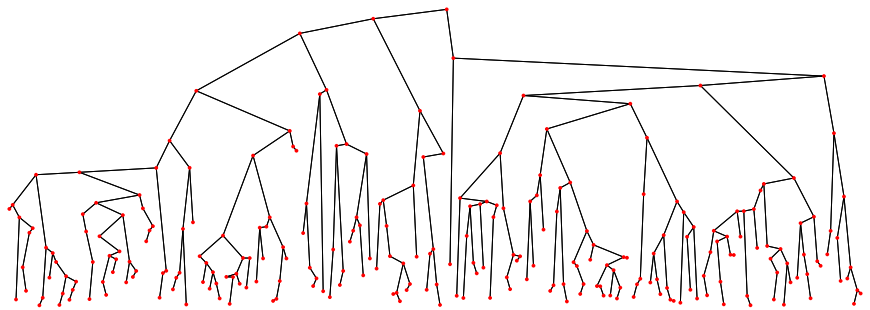


Open Data Structures (za programski jezik C++)v slovenščini

Izdaja 0.1Fβ

Pat Morin

CC BY

Kazalo

1 Uvod **1**

1.1 Zahteva po učinkovitosti 2
1.2 Mathematical Background 4
1.2.1 Eksponenti in Logaritmi 4
1.2.2 Fakulteta 5
1.2.3 Asimptotična Notacija 6
1.2.4 Naključnost in verjetnost 11
1.3 Interfaces 13
1.3.1 `Vrsta`, `Sklad`, in Deque Vmesniki 13
1.3.2 Vmesnik `seznama`: linearne sekvence 15
1.3.3 `USet` vmesnik: Neurejen set. 16
1.3.4 `SSet` vmesnik: Urejeni seti 17

2 Implementacija seznama s poljem **19**

2.1 `ArrayStack`: Implementacija sklada s poljem 21
2.1.1 Osnove 21
2.1.2 Večanje in krčenje 23
2.1.3 Povzetek 25
2.2 `FastArrayStack`: Optimiziran ArrayStack 26
2.3 `ArrayQueue`: An Array-Based Queue 27
2.3.1 Summary 30
2.4 `ArrayDeque`: Fast Deque Operations Using an Array 31
2.4.1 Summary 33
2.5 `DualArrayDeque`: Building a Deque from Two Stacks 33
2.5.1 Balancing 37
2.5.2 Summary 39

2.6 RootishArrayStack: A Space-Efficient Array Stack 39
2.6.1 Analysis of Growing and Shrinking 44
2.6.2 Space Usage 44
2.6.3 Summary 45
2.6.4 Computing Square Roots 46
2.7 Discussion and Exercises 49

3 Povezani seznam **53**
3.1 SLList: Enojno povezani seznam 54
3.1.1 Operaciji Vrste 56
3.1.2 Povzetek 56
3.2 DLList: Dvojno povezan seznam 57
3.2.1 Dodajanje in odstranjevanje 59
3.2.2 Povzetek 61

4 Preskočni seznami **63**
4.1 Osnovna struktura 63
4.2 SkiplistSSet: Učinkovit SSet 65
4.2.1 Povzetek 68
4.2.2 Summary 69
4.3 SkiplistList: An Efficient Random-Access List 69
4.3.1 Summary 74
4.4 Analysis of Skiplists 74
4.5 Discussion and Exercises 78

5 Dvojiška drevesa **79**
5.1 BinaryTree: Osnovno Binarno Drevo 81
5.1.1 Rekurzivni algoritmi 82
5.1.2 Obiskovanje Binarnega drevesa 82
5.2 BinarySearchTree: Neuravnoteženo binarno iskalno drevo 85
5.2.1 Iskanje 86
5.2.2 Vstavljanje 87
5.2.3 Brisanje 90
5.2.4 Povzetek 92

6 Naključna iskalna binarna drevesa **93**
6.1 Naključna iskalna binarna drevesa . . . 93
6.1.1 Dokaz Lemma 6.1 . . . 96
6.1.2 Povzetek . . . 98
6.2 Treap: Naključno generirano binarno iskalno drevo . . . 99
6.2.1 Povzetek . . . 106

