $i \in V$.

Exercise 9.9. Give a (simple) example where the `dfs(g, r)` code visits the nodes of a graph in an order that is different from that of the `dfs2(g, r)` code. Write a version of `dfs2(g, r)` that always visits nodes in exactly the same order as `dfs(g, r)`. (Hint: Just start tracing the execution of each algorithm on some graph where `r` is the source of more than 1 edge.)

Exercise 9.10. A *universal sink* in a graph G is a vertex that is the target of $n-1$ edges and the source of no edges.[1] Design and implement an

[1] A universal sink, `v`, is also sometimes called a *celebrity*: Everyone in the room recognizes `v`, but `v` doesn't recognize anyone else in the room.

algorithm that tests if a graph G, represented as an `AdjacencyMatrix`, has a universal sink. Your algorithm should run in $O(n)$ time.

Poglavje 10

Podatkovne strukture za cela števila

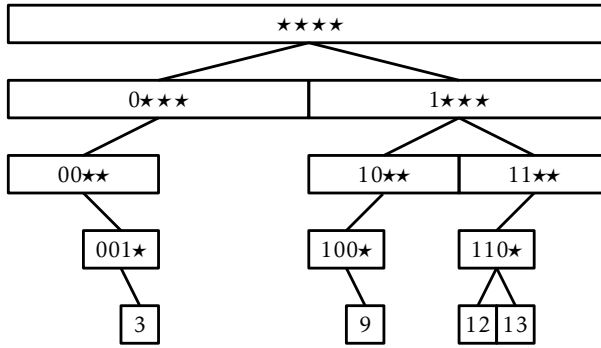
V tem poglavju se bomo vrnili k problemu implementiranja SSet-a. Razlika v implementaciji je ta, da zdaj privzamemo, da so elementi shranjeni v SSet-u, w-bitna cela števila. To pomeni da hočemo implementirati metode add(x), remove(x) in find(x), kjer velja da $x \in \{0, \ldots, 2^w - 1\}$. Če malo pomislimo 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



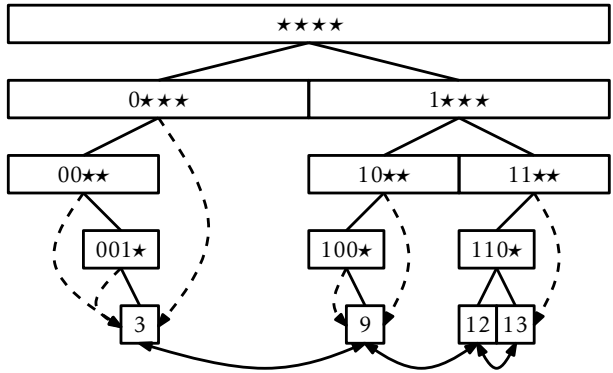
Slika 10.1: Cela števila shranjena v binary trie so zakodirana kot poti od korena do lista.

predstavljen 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.

10.1 `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. Figure 10.1 prikazuje primer, ko je $w = 4$, in trie shranjuje cela števila 3(0011), 9(1001), 12(1100), in 13(1101).

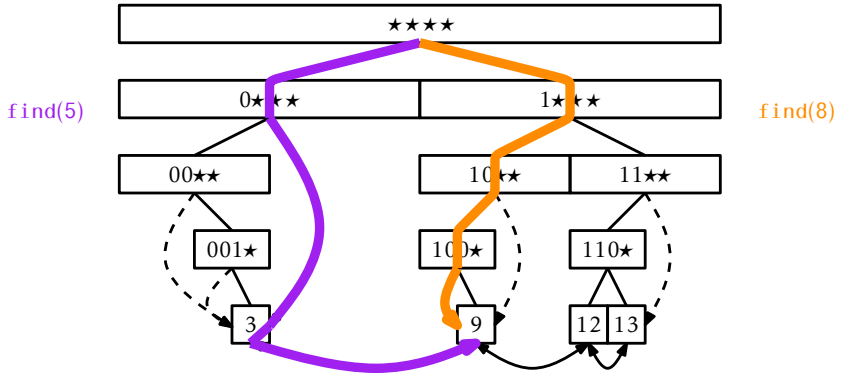
Ker iskalna pot za vrednost `x` odvisi od bitov `x`-a, nam bo koristilo, če otroka vozlišča poimenujemo `u`, `u.child`[0] (`left`) in `u.child`[1] (`right`). Tile kazalci na otroke bodo pravzaprav služili dvema namenoma. Ker listi v binary trie nimajo nobenega otroka, so kazalci uporabljeni za povezavo 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]



Slika 10.2: `BinaryTrie` z `jump` kazalci, prikazanami kot prekinjene ukrivljene povezave.

(`next`) je vozlišče, ki sledi `u`-ju v seznamu. Posebno vozlišče `dummy`, je uporabljeno pred prvim vozliščem in za zadnjim vozliščem v seznamu. (glej Section 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 Figure 10.2.

`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 Figure 10.3.



Slika 10.3: Poti po katerih gre `find(5)` in `find(8)`.

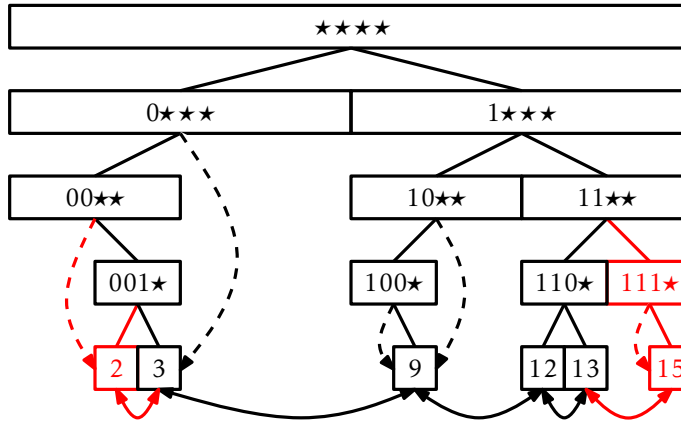
BinaryTrie

```
T find(T x) {
  int i, c = 0;
  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`.



Slika 10.4: Dodajanje vrednosti 2 in 15 v `BinaryTrie` na Figure 10.2.

3. Vozlišče u', ki vsebuje x, se doda povezanemu seznamu listov (metoda ima dostop do prednika, `pred`, u'-ja v povezanem seznamu 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 Figure 10.4.

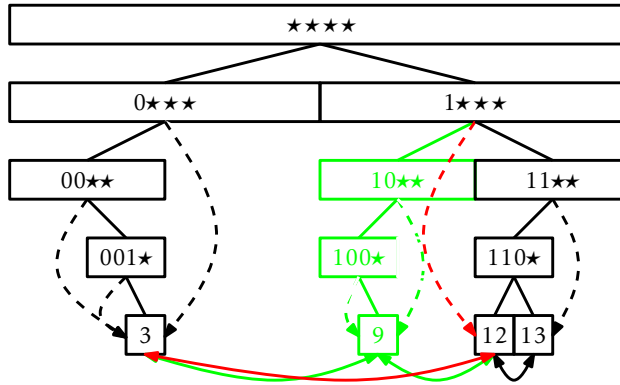
BinaryTrie

```
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 10.5: Odstranjevanje vrednosti 9 iz `BinaryTrie` na Figure 10.2.

Odstranjevanje je prikazano na Figure 10.5.

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;
}
```

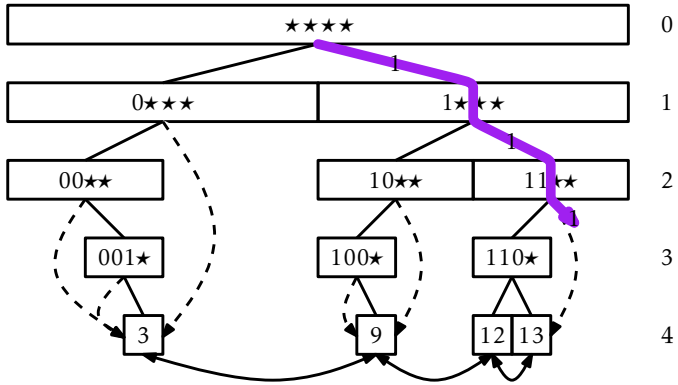
Theorem 10.1. *BinaryTrie implementira SSet vmesnik za w-bitna cela števila. BinaryTrie podpira operacije* add(x), remove(x) *in* find(x) *v časovni kompleksnosti* $O(w)$ *na operacijo. Prostor, ki ga BinaryTrie uporablja za shranjevanje* n *vrednosti je* $O(n \cdot w)$.

10.2 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 Figure 10.6; na sliki je zadnje vozlišče u na iskalni poti za število 14 (katerga binarni



Slika 10.6: Ker na sliki ni vozlišča označenega z 111★ se iskalna pot za 14 (1110) konča pri vozlišču 11★★ .

zapis je 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 (Chapter **??**). 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`:

Theorem 10.2. *A `XFastTrie` implementira `SSet` vmesnik za `w`-bitna cela števila. `XFastTrie` podpira operacije*

- `add(x)` *in* `remove(x)` *v* $O(w)$ *času za operacijo in*

- `find(x)` *v* $O(\log w)$ *času za operacijo*

Prostorska zahtevnost `XFastTrie`*, ki shrani* n *vrednosti je* $O(n \cdot w)$.

10.3 YFastTrie: Dvokratni-Logaritmični Čas SSet

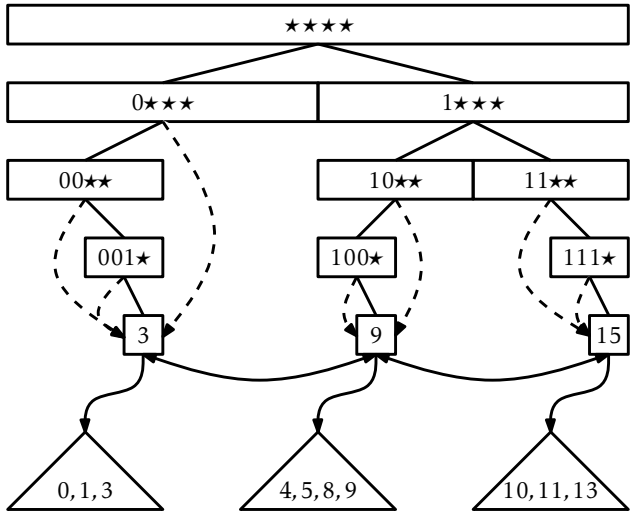
`XFastTrie` je velika, celo eksponentna, izboljšava `BinaryTrie` v kategoriji poizvedbenega časa, vendar operaciji `add(x)` in `remove(x)` še nista veliko hitrejši. Poleg tega je poraba prostora $O(n \cdot w)$ večja kot pri drugih `SSet` implementacijah, predstavljenih v tej knjigi, ki uporabljajo $O(n)$ 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 *secondary structures*, v tem primeru je to podaljšana verzija treaps (Section 6.2). Obstaja približno n/w takšnih sekundarnih 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 udobja 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 Figure 10.7.

`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 10.7: 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 $E[r] = w$, then $E[\log r] \leq \log w$.

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. Figure 10.8 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;
}
```

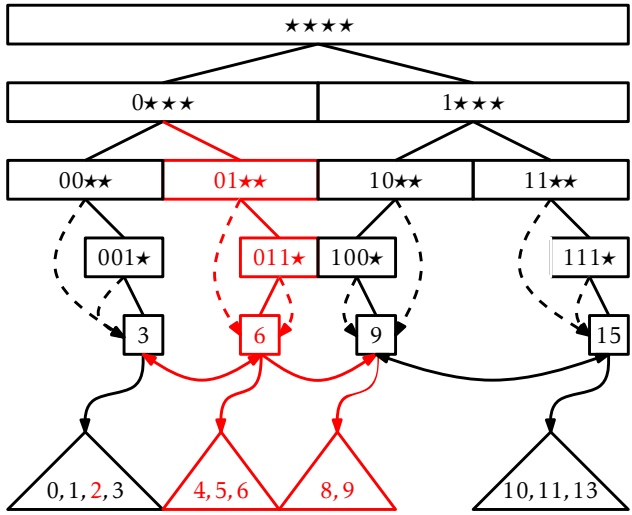
Dodajanje x k t vzame $O(\log w)$ časa. Exercise **??** prikazuje,da je razdelitev t v t1 in t′ lahko narejena v $O(\log w)$ pričakovanem času. Dodajanje para (x,t1) k xft vzame $O(w)$ časa, ampak se zgodi samo z verjetnostjo $1/w$. Zato je, pričakovan čas poteka add(x) operacije

$$O(\log w) + \frac{1}{w} O(w) = O(\log w) \ .$$

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 Figure 10.9.