7 Red-Black Trees **109**
7.1 2-4 Trees . . . 110
7.1.1 Adding a Leaf . . . 111
7.1.2 Removing a Leaf . . . 111
7.2 RedBlackTree: A Simulated 2-4 Tree . . . 114
7.2.1 Rdeče-Črna drevesa in 2-4 Drevesa . . . 114
7.2.2 Levo-viseca Rdece-Crna Drevesa . . . 118
7.2.3 Dodajanje . . . 120
7.2.4 Removal . . . 123
7.3 Summary . . . 128
7.4 Discussion and Exercises . . . 130

8 Kopice **135**
8.1 BinarnaKopica: inplicitno binarno drevo . . . 135
8.1.1 Summary . . . 139
8.2 MeldableHeap: A Randomized Meldable Heap . . . 141
8.2.1 Analysis of merge(h1,h2) . . . 144
8.2.2 Summary . . . 145
8.3 Discussion and Exercises . . . 146

9 Graphs **151**
9.1 AdjacencyMatrix: Representing a Graph by a Matrix . . . 153
9.2 AdjacencyLists: A Graph as a Collection of Lists . . . 156
9.3 Graph Traversal . . . 160
9.3.1 Breadth-First Search . . . 160
9.3.2 Depth-First Search . . . 162
9.4 Discussion and Exercises . . . 165

10 Podatkovne strukture za cela števila **169**
10.1 BinaryTrie: digitalno iskalno drevo 170
10.2 XFastTrie: Iskanje v dvojnem logaritmičnem času 176
10.3 YFastTrie: A Doubly-Logarithmic Time SSet 179
10.4 Discussion and Exercises 184

11 Iskanje v zunanjem pomnilniku **187**
11.1 Block Store 189
11.2 B-drevesa 189
11.2.1 Searching 191
11.2.2 Addition 194
11.2.3 Removal 199
11.2.4 Amortized Analysis of *B*-Trees 205
11.3 Discussion and Exercises 208

12 External Memory Searching **213**
12.1 The Block Store 215
12.2 B-drevesa 216
12.2.1 Searching 218
12.2.2 Addition 220
12.2.3 Removal 225
12.2.4 Amortized Analysis of *B*-Trees 231
12.3 Discussion and Exercises 234

Bibliografija **239**

Kazalo **243**

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. To 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. To ni preprosto: 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 spomina.

- Vpis v socialno omrežje: Omrežni strežniki uporabljajo naše vpisne podatke za vpogled v naš račun. To ni preprosto: največja socialna omrežja imajo stotine miljonov aktivnih uporabnikov.

- Spletno iskanje: Iskalniki uporabljajo podatkovne strukture za iskanje spletnih strani, ki vsebujejo naše iskalne pojme. To ni prepro-

sto: v Internetu je čez 8.5 miljard spletnih strani, kjer vsaka vsebuje veliko potencialnih iskalnih pojmov.

- Številke za klice v sili(1-1-2, 1-1-3): 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; 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 uporabljanih podatkovnih struktur. Vsak s 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 čez 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 sekund kar pomeni, da bi zahtevalo najmanj $4,500 \times 8.5 = 38,250$ zelo hitrih stežnikov samo za vzdrževanje.

Rešitev: Ti primeri nam povejo, da preproste implementacije podatkovnih struktur ne delujejo ko sta tako število elementov, n, v podatkovni strukturi kot tudi število operacij, m, opravljenih na podatkivni 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. Čepra se sliši nemogoče bomo spoznali podatkovne strukture, kjer iskanje zahteva primerjavo samo dveh elementov v povpreču, neodvisno od števila elementov v podatkovni strukturi. V naštem računalniku, ki opravi milijardo operacij na sekundo zahteva iskanje v podatkovni strukturi, ki vsebuje milijardo elementov (ali trilijardo, štirijardo ali celo petiljardo elementov), 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. Naprimer, lahko vzdržujemo sortiran niz milijarde elementov, med poizvedbo do največ 60 elementov med katerokoli operacijo. V naštem računalniku, ki opravi milijardo operacij na sekundo zahteva iskanje v podatkovni strukturi, ki vsebuje milijardo elementov, samo 0.000000002 sekund.