YFastTrie

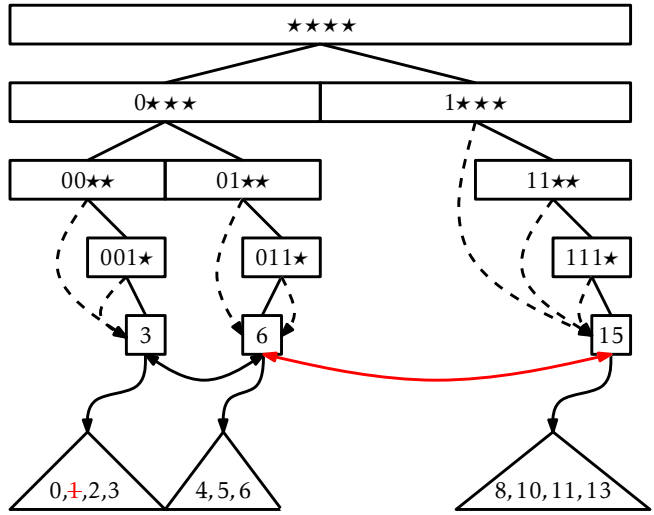
```
bool remove(T x) {
  unsigned ix = intValue(x);
  XFastTrieNode1<YPair<T> > *u = xft.findNode(ix);
  bool ret = u->x.t->remove(x);
```



Slika 10.8: 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, Exercise **??** 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.

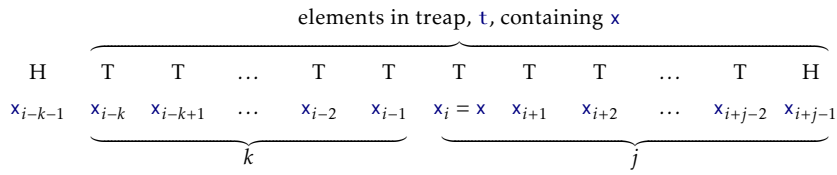


Slika 10.9: Odstranjevanje vrednosti 1 in 9 iz YFastTrie in Figure 10.8.

Lemma 10.1. *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 Figure 10.10. 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 Lemma **??** 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 prejšni odstavek, vendar v tem primeru zadnji met ni bil štet. V povzetku

[2] Ta analiza ingorira dejstvo da j nikoli ne preseže $n - i + 1$. Kakorkoli, to zgolj zmanjša vrednost $E[j]$, zgornja meja pa je še vedno enaka.



Slika 10.10: Število elementov v poddrevesu t, ki vsebujejo x je določeno z metanjem dveh kovancev.

$n_x = j + k$, torej velja

$$\mathrm{E}[n_x] = \mathrm{E}[j + k] = \mathrm{E}[j] + \mathrm{E}[k] \leq 2w - 1 \ . \qquad \square$$

Lemma 10.1 Je zadnji del v dokazu teorema, ki povzema učinkovitost YFastTrie:

Theorem 10.3. *YFastTrie impelementira SSet vmestnik za w-bitna cela števila. YFastTrie podpira operacije* add(x), remove(x), *and* find(x) *v pričakovanem času* $O(\log w)$ *na operacijo. Prostor, ki ga YFastTrie porabi, da hrani* n *vrednosti je* $O(n + w)$.

Clen w v porabi prostora 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 $O(n)$.

10.4 Razprava in vaje

Prvo podaktovno strukturo, ki zagotavlja časovno zahtevnost $O(\log w)$ za operacije dodaj(x), odstrani(x), in najdi(x) je predlagal van Emde Boas in je od takrat poznana kot *van Emde Boas* (ali *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 van Emde Boas drevesom; na primer, hash tabele v XFastTrie nadomestijo matrike v van Emde Boas drevesu. To se pravi, van Emde Boas drevo hrani matriko dolžine 2^i namesto hash 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 na `jdi(x)` operacija izvede v času $O((\log n)/(\log w))$. Z uporabo 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 na `jdi(x)` operacijo 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 stuktur, ki porabijo le $O(n)$ prostora.

Exercise 10.1. Sestavi in implementiraj poenostavljeno različico `BinaryTrie`, ki nima kazalcev povezanega seznama ali skakalnih kazalcev, operacija na `jdi(x)`
pa teče v $O(w)$ času.

Exercise 10.2. Sestavi in izpelji poenostavljeno implementacijo `XFastTrie`, ki ne uporablja binarnega drevesa. Namesto tega naj vaša implementacija vse hrani v dvojno povezanem seznamu in v $w+1$ hash tabelah.

Exercise 10.3. `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 `dodaj(s)`, `odstrani(s)`, in na `jdi(s)` v času sorazmernem dolžini `s`.
Namig: Vsako vozlišče v vaši podatkovni strukturi naj hrani hash tabelo, ki je indeksirana z vrednostjo znaka.

Exercise 10.4. Za število $x \in \{0, \ldots 2^w - 1\}$, kjer $d(x)$ pomeni razliko med x in vrednostjo, ki jo vrne na `jdi(x)` [če na `jdi(x)` vrne `null`, potem določi $d(x)$ kot 2^w]. Na primer, če na `jdi(23)` vrne 43, potem $d(23) = 20$.

1. Sestavi in implementiraj spremenjeno različico operacije na `jdi(x)` v `XFastTrie`, ki se izvaja v času $O(1 + \log d(x))$. Nasvet: Hash table $t[w]$ vsebuje vse vrednosti, x, kot so $d(x) = 0$, torej bi bilo tu najbolje začeti.

2. Sestavi in implementiraj spremenjeno različico operacije na `jdi(x)` v `XFastTrie`, ki se izvaja v času $O(1 + \log\log d(x))$.

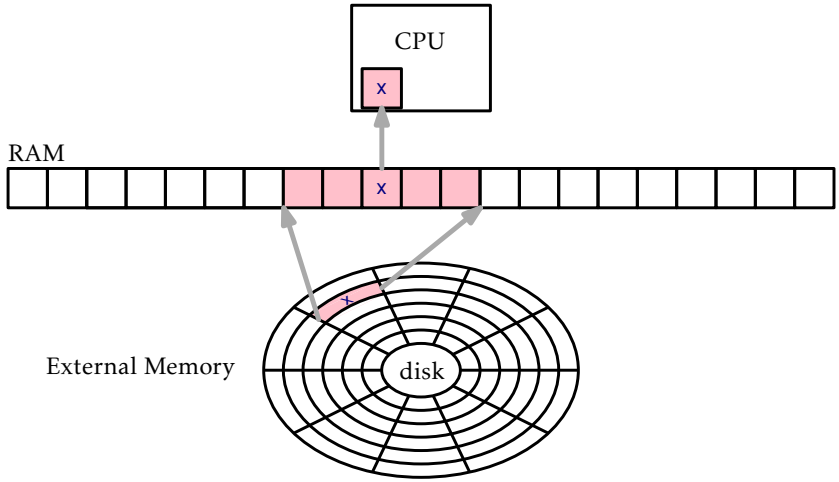
Poglavje 11

Iskanje v zunanjem pomnilniku

Skozi knjigo smo uporabljali w-bitni besedni-RAM model računanja, katerega smo opredelili v Section 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 pomnilnika za njihovo shranjevanje. V takih primerih se mora aplikacija zateči k shranjevanju podatkov na nek zunanji pomnilniški medij, kot je trdi disk, SSD disk ali celo omrežni datotečni strežnik (kateri ima svoje 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 19ms, SSD disk pa ima povprečen čas dostopa 0.3ms. Za primerjavo, bralno-pisalni pomnilnik v računalniku ima povprečen čas dostopa manj kot 0.000113ms. 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 zloga v RAM-u je tisočkrat hitrejše kot dostopanje do naključnega zloga na trdem disku ali SSD disku. Čas dostopa pa vseeno ne pove vsega. Ko dostopamo do zloga 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 zlog, nam disk vrne blok, ki vsebuje 4 096 zlogov. Če našo podatkovno strukturo skrbno organiziramo, to pomeni, da z vsakim dostopom do diska dobimo 4 096 zlogov, ki so nam v pomoč pri



Slika 11.1: V modelu zunanjega pomnilnika, dostop do posameznega elementa x v zunanjem pomnilniku, zahteva branje celotnega bloka, ki vsebuje x, v RAM.

dokončanju operacije.

To je ideja računanja z *modelom zunanjega pomnilnika*, shematsko prikazanega v Figure 12.1. 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 čudno, 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

$$\mathtt{n} \leq 1\,461\,501\,637\,330\,902\,918\,203\,684\,832\,716\,283\,019\,655\,932\,542\,976 \ .$$

11.1 Block Store

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 `BlockStore`. `BlockStore` hrani zbirko spominskih blokov, kjer ima vsak velikost B. Vsak blok je enolično določen s celoštevilskim indeksom. `BlockStore` 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.

`BlockStore` si najlažje predstavljamo tako, da si ga zamislimo kot shranjevanje datoteke na disk, kateri je razdeljen na bloke, kjer vsak vsebuje B zlogov. Na ta način `readBlock(i)` in `writeBlock(i,b)` preprosto bereta in zapisujeta zloge $\mathtt{i}B, \ldots, (\mathtt{i}+1)B-1$ te datoteke. Poleg tega bi preprost `BlockStore` lahko vodil *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.

11.2 B-drevesa

V tem poglavju bomo razpravljali o posplošitvah dvojiških dreves imenovanih B-drevesa, ki so učinkovita predvsem v zunanjem

spominskem modelu. Alternativno se na B-drevesa lahko gleda, kot na posplošitev 2-4 dreves, ki so opisana v Section 7.1. (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 od vozlišča `u` so shranjeni v polju, `u.otroci`. 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 ℓ, listov v B-drevesu izpolnjuje naslednji neenakosti:

$$2B^{h-1} \leq \ell \leq 2(2B)^{h-1} \ .$$

Vzamemo logaritem iz prve neenakosti in preuredimo. Dobimo:

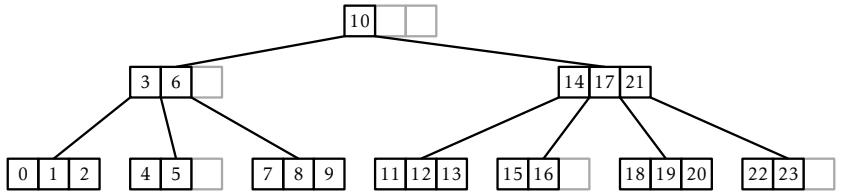
$$\begin{aligned} h &\leq \frac{\log \ell - 1}{\log B} + 1 \\ &\leq \frac{\log \ell}{\log B} + 1 \\ &= \log_B \ell + 1 \ . \end{aligned}$$

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.otroci`$[$`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 11.2: B-drevo, $B = 2$.

Primer B-drevesa z $B = 2$ je prikazan na sliki Figure **??**.
Upoštevajte, da so podatki shranjeni v vozliščih B-drevesa velikosti $O(B)$. Zato, je v nastavitvah zunanjega spomina, vrednost B v B-drevesu določena tako, da celotno vozlišče lahko ustreza enemu zunanje spominskemu 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

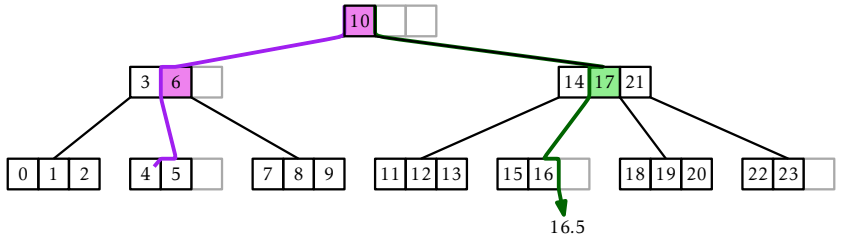
$$(4 + 4) \times 2B = 8 \times 512 = 4096$$

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, kateri 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 struktruri:

BTree
```
int n; // number of elements stored in the tree
int ri; // index of the root
BlockStore<Node*> bs;
```

11.2.1 Searching

Implementacija operacije na `jdi(x)`, ki je ilustrirana v Figure 12.3, je posplošitev operacije na `jdi(x)` v dvojiškem iskalnem drevesu. Iskanje

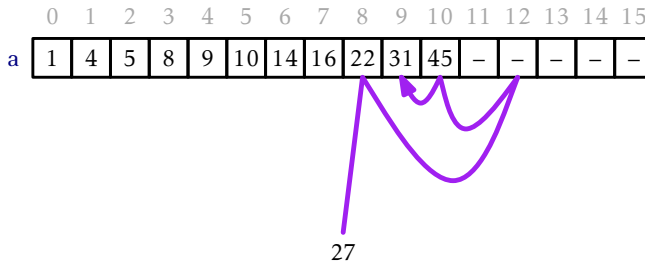


Slika 11.3: Uspešno iskanje (vrednosti 4) in neuspešno iskanje (za vrednost 16.5) v *B*-drevesu. Obarvana vozlišča predstavljajo, kje se je vrednost med iskanjem `z`ja spremenila.