Preostanek tega poglavja vsebuje kratek pregled glavnih pojmov, ki se uporabljajo 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 govorijo o:

- pregled matematičnega dela, ki vključuje eksponente, logaritme, fakultete, asimptotično (veliki O) notacijo, verjetnost in naključnost;
- računski model;
- pravilnost, časovna zahtevnost in prostorska zahtevnost;
- pregled ostalih poglavij; in
- vzorčne kode in navodila za pisanje

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

1.2 Mathematical Background

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 vstreznih poglavji iz zelo dobre in zastonj knjige o znanosti iz matematike in računalništva [24].

1.2.1 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 *logaritm 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 logaritm*. 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 pogoste operacije 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.

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

1.2.2 Fakulteta

V enem ali dveh delih knjige je uporabljena *fakulteta* Za nenegativna cela števila n, je uporabljena notacija $n!$ (izgovorjena kot "n fakulteta") in po-

meni naslednje:

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

Fakulteta se pojavi, ker je $n!$ število različnih permutaciji, 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 $\{1, \ldots, n\}$.

1.2.3 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 inštru 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-Oh 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 $g(n)$ za katere velja: $c \cdot f(n)$ začne prevladovati $g(n)$ 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 pisali stvari npr. $f_1(n) = O(f(n))$, medtem ko s to notacijo 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. Zelo čuden 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, nemoremo 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 računalniški znanosti. Uporabljal jo je številčni teoretik Paul Bachmann že leta 1893 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]`), and
- `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, 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 splošnjem ne bo mogoče primerjati dveh časov izvedbe, da bi razločili kateri je hitrejši, brez da bi vedeli vrednosti konstant. Tudi če uspemo določiti te konstante (npr. z časovnimi testi), bi naša ugotovitev veljala samo za napravo na kateri smo izvajali teste.

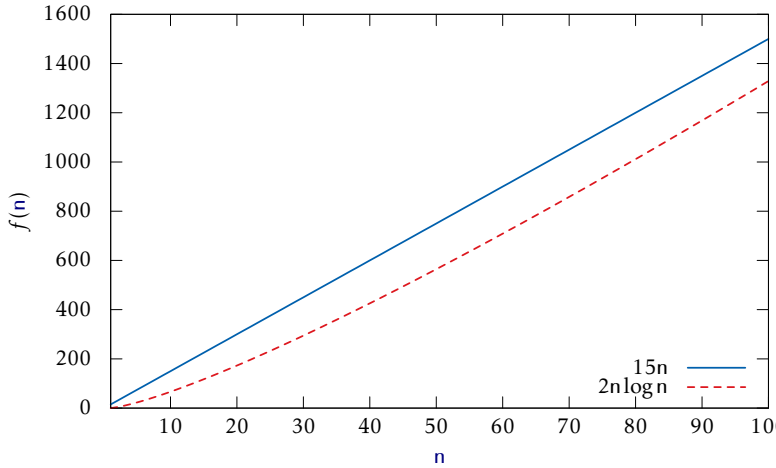
Velika-O notacija daje smisel analiziranju zapletenih funkcij pri višjih stopnjah. Če imata dva algoritma enako veliko-O časovno izvedbo, potem ne moremo točno vedeti kateri je hitrejši in ni očitnega zmagovalca. En algoritem je lahko hitrejši na eni napravi, drugi pa na drugi napravi. Če imata dva algoritma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

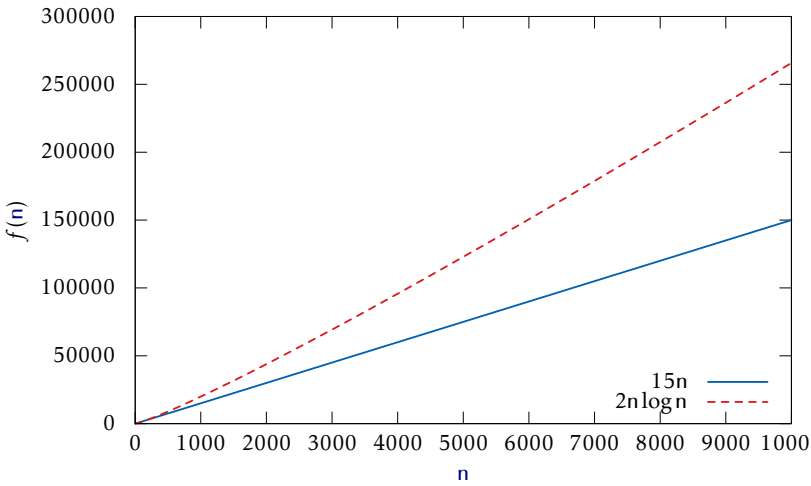
Kako lahko primerjamo veliko-O notacijo dveh različnih funkcij, prikazuje Figure 1.1, ki primerja stopnjo rasti $f_1(\mathtt{n}) = 15\mathtt{n}$ proti $f_2(n) = 2\mathtt{n}\log\mathtt{n}$ Npr., da je $f_1(n)$ časovna komplesnost zapletenega linearnega časovnega algoritma in je $f_2(n)$ časovna kompleksnost 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 n, velja nasprotno za velike vrednosti 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:

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

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 defini-





Slika 1.1: Plots of $15n$ versus $2n \log n$.

cijo $O(f(n))$ ko je $f(n)$ naraščujoča funkcija n. Bralci naj bodo pozorni, da je lahko v drugih besedilih uporabljena asimptotična notacija drugače.

1.2.4 Naključnost in verjetnost

Nekatere podatkovne strukture predstavljene v knjigi so *naključne*; odločaj se naključno in neodvisno od podatkov ki so spravljeni v njih in od opraciji 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, naključna spremenljivka, želimo pa preučevati njeno *pričakovano vrednost*.

Za diskretno naključno spremenljivko X, ki zavzame vrednosti neke univerzalne mnozice 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 operaciji izvedene na podatkovni strukturi.

Ena pomembnejših lastnosti pričakovane verjetnosti je *linearnost pričak*
Za katerekoli dve naključne spremenljivke X in Y,

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

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 najboljše ponazorjene s primerom. Npr. da 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 $i\binom{k}{i} = k\binom{k-1}{i}$ and $\sum_{i=0}^{k}\binom{k}{i} = 2^k$.

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

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

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.

1.3 Interfaces

Ko govorimo o podatkovnih strukturah, se moramo zavedati razlike med vmesnikom podatkovne strukture in njegovo implementacijo. Vmesnih opisuje kaj podatkovna struktura počne, medtem ko implementacija opisuje, kako to počne.

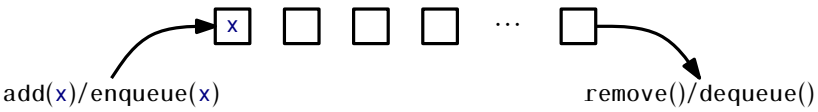
vmesnik, včasih imenovana tudi *abstrakten podatkovni tip*, definira set operaciji ki so podprte s strani podatkovne strukture in semantiko oziroma pomenom teh operacij. Vmesnih nam ne pove nič o temu kako podatkovna struktura implementira te operacije; pove nam samo katere operacije so podprte, vključno z 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. Zatorej imamo lahko veliko implementaciji 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.

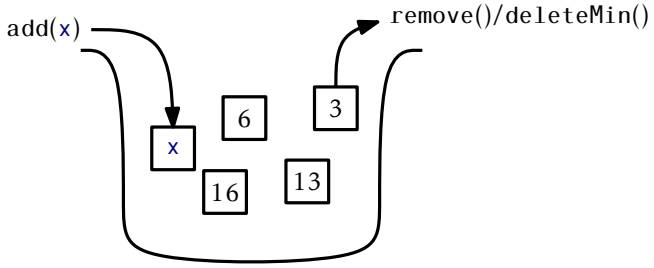
1.3.1 `Vrsta`, `Sklad`, in `Deque` Vmesniki

Vmesnik `Vrste` predstavlja zbirko elementov med katere lahko dodamo ali izbrisemo naslednji element. Bolj natančno, operaceije podprte z vmesnikom `vrste` so

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



Slika 1.2: FIFO vrsta.