`x`-a se začne v korenu. 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.otrici[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 na `jdi(x)` je metoda na `jdiIt(a,x)`, ki išče v `null`-napolnjeno urejeno polje, `a`, vrednost `x`. Ta metoda, predstavljena



Slika 11.4: The execution of `findIt(a,27)`.

v Figure 12.4, deluje za vsako polje, a, kjer je a[0],...,a[$k-1$] urejeno zaporedje ključev in so a[k],...,a[a.length -1] vsi postavljeni na `null`. Če je `x` v polju na mestu `i`, potem metoda na `jdIt(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;
}
```

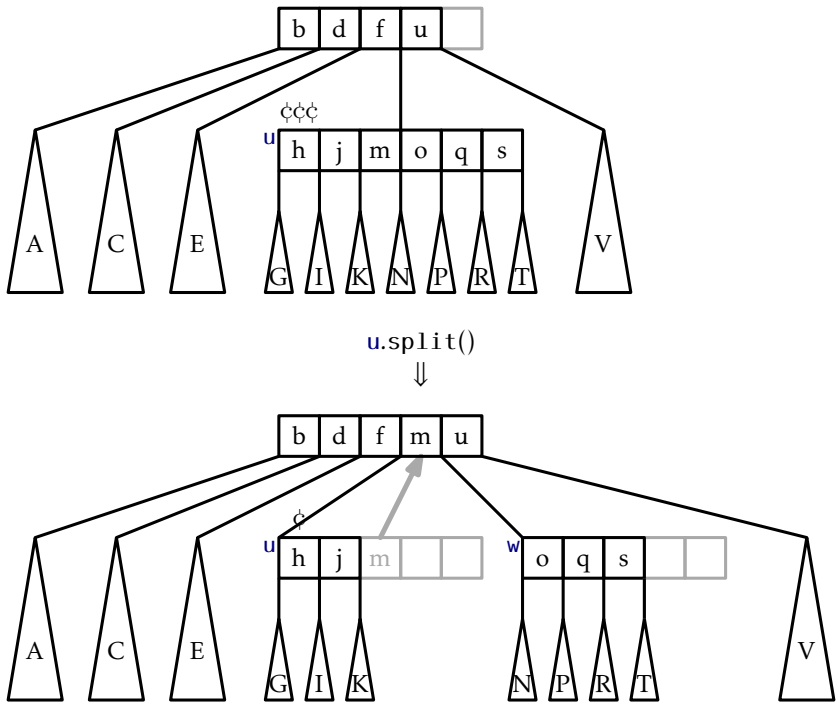
Metoda na `jdiIt(a,x)` uporabi binarno iskanje ki razpolovi iskanje pri vsakem koraku. Za delovanje porabi $O(\log($a.length$))$ časa. V našem primeru, a.length $= 2B$, zato na `jdiIt(a,x)` porabi $O(\log B)$ časa.
Čas delovanja obeh operacij B-drevesa `find(x)` lahko analiziramo v običajni besedi-RAM modelu (kjer štejemo vsako navodilo) in v zunanjem spominskem 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 spominskem modelu, čas, ki ga porabi operacija na `jdi(x)` $O(\log_B n)$. Da določimo čas delovanja v RAM modelu, moramo računati čas klicanja operacije na `jdiIt(a,x)` za vsako vozlišče, ki ga obiščemo. Čas delovanja operacije na `jdi(x)` v RAM modelu je

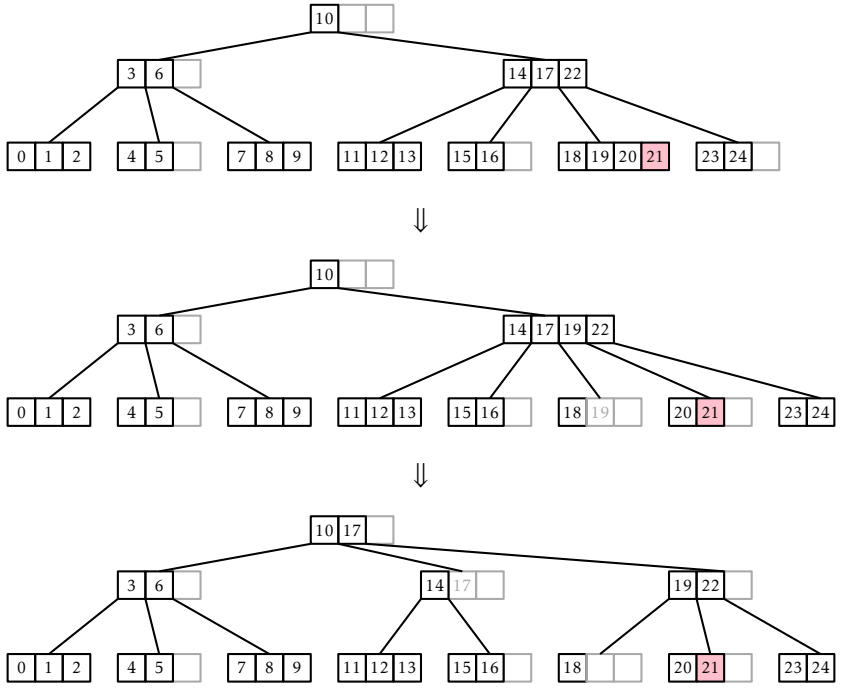
$$O(\log_B n) \times O(\log B) = O(\log n) \ .$$

11.2.2 Addition

One important difference between B-trees and the `BinarySearchTree` data structure from Section 5.2 is that the nodes of a B-tree do not store pointers to their parents. The reason for this will be explained shortly. The lack of parent pointers means that the `add(x)` and `remove(x)` operations on B-trees are most easily implemented using recursion.
Like all balanced search trees, some form of rebalancing is required during an `add(x)` operation. In a B-tree, this is done by *splitting* nodes. Refer to Figure 12.5 for what follows. Although splitting takes place across two levels of recursion, it is best understood as an operation that takes a node `u` containing $2B$ keys and having $2B+1$ children. It creates a new node, `w`, that adopts `u.children`$[B],\ldots,$`u.children`$[2B]$. The new node `w` also takes `u`'s B largest keys, `u.keys`$[B],\ldots,$`u.keys`$[2B-1]$. At this point, `u` has B children and B keys. The extra key, `u.keys`$[B-1]$, is passed up to the parent of `u`, which also adopts `w`.
Notice that the splitting operation modifies three nodes: `u`, `u`'s parent, and the new node, `w`. This is why it is important that the nodes of a B-tree do not maintain parent pointers. If they did, then the $B+1$ children adopted by `w` would all need to have their parent pointers modified. This would increase the number of external memory accesses from 3 to $B+4$ and would make B-trees much less efficient for large values of B.
The `add(x)` method in a B-tree is illustrated in Figure 12.6. At a high level, this method finds a leaf, `u`, at which to add the value `x`. If this causes `u` to become overfull (because it already contained $B-1$ keys), then `u` is split. If this causes `u`'s parent to become overfull, then `u`'s parent is also split, which may cause `u`'s grandparent to become overfull,



Slika 11.5: Splitting the node u in a B-tree ($B = 3$). Notice that the key u.keys[2] = m passes from u to its parent.



Slika 11.6: The `add(x)` operation in a `BTree`. Adding the value 21 results in two nodes being split.

and so on. This process continues, moving up the tree one level at a time until reaching a node that is not overfull or until the root is split. In the former case, the process stops. In the latter case, a new root is created whose two children become the nodes obtained when the original root was split.

The executive summary of the `add(x)` method is that it walks from the root to a leaf searching for `x`, adds `x` to this leaf, and then walks back up to the root, splitting any overfull nodes it encounters along the way. With this high level view in mind, we can now delve into the details of how this method can be implemented recursively.

The real work of `add(x)` is done by the `addRecursive(x,ui)` method, which adds the value `x` to the subtree whose root, `u`, has the identifier `ui`. If `u` is a leaf, then `x` is simply inserted into `u.keys`. Otherwise, `x` is added

recursively into the appropriate child, u', of u. The result of this recursive call is normally null but may also be a reference to a newly-created node, w, that was created because u' was split. In this case, u adopts w and takes its first key, completing the splitting operation on u'.

After the value x has been added (either to u or to a descendant of u), the addRecursive(x,ui) method checks to see if u is storing too many (more than $2B-1$) keys. If so, then u needs to be *split* with a call to the u.split() method. The result of calling u.split() is a new node that is used as the return value for 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;
}
```

The addRecursive(x,ui) method is a helper for the add(x) method, which calls addRecursive(x,ri) to insert x into the root of the *B*-tree. If addRecursive(x,ri) causes the root to split, then a new root is created that takes as its children both the old root and the new node created by the splitting of the old root.

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;
}
```

The add(x) method and its helper, addRecursive(x,ui), can be analyzed in two phases:

Downward phase: During the downward phase of the recursion, before x has been added, they access a sequence of BTree nodes and call findIt(a,x) on each node. As with the find(x) method, this takes $O(\log_B n)$ time in the external memory model and $O(\log n)$ time in the word-RAM model.

Upward phase: During the upward phase of the recursion, after x has been added, these methods perform a sequence of at most $O(\log_B n)$ splits. Each split involves only three nodes, so this phase takes $O(\log_B n)$ time in the external memory model. However, each split involves moving B keys and children from one node to another, so in the word-RAM model, this takes $O(B \log n)$ time.

Recall that the value of B can be quite large, much larger than even $\log n$. Therefore, in the word-RAM model, adding a value to a B-tree can be much slower than adding into a balanced binary search tree. Later, in Section 12.2.4, we will show that the situation is not quite so bad; the amortized number of split operations done during an add(x) operation is

constant. This shows that the (amortized) running time of the add(x) operation in the word-RAM model is $O(B + \log n)$.

11.2.3 Removal

The remove(x) operation in a BTree is, again, most easily implemented as a recursive method. Although the recursive implementation of remove(x) spreads the complexity across several methods, the overall process, which is illustrated in Figure 12.7, is fairly straightforward. By shuffling keys around, removal is reduced to the problem of removing a value, x′, from some leaf, u. Removing x′ may leave u with less than $B-1$ keys; this situation is called an *underflow*.

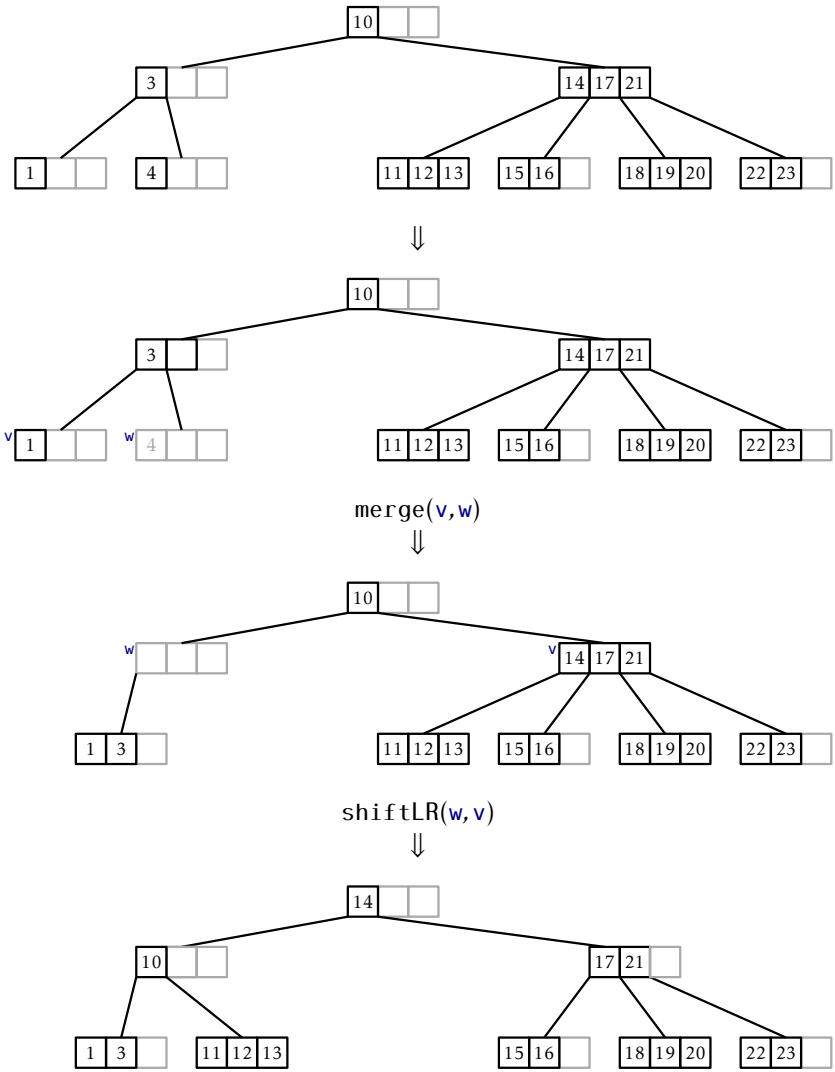
When an underflow occurs, u either borrows keys from, or is merged with, one of its siblings. If u is merged with a sibling, then u's parent will now have one less child and one less key, which can cause u's parent to underflow; this is again corrected by borrowing or merging, but merging may cause u's grandparent to underflow. This process works its way back up to the root until there is no more underflow or until the root has its last two children merged into a single child. When the latter case occurs, the root is removed and its lone child becomes the new root.

Next we delve into the details of how each of these steps is implemented. The first job of the remove(x) method is to find the element x that should be removed. If x is found in a leaf, then x is removed from this leaf. Otherwise, if x is found at u.keys[i] for some internal node, u, then the algorithm removes the smallest value, x′, in the subtree rooted at u.children[i + 1]. The value x′ is the smallest value stored in the BTree that is greater than x. The value of x′ is then used to replace x in u.keys[i]. This process is illustrated in Figure 12.8.

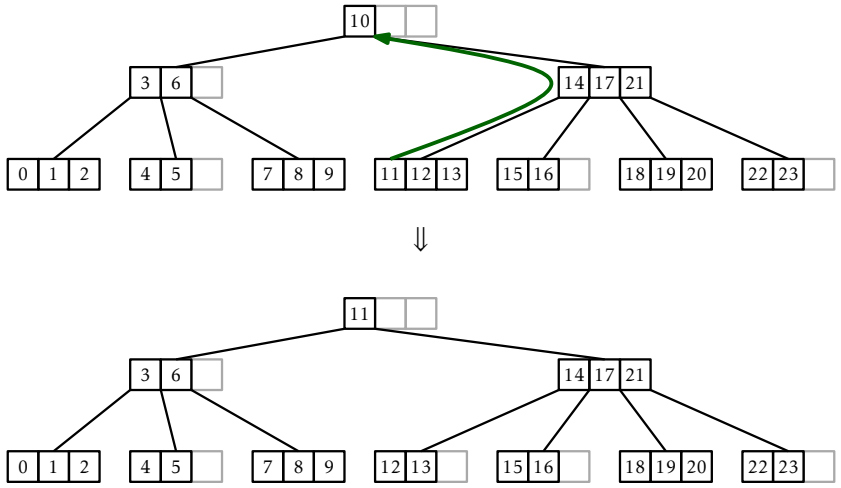
The removeRecursive(x,ui) method is a recursive implementation of the preceding algorithm:

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);
```



Slika 11.7: Removing the value 4 from a *B*-tree results in one merge and one borrowing operation.



Slika 11.8: The `remove(x)` operation in a `BTree`. To remove the value `x` = 10 we replace it with the the value $x' = 11$ and remove 11 from the leaf that contains it.

```
  return y;
}
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;
}
```

Note that, after recursively removing the value x from the ith child of u, removeRecursive(x,ui) needs to ensure that this child still has at least $B-1$ keys. In the preceding code, this is done using a method called checkUnderflow(x,i), which checks for and corrects an underflow in the ith child of u. Let w be the ith child of u. If w has only $B-2$ keys, then this needs to be fixed. The fix requires using a sibling of w. This can be either child $i+1$ of u or child $i-1$ of u. We will usually use child $i-1$ of u, which is the sibling, v, of w directly to its left. The only time this doesn't work is when $i=0$, in which case we use the sibling directly to w's right.

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);
}
```

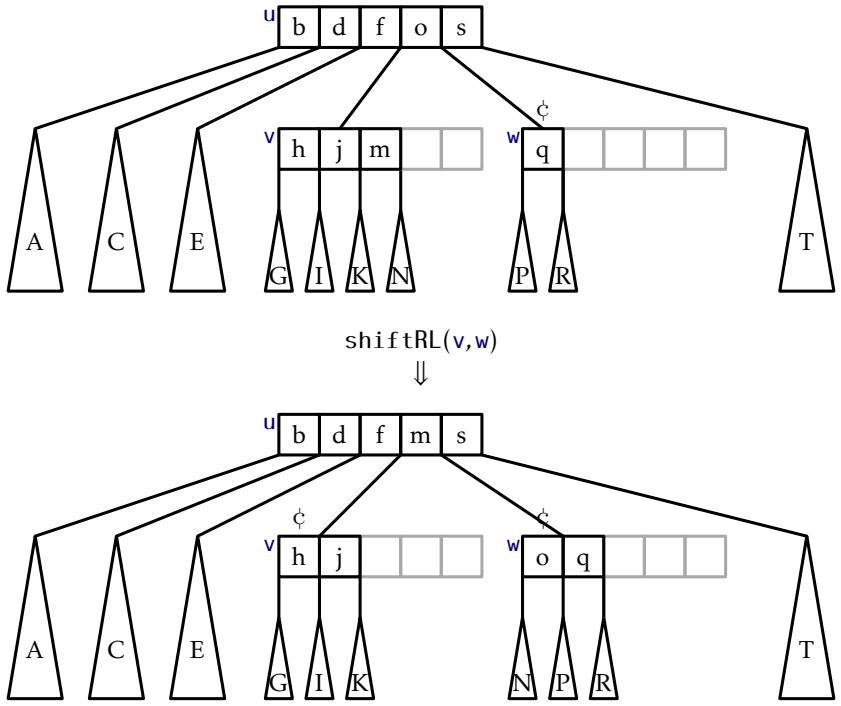
In the following, we focus on the case when $i \neq 0$ so that any underflow at the ith child of u will be corrected with the help of the $(i-1)$st child of u. The case $i=0$ is similar and the details can be found in the accompanying source code.
To fix an underflow at node w, we need to find more keys (and possibly also children), for w. There are two ways to do this:

Borrowing: If w has a sibling, v, with more than $B-1$ keys, then w can borrow some keys (and possibly also children) from v. More specifically, if v stores size(v) keys, then between them, v and w have a total of

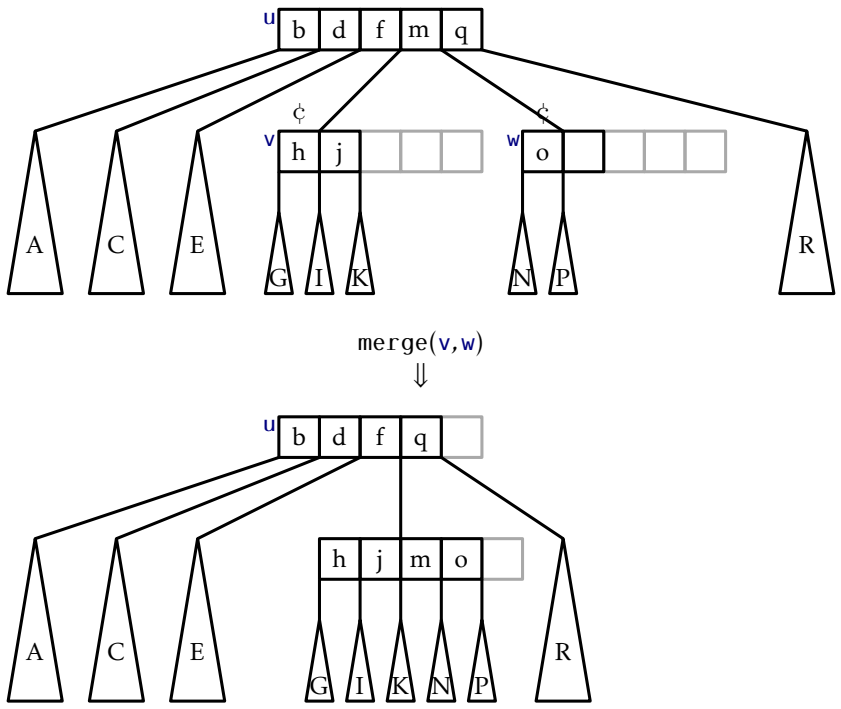
$$B-2+\texttt{size}(\texttt{w}) \geq 2B-2$$

keys. We can therefore shift keys from v to w so that each of v and w has at least $B-1$ keys. This process is illustrated in Figure 12.9.

Merging: If v has only $B-1$ keys, we must do something more drastic, since v cannot afford to give any keys to w. Therefore, we *merge* v and w as shown in Figure 12.10. The merge operation is the



Slika 11.9: If v has more than $B-1$ keys, then w can borrow keys from v.



Slika 11.10: Merging two siblings v and w in a B-tree ($B = 3$).

opposite of the split operation. It takes two nodes that contain a total of $2B-3$ keys and merges them into a single node that contains $2B-2$ keys. (The additional key comes from the fact that, when we merge v and w, their common parent, u, now has one less child and therefore needs to give up one of its keys.)

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);
    }
  }
}
```

To summarize, the `remove(x)` method in a B-tree follows a root to leaf path, removes a key `x`′ from a leaf, `u`, and then performs zero or more merge operations involving `u` and its ancestors, and performs at most one borrowing operation. Since each merge and borrow operation involves modifying only three nodes, and only $O(\log_B n)$ of these operations occur, the entire process takes $O(\log_B n)$ time in the external memory model. Again, however, each merge and borrow operation takes $O(B)$ time in the word-RAM model, so (for now) the most we can say about the running time required by `remove(x)` in the word-RAM model is that it is $O(B \log_B n)$.

11.2.4 Amortized Analysis of B-Trees

Thus far, we have shown that

1. In the external memory model, the running time of `find(x)`, `add(x)`, and `remove(x)` in a B-tree is $O(\log_B n)$.

2. In the word-RAM model, the running time of `find(x)` is $O(\log n)$ and the running time of `add(x)` and `remove(x)` is $O(B \log n)$.

The following lemma shows that, so far, we have overestimated the number of merge and split operations performed by B-trees.

Lemma 11.1. *Starting with an empty B-tree and performing any sequence of* m `add(x)` *and* `remove(x)` *operations results in at most* $3m/2$ *splits, merges, and borrows being performed.*

Dokaz. The proof of this has already been sketched in Section 7.3 for the special case in which $B = 2$. The lemma can be proven using a credit scheme, in which

1. each split, merge, or borrow operation is paid for with two credits, i.e., a credit is removed each time one of these operations occurs; and

2. at most three credits are created during any `add(x)` or `remove(x)` operation.

Since at most $3m$ credits are ever created and each split, merge, and borrow is paid for with with two credits, it follows that at most $3m/2$ splits, merges, and borrows are performed. These credits are illustrated using the ¢ symbol in Figures 12.5, 12.9, and 12.10.
To keep track of these credits the proof maintains the following *credit invariant*: Any non-root node with $B-1$ keys stores one credit and any node with $2B-1$ keys stores three credits. A node that stores at least B keys and most $2B-2$ keys need not store any credits. What remains is to show that we can maintain the credit invariant and satisfy properties 1 and 2, above, during each `add(x)` and `remove(x)` operation.

Adding: The `add(x)` method does not perform any merges or borrows, so we need only consider split operations that occur as a result of calls to `add(x)`.
Each split operation occurs because a key is added to a node, `u`, that already contains $2B-1$ keys. When this happens, `u` is split into two nodes, `u'` and `u''` having $B-1$ and B keys, respectively. Prior to this operation, `u` was storing $2B-1$ keys, and hence three credits. Two of these credits can be used to pay for the split and the other credit can be given to `u'` (which has $B-1$ keys) to maintain the credit invariant. Therefore, we can pay for the split and maintain the credit invariant during any split.

The only other modification to nodes that occur during an `add(x)` operation happens after all splits, if any, are complete. This modification involves adding a new key to some node `u`′. If, prior to this, `u`′ had $2B-2$ children, then it now has $2B-1$ children and must therefore receive three credits. These are the only credits given out by the `add(x)` method.

Removing: During a call to `remove(x)`, zero or more merges occur and are possibly followed by a single borrow. Each merge occurs because two nodes, `v` and `w`, each of which had exactly $B-1$ keys prior to calling `remove(x)` were merged into a single node with exactly $2B-2$ keys. Each such merge therefore frees up two credits that can be used to pay for the merge.

After any merges are performed, at most one borrow operation occurs, after which no further merges or borrows occur. This borrow operation only occurs if we remove a key from a leaf, `v`, that has $B-1$ keys. The node `v` therefore has one credit, and this credit goes towards the cost of the borrow. This single credit is not enough to pay for the borrow, so we create one credit to complete the payment.

At this point, we have created one credit and we still need to show that the credit invariant can be maintained. In the worst case, `v`'s sibling, `w`, has exactly B keys before the borrow so that, afterwards, both `v` and `w` have $B-1$ keys. This means that `v` and `w` each should be storing a credit when the operation is complete. Therefore, in this case, we create an additional two credits to give to `v` and `w`. Since a borrow happens at most once during a `remove(x)` operation, this means that we create at most three credits, as required.

If the `remove(x)` operation does not include a borrow operation, this is because it finishes by removing a key from some node that, prior to the operation, had B or more keys. In the worst case, this node had exactly B keys, so that it now has $B-1$ keys and must be given one credit, which we create.