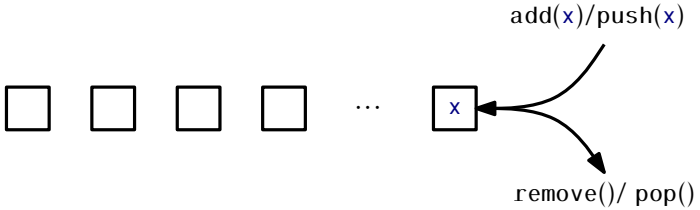
Slika 1.3: Vrsta s prednostjo.

vrni y

Opazimo lahko da metoda remove() ne sprejme nobenega argumenta. Implementacija vrste odloča kateri element bo izbrisan iz vrste. Poznamo veliko implementaciji vrste, najbolj pogoste pa so FIFO, vrste s prednostjo in LIFO. *FIFO (first-in-first-out)* vrsta, ki je narisana v Figure 1.2, 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 ključejo enqueue(x) oziroma dequeue(x)

Vrste s prednostjo, prikazane na Figure 1.3, 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().

Zelo pogosta implementacija vrste je LIFO (last-in-first-out) prikazana na Figure 1.4. Na *LIFO vrsti* je izbrisan nazadnje dodan element. To je najboljse prikazano s kupom krožnikov; krožniki so postavljeni na vrh kupa, pravtako so odstranjeni iz vrha kupa. Ta struktura je tako pogosta,



Slika 1.4: sklad.

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.

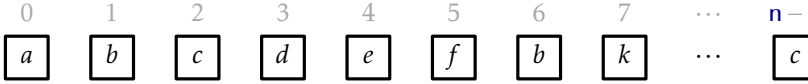
Deque je generalizacija obeh FIFO vrste in LIFO vrste (sklad). Deque predstavlja sekvenco elementov, z začetkom in koncem. Elementi so lahko dodani na začetek ali pa na konec. Imena deque so samoumevna: addFirst removeFirst(), addLast(x), in removeLast(). Sklad je lahko implementiran samo z uporabo addFirst(x)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

1.3.2 Vmesnik seznama: linearne sekvence

Ta knjiga govori zelo malo o FIFO vrsti, skladu ali deque vmesnikih, ker so vmesniki vključeni z vmesnikom seznama. Seznam,

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 $x_i = x$
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:



Slika 1.5: Seznam predstavlja sekvenco indeksov $0, 1, 2, \ldots, n-1$. V tem seznamu, bi klic get(2) vrnil vrednost c.

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

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, zelimo poudariti, da lahko te podatkovne strukture uporabimo za implementacijo vmesnika sklada in deque zelo efektivno. Na primer, ArrayDeque razred je implementacije vmesnika seznama ki implementira vse deque operacije v konstantnem času na operacijo.

1.3.3 USet vmesnik: Neurejen set.

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, false otherwise.

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 najdli, od elementa `y`, element ki ga bomo verjetno odstranili ali najdli. 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`)m bi rezultat `y` = (`k`,`v`). Z drugimi besedami povedano, možno je dobiti vrednost `v`, če podamo samo ključ `k`.

1.3.4 SSet vmesnik: Urejeni seti

SSet vmesnik 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 USet vmesnik. Razlika med USet in SSet 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 po-

navadi pride s ceno, da metoda porabi več časa za iskanje in večjo kompleknostjo kode. Na primer, večina implementacij SSeta omenjenih v tej knjigi imajo find(x) operacije, ki potrebujejo logaritmičen čas glede na velikost podatkov. Na drugi strani, implementacija USeta kot ChainedHashTable v Chapter **??** ima 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.

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 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`	$O(1)$	$O(n-i)$

V podatkovnih strukturah, kjer podatke shranjujemo v enojno polje imajo veliko prednosti, a tudi omejitev:

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

- Polja niso dinamična. Če želimo vstaviti ali pa 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 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. Če natančneje pogledamo, če začnemo s prazno podatkovno strukturo in izvedemo sekvenco 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 indexu 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