In either case—whether the removal finishes with a borrow operation or not—at most three credits need to be created during a call to `remove(x)` to maintain the credit invariant and pay for all borrows and merges that occur. This completes the proof of the lemma. □

The purpose of Lemma 12.1 is to show that, in the word-RAM model the cost of splits, merges and joins during a sequence of m `add(x)` and `remove(x)` operations is only $O(Bm)$. That is, the amortized cost per operation is only $O(B)$, so the amortized cost of `add(x)` and `remove(x)` in the word-RAM model is $O(B + \log n)$. This is summarized by the following pair of theorems:

Theorem 11.1 (External Memory B-Trees)**.** *A `BTree` implements the `SSet` interface. In the external memory model, a `BTree` supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log_B n)$ *time per operation.*

Theorem 11.2 (Word-RAM B-Trees)**.** *A `BTree` implements the `SSet` interface. In the word-RAM model, and ignoring the cost of splits, merges, and borrows, a `BTree` supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log n)$ *time per operation. Furthermore, beginning with an empty `BTree`, any sequence of m* `add(x)` *and* `remove(x)` *operations results in a total of* $O(Bm)$ *time spent performing splits, merges, and borrows.*

11.3 Discussion and Exercises

The external memory model of computation was introduced by Aggarwal and Vitter [**?**]. It is sometimes also called the *I/O model* or the *disk access model*.

B-Trees are to external memory searching what binary search trees are to internal memory searching. B-trees were introduced by Bayer and McCreight [**?**] in 1970 and, less than ten years later, the title of Comer's ACM Computing Surveys article referred to them as ubiquitous [**?**]. Like binary search trees, there are many variants of B-Trees, including B^+-trees, B^*-trees, and counted B-trees. B-trees are indeed ubiquitous and are the primary data structure in many file systems, including Apple's HFS+, Microsoft's NTFS, and Linux's Ext4; every major database system; and key-value stores used in cloud computing. Graefe's recent survey [**?**] provides a 200+ page overview of the many modern applications, variants, and optimizations of B-trees.

B-trees implement the `SSet` interface. If only the `USet` interface is needed, then external memory hashing could be used as an alternative to

B-trees. External memory hashing schemes do exist; see, for example, Jensen and Pagh [**?**]. These schemes implement the USet operations in $O(1)$ expected time in the external memory model. However, for a variety of reasons, many applications still use B-trees even though they only require USet operations.

One reason B-trees are such a popular choice is that they often perform better than their $O(\log_B n)$ running time bounds suggest. The reason for this is that, in external memory settings, the value of B is typically quite large—in the hundreds or even thousands. This means that 99% or even 99.9% of the data in a B-tree is stored in the leaves. In a database system with a large memory, it may be possible to cache all the internal nodes of a B-tree in RAM, since they only represent 1% or 0.1% of the total data set. When this happens, this means that a search in a B-tree involves a very fast search in RAM, through the internal nodes, followed by a single external memory access to retrieve a leaf.

Exercise 11.1. Show what happens when the keys 1.5 and then 7.5 are added to the B-tree in Figure **??**.

Exercise 11.2. Show what happens when the keys 3 and then 4 are removed from the B-tree in Figure **??**.

Exercise 11.3. What is the maximum number of internal nodes in a B-tree that stores n keys (as a function of n and B)?

Exercise 11.4. The introduction to this chapter claims that B-trees only need an internal memory of size $O(B + \log_B n)$. However, the implementation given here actually requires more memory.

1. Show that the implementation of the add(x) and remove(x) methods given in this chapter use an internal memory proportional to $B \log_B n$.
2. Describe how these methods could be modified in order to reduce their memory consumption to $O(B + \log_B n)$.

Exercise 11.5. Draw the credits used in the proof of Lemma 12.1 on the trees in Figures 12.6 and 12.7. Verify that (with three additional credits) it is possible to pay for the splits, merges, and borrows and maintain the credit invariant.

Exercise 11.6. Design a modified version of a B-tree in which nodes can have anywhere from B up to $3B$ children (and hence $B-1$ up to $3B-1$ keys). Show that this new version of B-trees performs only $O(m/B)$ splits, merges, and borrows during a sequence of m operations. (Hint: For this to work, you will have to be more agressive with merging, sometimes merging two nodes before it is strictly necessary.)

Exercise 11.7. In this exercise, you will design a modified method of splitting and merging in B-trees that asymptotically reduces the number of splits, borrows and merges by considering up to three nodes at a time.

1. Let u be an overfull node and let v be a sibling immediately to the right of u. There are two ways to fix the overflow at u:

   (a) u can give some of its keys to v; or

   (b) u can be split and the keys of u and v can be evenly distributed among u, v, and the newly created node, w.

   Show that this can always be done in such a way that, after the operation, each of the (at most 3) affected nodes has at least $B+\alpha B$ keys and at most $2B-\alpha B$ keys, for some constant $\alpha > 0$.

2. Let u be an underfull node and let v and w be siblings of u There are two ways to fix the underflow at u:

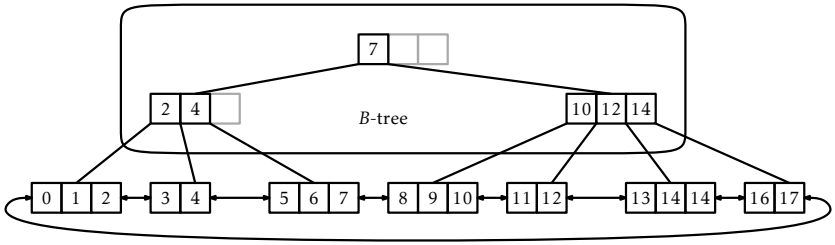
   (a) keys can be redistributed among u, v, and w; or

   (b) u, v, and w can be merged into two nodes and the keys of u, v, and w can be redistributed amongst these nodes.

   Show that this can always be done in such a way that, after the operation, each of the (at most 3) affected nodes has at least $B+\alpha B$ keys and at most $2B-\alpha B$ keys, for some constant $\alpha > 0$.

3. Show that, with these modifications, the number of merges, borrows, and splits that occur during m operations is $O(m/B)$.

Exercise 11.8. A B^+-tree, illustrated in Figure 12.11 stores every key in a leaf and keeps its leaves stored as a doubly-linked list. As usual, each leaf stores between $B-1$ and $2B-1$ keys. Above this list is a standard B-tree that stores the largest value from each leaf but the last.



Slika 11.11: A B^+-tree is a B-tree on top of a doubly-linked list of blocks.

1. Describe fast implementations of `add(x)`, `remove(x)`, and `find(x)` in a B^+-tree.

2. Explain how to efficiently implement the `findRange(x,y)` method, that reports all values greater than `x` and less than or equal to `y`, in a B^+-tree.

3. Implement a class, `BPlusTree`, that implements `find(x)`, `add(x)`, `remove(x)`, and `findRange(x,y)`.

4. B^+-trees duplicate some of the keys because they are stored both in the B-tree and in the list. Explain why this duplication does not add up to much for large values of B.

=======

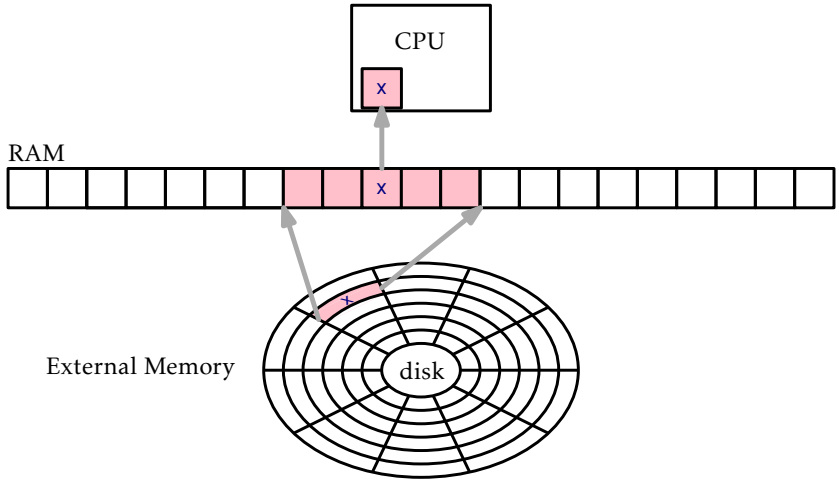
Poglavje 12

External Memory Searching

Throughout this book, we have been using the w-bit word-RAM model of computation defined in Section 1.4. An implicit assumption of this model is that our computer has a large enough random access memory to store all of the data in the data structure. In some situations, this assumption is not valid. There exist collections of data so large that no computer has enough memory to store them. In such cases, the application must resort to storing the data on some external storage medium such as a hard disk, a solid state disk, or even a network file server (which has its own external storage).

Accessing an item from external storage is extremely slow. The hard disk attached to the computer on which this book was written has an average access time of 19ms and the solid state drive attached to the computer has an average access time of 0.3ms. In contrast, the random access memory in the computer has an average access time of less than 0.000113ms. Accessing RAM is more than 2 500 times faster than accessing the solid state drive and more than 160 000 times faster than accessing the hard drive.

These speeds are fairly typical; accessing a random byte from RAM is thousands of times faster than accessing a random byte from a hard disk or solid-state drive. Access time, however, does not tell the whole story. When we access a byte from a hard disk or solid state disk, an entire *block* of the disk is read. Each of the drives attached to the computer has a block size of 4 096; each time we read one byte, the drive gives us a block containing 4 096 bytes. If we organize our data structure carefully,



Slika 12.1: In the external memory model, accessing an individual item, x, in the external memory requires reading the entire block containing x into RAM.

this means that each disk access could yield 4 096 bytes that are helpful in completing whatever operation we are doing.

This is the idea behind the *external memory model* of computation, illustrated schematically in Figure 12.1. In this model, the computer has access to a large external memory in which all of the data resides. This memory is divided into memory *blocks* each containing B words. The computer also has limited internal memory on which it can perform computations. Transferring a block between internal memory and external memory takes constant time. Computations performed within the internal memory are *free*; they take no time at all. The fact that internal memory computations are free may seem a bit strange, but it simply emphasizes the fact that external memory is so much slower than RAM.

In the full-blown external memory model, the size of the internal memory is also a parameter. However, for the data structures described in this chapter, it is sufficient to have an internal memory of size $O(B + \log_B n)$. That is, the memory needs to be capable of storing a constant number of blocks and a recursion stack of height $O(\log_B n)$. In most cases, the $O(B)$ term dominates the memory requirement. For

example, even with the relatively small value $B = 32$, $B \geq \log_B n$ for all $n \leq 2^{160}$. In decimal, $B \geq \log_B n$ for any

$$n \leq 1\,461\,501\,637\,330\,902\,918\,203\,684\,832\,716\,283\,019\,655\,932\,542\,976 \ .$$

12.1 The Block Store

The notion of external memory includes a large number of possible different devices, each of which has its own block size and is accessed with its own collection of system calls. To simplify the exposition of this chapter so that we can focus on the common ideas, we encapsulate external memory devices with an object called a `BlockStore`. A `BlockStore` stores a collection of memory blocks, each of size B. Each block is uniquely identified by its integer index. A `BlockStore` supports these operations:

1. `readBlock(i)`: Return the contents of the block whose index is `i`.
2. `writeBlock(i,b)`: Write contents of `b` to the block whose index is `i`.
3. `placeBlock(b)`: Return a new index and store the contents of `b` at this index.
4. `freeBlock(i)`: Free the block whose index is `i`. This indicates that the contents of this block are no longer used so the external memory allocated by this block may be reused.

The easiest way to imagine a `BlockStore` is to imagine it as storing a file on disk that is partitioned into blocks, each containing B bytes. In this way, `readBlock(i)` and `writeBlock(i,b)` simply read and write bytes $iB, \ldots, (i+1)B-1$ of this file. In addition, a simple `BlockStore` could keep a *free list* of blocks that are available for use. Blocks freed with `freeBlock(i)` are added to the free list. In this way, `placeBlock(b)` can use a block from the free list or, if none is available, append a new block to the end of the file.

12.2 B-drevesa

V tem poglavju bomo razpravljali o posplošitvah dvojiških dreves imenovanih B-drevesa, ki so učinkovita predvsem v zunanjem spominu. Alternativno se na B-drevesa lahko gleda, kot na posplošitev 2-4 dreves, ki so opisana v Section 7.1. (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 od vozlišča u so shranjeni v polju, u.otroci. 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 ℓ, listov v B-drevesu izpolnjuje

$$2B^{h-1} \leq \ell \leq 2(2B)^{h-1} \ .$$

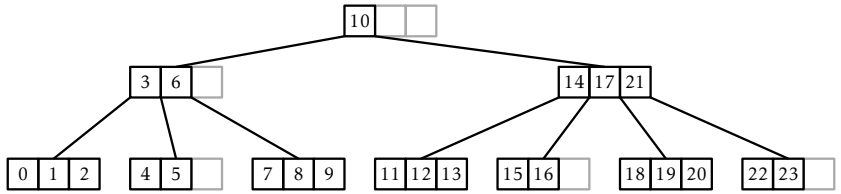
Vzamemo logaritem iz prve neenakosti in preuredimo. Dobimo:

$$\begin{aligned} h &\leq \frac{\log \ell - 1}{\log B} + 1 \\ &\leq \frac{\log \ell}{\log B} + 1 \\ &= \log_B \ell + 1 \ . \end{aligned}$$

Višina B-drevesa je sorazmerna z stopnjo-B logaritma od števila listov.
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,

$$\text{u.keys}[0] < \text{u.keys}[1] < \cdots < \text{u.keys}[k-2] \ .$$

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



Slika 12.2: B-drevo z $B = 2$.

zakoreninjenega na `u.otroci[i]` vendar manjši od vseh ključev shranjenih v poddrevedsu, ki je zakoreninjen na `u.children[i + 1]`.

$$\texttt{u.children[i]} \prec \texttt{u.keys[i]} \prec \texttt{u.children[i + 1]} \ .$$

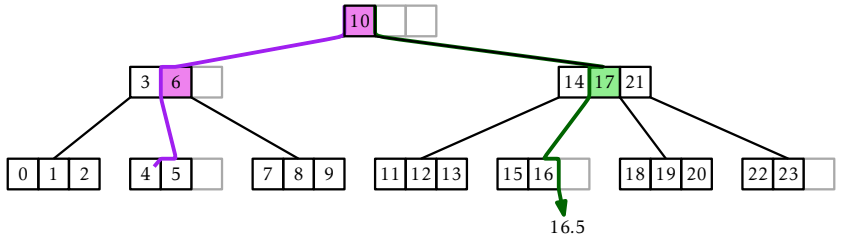
Primer B-drevesa z $B = 2$ je prikazan na sliki Figure **??**.
Upoštevajte, da so podatki shranjeni v vozliščih B-drevesa velikosti $O(B)$. Zato, je v nastavitvah zunanjega spomina, vrednost B v B-drevesu določena tako, da celotno vozlišče lahko ustreza enemu zunanje spominskemu bloku. V tem primeru je čas izvajanja operacij na B-drevesu v zunanjem spominskem modelu sorazmerno številu vozlišč, ki jih obiščemo (branje ali pisanje) z operacijo.
Na 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, kateri 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 struktruri:

BTree

```
int n; // number of elements stored in the tree
int ri; // index of the root
BlockStore<Node*> bs;
```



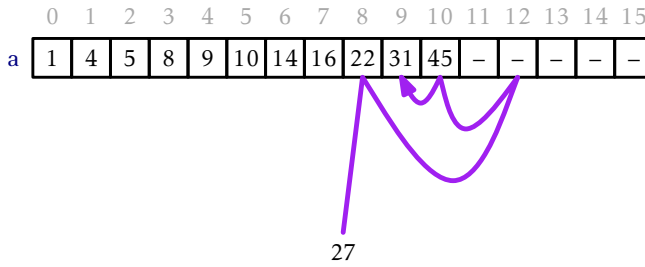
Slika 12.3: Uspešno iskanje (vrednosti 4) in neuspešno iskanje (za vrednost 16.5) v *B*-drevesu. Obarvana vozlišča predstavljajo, kje se je vrednost med iskanjem `z` spremenila.

12.2.1 Searching

Implementacija operacije na `jdi(x)`, ki je ilustrirana v Figure 12.3, je posplošitev operacije na `jdi(x)` v dvojiškem iskalnem drevesu. Iskanje `x`-a se začne v korenu. 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.otrici[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;
```



Slika 12.4: The execution of `findIt(a,27)`.

```
}
```

Osrednjega pomena za metodo na `jdi(x)` je metoda na `jdiIt(a,x)`, ki išče v `null`-napolnjeno urejeno polje, `a`, vrednost `x`. Ta metoda, predstavljena v Figure 12.4, 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 na `jdIt(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 na `jdiIt(a,x)` uporabi binarno iskanje ki razpolovi iskanje pri vsakem koraku. Za delovanje porabi $O(\log(a.length))$ časa. V našem primeru, $a.length = 2B$, zato na `jdiIt(a,x)` porabi $O(\log B)$ časa.

Čas delovanja obeh operacij B-drevesa `find(x)` lahko analiziramo v običajni besedi-RAM modelu (kjer vsako navodilo šteje) in v zunanjem spominskem modelu (kjer štejemo samo število obiskanih vozlišč). Ker vsak list v B-drevesu shranjuje vsaj en ključ in je višina od B-drevesa z ℓ listi $O(\log_B \ell)$, je višina od B-drevesa, ki shranjuje `n` ključev $O(\log_B \mathtt{n})$. Zato je v zunanjem spominskem modelu, čas, ki ga porabi operacija na `jdi(x)` $O(\log_B \mathtt{n})$. Da določimo čas delovanja v RAM modelu, moramo računati čas klicanja operacije na `jdiIt(a,x)` za vsako vozlišče, ki ga obiščemo. Čas delovanja operacije na `jdi(x)` v RAM modelu je

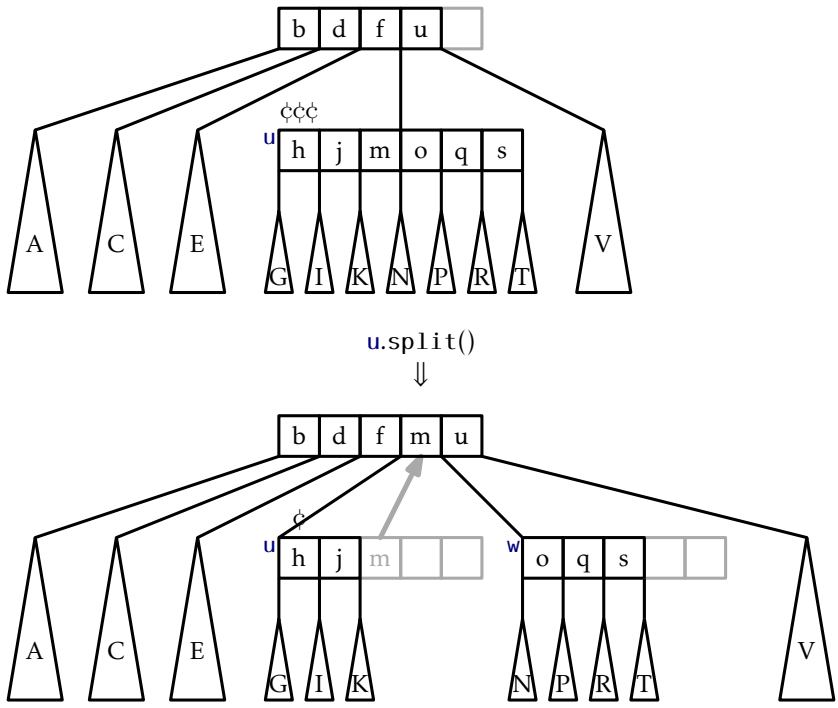
$$O(\log_B \mathtt{n}) \times O(\log B) = O(\log \mathtt{n}) \ .$$

12.2.2 Addition

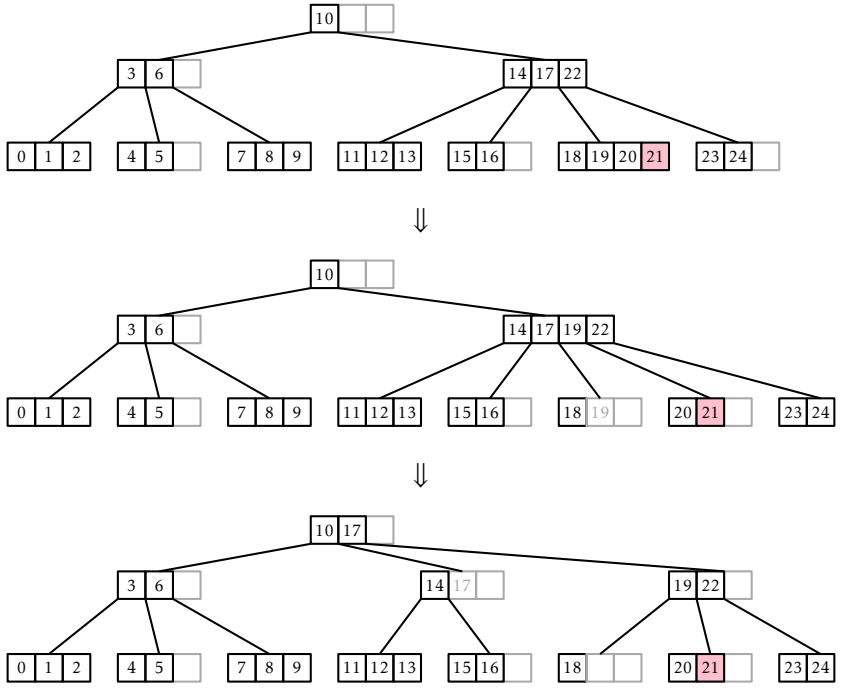
One important difference between B-trees and the `BinarySearchTree` data structure from Section 5.2 is that the nodes of a B-tree do not store pointers to their parents. The reason for this will be explained shortly. The lack of parent pointers means that the `add(x)` and `remove(x)` operations on B-trees are most easily implemented using recursion. Like all balanced search trees, some form of rebalancing is required during an `add(x)` operation. In a B-tree, this is done by *splitting* nodes. Refer to Figure 12.5 for what follows. Although splitting takes place across two levels of recursion, it is best understood as an operation that takes a node `u` containing $2B$ keys and having $2B+1$ children. It creates a new node, `w`, that adopts `u.children`$[B],\ldots,$`u.children`$[2B]$. The new node `w` also takes `u`'s B largest keys, `u.keys`$[B],\ldots,$`u.keys`$[2B-1]$. At this point, `u` has B children and B keys. The extra key, `u.keys`$[B-1]$, is passed up to the parent of `u`, which also adopts `w`.

Notice that the splitting operation modifies three nodes: `u`, `u`'s parent, and the new node, `w`. This is why it is important that the nodes of a B-tree do not maintain parent pointers. If they did, then the $B+1$ children adopted by `w` would all need to have their parent pointers modified. This would increase the number of external memory accesses from 3 to $B+4$ and would make B-trees much less efficient for large values of B.

The `add(x)` method in a B-tree is illustrated in Figure 12.6. At a high



Slika 12.5: Splitting the node u in a B-tree ($B = 3$). Notice that the key u.keys[2] = m passes from u to its parent.



Slika 12.6: The add(x) operation in a BTree. Adding the value 21 results in two nodes being split.

level, this method finds a leaf, u, at which to add the value x. If this causes u to become overfull (because it already contained $B-1$ keys), then u is split. If this causes u's parent to become overfull, then u's parent is also split, which may cause u's grandparent to become overfull, and so on. This process continues, moving up the tree one level at a time until reaching a node that is not overfull or until the root is split. In the former case, the process stops. In the latter case, a new root is created whose two children become the nodes obtained when the original root was split.

The executive summary of the add(x) method is that it walks from the root to a leaf searching for x, adds x to this leaf, and then walks back up to the root, splitting any overfull nodes it encounters along the way. With this high level view in mind, we can now delve into the details of

how this method can be implemented recursively.
The real work of `add(x)` is done by the `addRecursive(x,ui)` method, which adds the value `x` to the subtree whose root, `u`, has the identifier `ui`. If `u` is a leaf, then `x` is simply inserted into `u.keys`. Otherwise, `x` is added recursively into the appropriate child, `u'`, of `u`. The result of this recursive call is normally `null` but may also be a reference to a newly-created node, `w`, that was created because `u'` was split. In this case, `u` adopts `w` and takes its first key, completing the splitting operation on `u'`.
After the value `x` has been added (either to `u` or to a descendant of `u`), the `addRecursive(x,ui)` method checks to see if `u` is storing too many (more than $2B-1$) keys. If so, then `u` needs to be *split* with a call to the `u.split()` method. The result of calling `u.split()` is a new node that is used as the return value for `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;
}
```

The `addRecursive(x,ui)` method is a helper for the `add(x)` method, which calls `addRecursive(x,ri)` to insert `x` into the root of the B-tree. If `addRecursive(x,ri)` causes the root to split, then a new root is created that takes as its children both the old root and the new node created by the splitting of the old root.

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;
}
```

The `add(x)` method and its helper, `addRecursive(x,ui)`, can be analyzed in two phases:

Downward phase: During the downward phase of the recursion, before `x` has been added, they access a sequence of `BTree` nodes and call `findIt(a,x)` on each node. As with the `find(x)` method, this takes $O(\log_B n)$ time in the external memory model and $O(\log n)$ time in the word-RAM model.

Upward phase: During the upward phase of the recursion, after `x` has been added, these methods perform a sequence of at most $O(\log_B n)$ splits. Each split involves only three nodes, so this phase takes $O(\log_B n)$ time in the external memory model. However, each split involves moving B keys and children from one node to another, so in the word-RAM model, this takes $O(B \log n)$ time.

Recall that the value of B can be quite large, much larger than even $\log n$. Therefore, in the word-RAM model, adding a value to a B-tree can be

much slower than adding into a balanced binary search tree. Later, in Section 12.2.4, we will show that the situation is not quite so bad; the amortized number of split operations done during an add(x) operation is constant. This shows that the (amortized) running time of the add(x) operation in the word-RAM model is $O(B + \log n)$.

12.2.3 Removal

The remove(x) operation in a BTree is, again, most easily implemented as a recursive method. Although the recursive implementation of remove(x) spreads the complexity across several methods, the overall process, which is illustrated in Figure 12.7, is fairly straightforward. By shuffling keys around, removal is reduced to the problem of removing a value, x′, from some leaf, u. Removing x′ may leave u with less than $B-1$ keys; this situation is called an *underflow*.

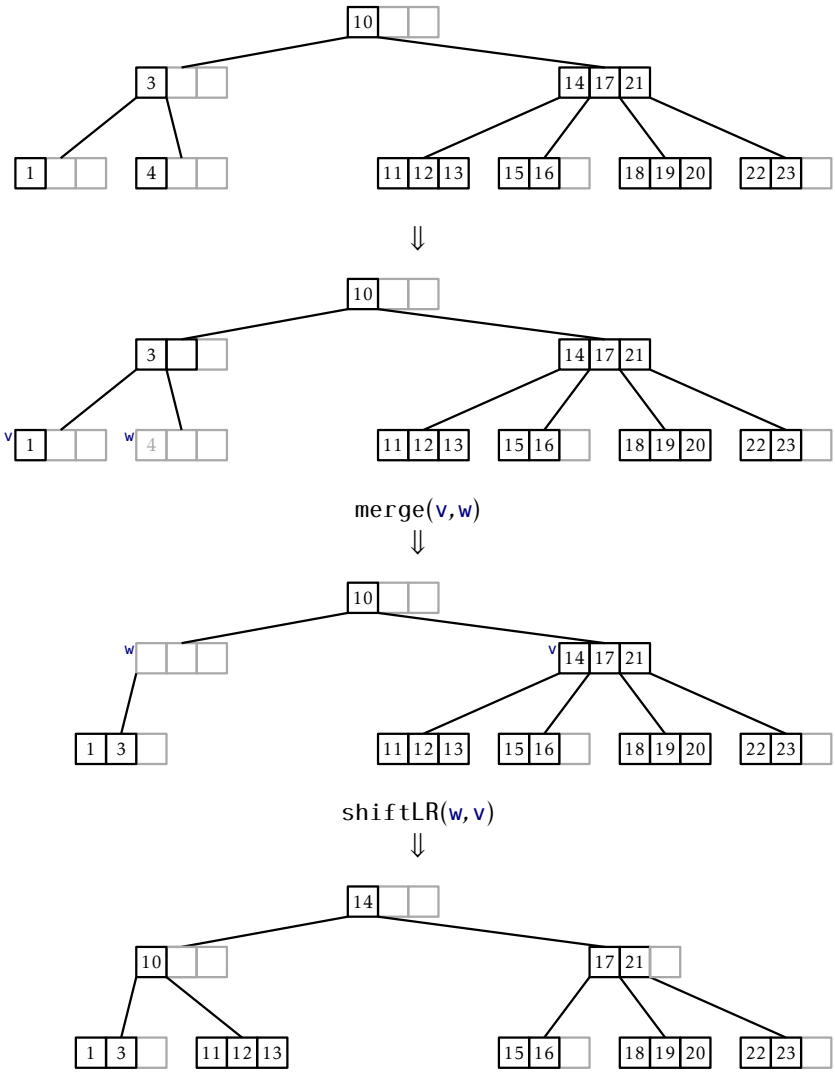
When an underflow occurs, u either borrows keys from, or is merged with, one of its siblings. If u is merged with a sibling, then u's parent will now have one less child and one less key, which can cause u's parent to underflow; this is again corrected by borrowing or merging, but merging may cause u's grandparent to underflow. This process works its way back up to the root until there is no more underflow or until the root has its last two children merged into a single child. When the latter case occurs, the root is removed and its lone child becomes the new root.

Next we delve into the details of how each of these steps is implemented. The first job of the remove(x) method is to find the element x that should be removed. If x is found in a leaf, then x is removed from this leaf. Otherwise, if x is found at u.keys[i] for some internal node, u, then the algorithm removes the smallest value, x′, in the subtree rooted at u.children[i + 1]. The value x′ is the smallest value stored in the BTree that is greater than x. The value of x′ is then used to replace x in u.keys[i]. This process is illustrated in Figure 12.8.

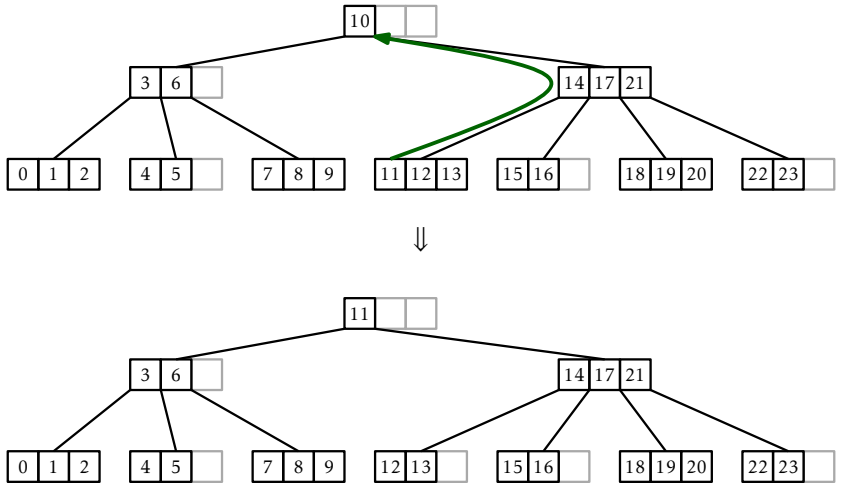
The removeRecursive(x,ui) method is a recursive implementation of the preceding algorithm:

BTree

```
T removeSmallest(int ui) {
  Node* u = bs.readBlock(ui);
```



Slika 12.7: Removing the value 4 from a *B*-tree results in one merge and one borrowing operation.



Slika 12.8: The `remove(x)` operation in a `BTree`. To remove the value `x` = 10 we replace it with the the value `x'` = 11 and remove 11 from the leaf that contains it.

```
  if (u->isLeaf())
    return u->remove(0);
  T y = removeSmallest(u->children[0]);
  checkUnderflow(u, 0);
  return y;
}
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;
}
```

Note that, after recursively removing the value x from the ith child of u, removeRecursive(x,ui) needs to ensure that this child still has at least $B-1$ keys. In the preceding code, this is done using a method called checkUnderflow(x,i), which checks for and corrects an underflow in the ith child of u. Let w be the ith child of u. If w has only $B-2$ keys, then this needs to be fixed. The fix requires using a sibling of w. This can be either child $i+1$ of u or child $i-1$ of u. We will usually use child $i-1$ of u, which is the sibling, v, of w directly to its left. The only time this doesn't work is when $i=0$, in which case we use the sibling directly to w's right.

BTree

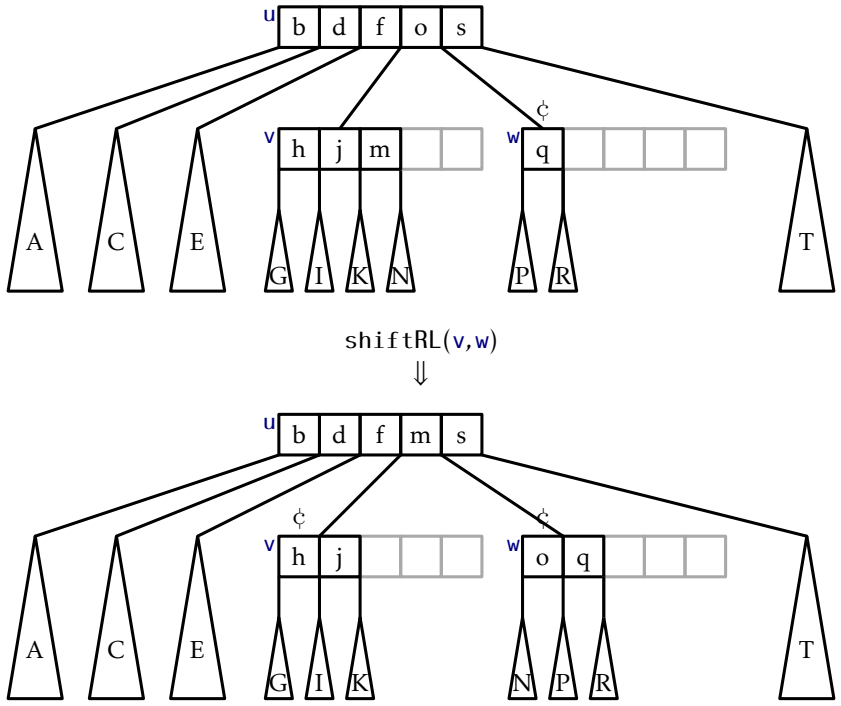
```
void checkUnderflow(Node* u, int i) {
  if (u->children[i] < 0) return;
  if (i == 0)
    checkUnderflowZero(u, i); // use u's right sibling
  else
    checkUnderflowNonZero(u, i);
}
```

In the following, we focus on the case when $i \neq 0$ so that any underflow at the ith child of u will be corrected with the help of the $(i-1)$st child of u. The case $i=0$ is similar and the details can be found in the accompanying source code.
To fix an underflow at node w, we need to find more keys (and possibly also children), for w. There are two ways to do this:

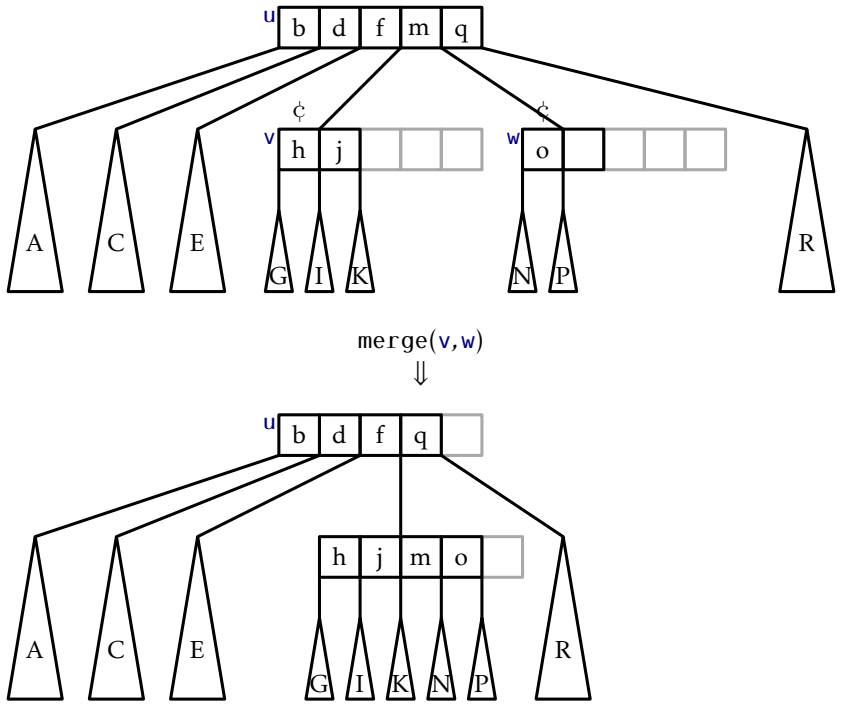
Borrowing: If w has a sibling, v, with more than $B-1$ keys, then w can borrow some keys (and possibly also children) from v. More specifically, if v stores size(v) keys, then between them, v and w have a total of

$$B-2+\texttt{size}(\texttt{w}) \geq 2B-2$$

keys. We can therefore shift keys from v to w so that each of v and w has at least $B-1$ keys. This process is illustrated in Figure 12.9.



Slika 12.9: If v has more than $B-1$ keys, then w can borrow keys from v.



Slika 12.10: Merging two siblings v and w in a B-tree ($B = 3$).

Merging: If v has only $B-1$ keys, we must do something more drastic, since v cannot afford to give any keys to w. Therefore, we *merge* v and w as shown in Figure 12.10. The merge operation is the opposite of the split operation. It takes two nodes that contain a total of $2B-3$ keys and merges them into a single node that contains $2B-2$ keys. (The additional key comes from the fact that, when we merge v and w, their common parent, u, now has one less child and therefore needs to give up one of its keys.)

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);
    }
  }
}
```

To summarize, the `remove(x)` method in a B-tree follows a root to leaf path, removes a key `x` from a leaf, `u`, and then performs zero or more merge operations involving `u` and its ancestors, and performs at most one borrowing operation. Since each merge and borrow operation involves modifying only three nodes, and only $O(\log_B n)$ of these operations occur, the entire process takes $O(\log_B n)$ time in the external memory model. Again, however, each merge and borrow operation takes $O(B)$ time in the word-RAM model, so (for now) the most we can say about the running time required by `remove(x)` in the word-RAM model is that it is $O(B \log_B n)$.

12.2.4 Amortized Analysis of B-Trees

Thus far, we have shown that

1. In the external memory model, the running time of `find(x)`, `add(x)`, and `remove(x)` in a B-tree is $O(\log_B n)$.
2. In the word-RAM model, the running time of `find(x)` is $O(\log n)$ and the running time of `add(x)` and `remove(x)` is $O(B \log n)$.

The following lemma shows that, so far, we have overestimated the number of merge and split operations performed by B-trees.

Lemma 12.1. *Starting with an empty B-tree and performing any sequence of m* `add(x)` *and* `remove(x)` *operations results in at most $3m/2$ splits, merges, and borrows being performed.*

Dokaz. The proof of this has already been sketched in Section 7.3 for the special case in which $B = 2$. The lemma can be proven using a credit scheme, in which

1. each split, merge, or borrow operation is paid for with two credits, i.e., a credit is removed each time one of these operations occurs; and

2. at most three credits are created during any `add(x)` or `remove(x)` operation.

Since at most $3m$ credits are ever created and each split, merge, and borrow is paid for with with two credits, it follows that at most $3m/2$ splits, merges, and borrows are performed. These credits are illustrated using the ¢ symbol in Figures 12.5, 12.9, and 12.10.
To keep track of these credits the proof maintains the following *credit invariant*: Any non-root node with $B-1$ keys stores one credit and any node with $2B-1$ keys stores three credits. A node that stores at least B keys and most $2B-2$ keys need not store any credits. What remains is to show that we can maintain the credit invariant and satisfy properties 1 and 2, above, during each `add(x)` and `remove(x)` operation.

Adding: The `add(x)` method does not perform any merges or borrows, so we need only consider split operations that occur as a result of calls to `add(x)`.
Each split operation occurs because a key is added to a node, `u`, that already contains $2B-1$ keys. When this happens, `u` is split into two nodes, `u'` and `u''` having $B-1$ and B keys, respectively. Prior to this operation, `u` was storing $2B-1$ keys, and hence three credits. Two of these credits can be used to pay for the split and the other credit can be given to `u'` (which has $B-1$ keys) to maintain the credit invariant.

Therefore, we can pay for the split and maintain the credit invariant during any split.
The only other modification to nodes that occur during an `add(x)` operation happens after all splits, if any, are complete. This modification involves adding a new key to some node `u`'. If, prior to this, `u`' had $2B-2$ children, then it now has $2B-1$ children and must therefore receive three credits. These are the only credits given out by the `add(x)` method.

Removing: During a call to `remove(x)`, zero or more merges occur and are possibly followed by a single borrow. Each merge occurs because two nodes, `v` and `w`, each of which had exactly $B-1$ keys prior to calling `remove(x)` were merged into a single node with exactly $2B-2$ keys. Each such merge therefore frees up two credits that can be used to pay for the merge.
After any merges are performed, at most one borrow operation occurs, after which no further merges or borrows occur. This borrow operation only occurs if we remove a key from a leaf, `v`, that has $B-1$ keys. The node `v` therefore has one credit, and this credit goes towards the cost of the borrow. This single credit is not enough to pay for the borrow, so we create one credit to complete the payment.
At this point, we have created one credit and we still need to show that the credit invariant can be maintained. In the worst case, `v`'s sibling, `w`, has exactly B keys before the borrow so that, afterwards, both `v` and `w` have $B-1$ keys. This means that `v` and `w` each should be storing a credit when the operation is complete. Therefore, in this case, we create an additional two credits to give to `v` and `w`. Since a borrow happens at most once during a `remove(x)` operation, this means that we create at most three credits, as required.
If the `remove(x)` operation does not include a borrow operation, this is because it finishes by removing a key from some node that, prior to the operation, had B or more keys. In the worst case, this node had exactly B keys, so that it now has $B-1$ keys and must be given one credit, which we create.
In either case—whether the removal finishes with a borrow operation or not—at most three credits need to be created during a call to `remove(x)` to maintain the credit invariant and pay for all borrows and merges that

occur. This completes the proof of the lemma. □

The purpose of Lemma 12.1 is to show that, in the word-RAM model the cost of splits, merges and joins during a sequence of m `add(x)` and `remove(x)` operations is only $O(Bm)$. That is, the amortized cost per operation is only $O(B)$, so the amortized cost of `add(x)` and `remove(x)` in the word-RAM model is $O(B + \log n)$. This is summarized by the following pair of theorems:

Theorem 12.1 (External Memory B-Trees)**.** *A `BTree` implements the `SSet` interface. In the external memory model, a `BTree` supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log_B n)$ *time per operation.*

Theorem 12.2 (Word-RAM B-Trees)**.** *A `BTree` implements the `SSet` interface. In the word-RAM model, and ignoring the cost of splits, merges, and borrows, a `BTree` supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log n)$ *time per operation. Furthermore, beginning with an empty `BTree`, any sequence of m* `add(x)` *and* `remove(x)` *operations results in a total of $O(Bm)$ time spent performing splits, merges, and borrows.*

12.3 Discussion and Exercises

The external memory model of computation was introduced by Aggarwal and Vitter [**?**]. It is sometimes also called the *I/O model* or the *disk access model*.

B-Trees are to external memory searching what binary search trees are to internal memory searching. B-trees were introduced by Bayer and McCreight [**?**] in 1970 and, less than ten years later, the title of Comer's ACM Computing Surveys article referred to them as ubiquitous [**?**]. Like binary search trees, there are many variants of B-Trees, including B^+-trees, B^*-trees, and counted B-trees. B-trees are indeed ubiquitous and are the primary data structure in many file systems, including Apple's HFS+, Microsoft's NTFS, and Linux's Ext4; every major database system; and key-value stores used in cloud computing. Graefe's recent survey [**?**] provides a 200+ page overview of the many modern applications, variants, and optimizations of B-trees.

B-trees implement the SSet interface. If only the USet interface is needed, then external memory hashing could be used as an alternative to B-trees. External memory hashing schemes do exist; see, for example, Jensen and Pagh [**?**]. These schemes implement the USet operations in $O(1)$ expected time in the external memory model. However, for a variety of reasons, many applications still use B-trees even though they only require USet operations.

One reason B-trees are such a popular choice is that they often perform better than their $O(\log_B n)$ running time bounds suggest. The reason for this is that, in external memory settings, the value of B is typically quite large—in the hundreds or even thousands. This means that 99% or even 99.9% of the data in a B-tree is stored in the leaves. In a database system with a large memory, it may be possible to cache all the internal nodes of a B-tree in RAM, since they only represent 1% or 0.1% of the total data set. When this happens, this means that a search in a B-tree involves a very fast search in RAM, through the internal nodes, followed by a single external memory access to retrieve a leaf.

Exercise 12.1. Show what happens when the keys 1.5 and then 7.5 are added to the B-tree in Figure **??**.

Exercise 12.2. Show what happens when the keys 3 and then 4 are removed from the B-tree in Figure **??**.

Exercise 12.3. What is the maximum number of internal nodes in a B-tree that stores n keys (as a function of n and B)?

Exercise 12.4. The introduction to this chapter claims that B-trees only need an internal memory of size $O(B + \log_B n)$. However, the implementation given here actually requires more memory.

1. Show that the implementation of the add(x) and remove(x) methods given in this chapter use an internal memory proportional to $B \log_B n$.
2. Describe how these methods could be modified in order to reduce their memory consumption to $O(B + \log_B n)$.

Exercise 12.5. Draw the credits used in the proof of Lemma 12.1 on the trees in Figures 12.6 and 12.7. Verify that (with three additional credits)

it is possible to pay for the splits, merges, and borrows and maintain the credit invariant.

Exercise 12.6. Design a modified version of a B-tree in which nodes can have anywhere from B up to $3B$ children (and hence $B-1$ up to $3B-1$ keys). Show that this new version of B-trees performs only $O(m/B)$ splits, merges, and borrows during a sequence of m operations. (Hint: For this to work, you will have to be more agressive with merging, sometimes merging two nodes before it is strictly necessary.)

Exercise 12.7. In this exercise, you will design a modified method of splitting and merging in B-trees that asymptotically reduces the number of splits, borrows and merges by considering up to three nodes at a time.

1. Let u be an overfull node and let v be a sibling immediately to the right of u. There are two ways to fix the overflow at u:

   (a) u can give some of its keys to v; or

   (b) u can be split and the keys of u and v can be evenly distributed among u, v, and the newly created node, w.

   Show that this can always be done in such a way that, after the operation, each of the (at most 3) affected nodes has at least $B+\alpha B$ keys and at most $2B-\alpha B$ keys, for some constant $\alpha > 0$.

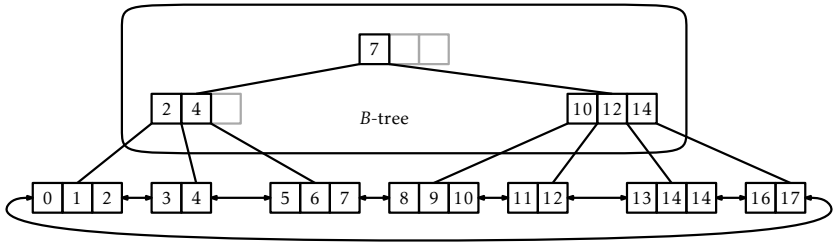
2. Let u be an underfull node and let v and w be siblings of u There are two ways to fix the underflow at u:

   (a) keys can be redistributed among u, v, and w; or

   (b) u, v, and w can be merged into two nodes and the keys of u, v, and w can be redistributed amongst these nodes.

   Show that this can always be done in such a way that, after the operation, each of the (at most 3) affected nodes has at least $B+\alpha B$ keys and at most $2B-\alpha B$ keys, for some constant $\alpha > 0$.

3. Show that, with these modifications, the number of merges, borrows, and splits that occur during m operations is $O(m/B)$.



Slika 12.11: A B^+-tree is a B-tree on top of a doubly-linked list of blocks.

Exercise 12.8. A B^+-tree, illustrated in Figure 12.11 stores every key in a leaf and keeps its leaves stored as a doubly-linked list. As usual, each leaf stores between $B-1$ and $2B-1$ keys. Above this list is a standard B-tree that stores the largest value from each leaf but the last.

1. Describe fast implementations of `add(x)`, `remove(x)`, and `find(x)` in a B^+-tree.

2. Explain how to efficiently implement the `findRange(x,y)` method, that reports all values greater than `x` and less than or equal to `y`, in a B^+-tree.

3. Implement a class, `BPlusTree`, that implements `find(x)`, `add(x)`, `remove(x)`, and `findRange(x,y)`.

4. B^+-trees duplicate some of the keys because they are stored both in the B-tree and in the list. Explain why this duplication does not add up to much for large values of B.

¿¿¿¿¿¿¿ .r70

Literatura

[1] G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(1259-1262):4, 1962.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] 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.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *SIGFIDET Workshop*, pages 107–141. ACM, 1970.

[5] 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.

[6] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[7] C. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Computer Science Department, Stanford University, 1972.

[8] 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.

[9] 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.

[10] M. Eytzinger. *Thesaurus principum hac aetate in Europa viventium (Cologne)*. 1590. In commentaries, 'Eytzinger' may appear in variant forms, including: Aitsingeri, Aitsingero, Aitsingerum, Eyzingern.

[11] 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.

[12] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[13] 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.

[14] A. Gambin and A. Malinowski. Randomized meldable priority queues. In *SOFSEM'98: Theory and Practice of Informatics*, pages 344–349. Springer, 1998.

[15] M. T. Goodrich and J. G. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In Dehne et al. [**?**], pages 205–216.

[16] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2010.

[17] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.

[18] 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.

[19] J. E. Hopcroft and R. E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[20] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[21] M. S. Jensen and R. Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008.

[22] D. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[23] D. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[24] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transaction on Electronic Computers*, EC-10(3):346–365, 1961.

[25] 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].

[26] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292, 1959.

[27] Oracle. *The Collections Framework*. URL: http://download.oracle.com/javase/1.5.0/docs/guide/collections/ [cited 2011-07-19].

[28] 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.

[29] S. M. Ross. *Probability Models for Computer Science*. Academic Press, Inc., Orlando, FL, USA, 2001.

[30] R. Sedgewick. Left-leaning red-black trees, September 2008. URL: http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf [cited 2011-07-21].

[31] 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.

[32] P. Sinha. A memory-efficient doubly linked list. *Linux Journal*, 129, 2005. URL: http://www.linuxjournal.com/article/6828 [cited 2013-06-05].

[33] 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.

[34] S. P. Thompson. *Calculus Made Easy*. MacMillan, Toronto, 1914. Project Gutenberg EBook 33283. URL: http://www.gutenberg.org/ebooks/33283 [cited 2012-06-14].

[35] 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.

[36] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[37] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

[38] J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.