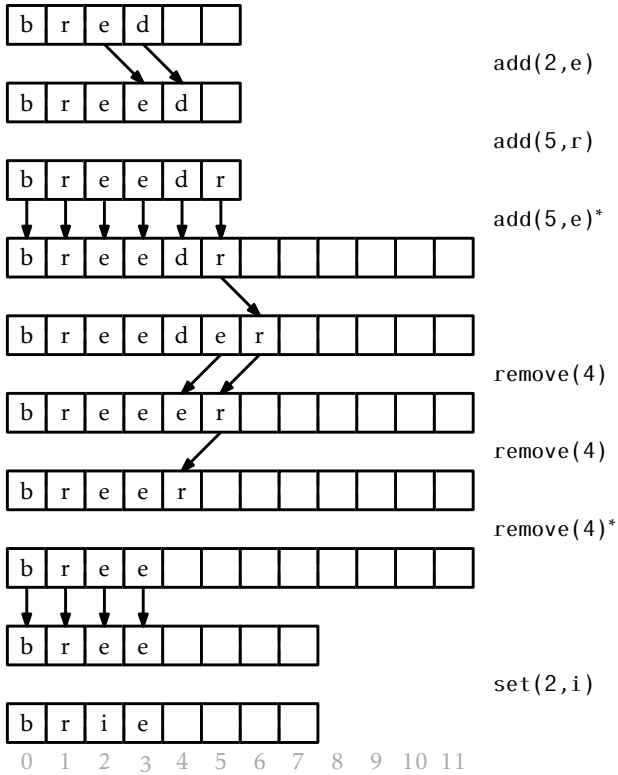
```
array<T> a;
int n;
int size() {
  return n;
}
```

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: Sekvenca 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],...,a[n − 1] za ena v desno, da naredimo prostor za x, množico a[i] spravimo v x, ain 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 resiz 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; naredi novo polje b velikosti $2n$ in kopira 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 kompleksnosti 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 ušteta 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 `remov` operacijo. Namesto tega, se posveti ceni vseh klicev `resize()` med sekvenco m klicov funkcije `add(i,x)` ali `remove(i)`. Predvsem pokažemo:

Lemma 2.1. *Če je ustvarjen prazen* `ArrayList` *in katerakoli sekvenca, 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 `resize()` klican, 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 the $(i-1)$tim in $i$timi 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` $= n = n_i$. Če pogledamo prejšnji klic funkcije `resize()`: po tem klicu, je bila velikost `a`-ja 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()` zgoditi 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()` nako vsaj $n_i/2-1$, oliko kot je tudi potrebno za dokončanje dokaza. □

2.1.3 Povzetek

Naslednji izrek povzema kvaliteto izvedbe `ArrayStack`-a:

Theorem 2.1. *`ArrayStack` implementira `List` vmesnik. Če ignoriramo ceno klicev funkcije* `resize()`*, potem `ArrayStack` podpira slednje 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] The -1 in this formula accounts for the special case that occurs when `n` $= 0$ and `a.length` $= 1$.

Poleg tega, če začnemo z praznim ArrayStack-om in potem izvajamo katerokoli zaporedje od m `add(i,x)` *in* `remove(i)` *operacij to 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)` n funkcijo `pop()` kot `remove(n-` V tem primeru bodo te operacije potrebovale $O(1)$ amortiziranega časa.

2.2 FastArrayStack: Optimiziran ArrayStack

Večina opravljenega dela `ArrayStack`-a vključuje premikanje (z `add(i,x)` in `remove(i)`) in kopiranjem (s `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 `memcpy(d,s,n)` in `memmove(d,s,n)` funkcije. V C++ jeziku je `std::copy(a0,a1,b)` algoritem. V Javi je metoda `System.arraycopy(s,i,d,j`

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++;
}
```

e funkcije so ponavadi zelo optimizirane in lahko uporabljajo tudi posebne strojne ukaze, ki lahko kopirajo veliko hitreje, kot z uporabo zanke `for`. Čeprav s pomočjo teh funkcij ne moremo asimptotično zmanjšati izvajalnih časov, je ta optimizacija še vedno koristna. V teh C++ izvedbah Jave, uporaba nativnega `std::copy(a0,a1,b)` povzroči pohitritve za faktor

med 2 in 3, odvisno od vrste izvajanih operacij. Naše izkušnje se lahko razlikujejo.

2.3 ArrayQueue: An Array-Based Queue

In this section, we present the ArrayQueue data structure, which implements a FIFO (first-in-first-out) queue; elements are removed (using the remove() operation) from the queue in the same order they are added (using the add(x) operation).

Notice that an ArrayStack is a poor choice for an implementation of a FIFO queue. It is not a good choice because we must choose one end of the list upon which to add elements and then remove elements from the other end. One of the two operations must work on the head of the list, which involves calling add(i, x) or remove(i) with a value of i = 0. This gives a running time proportional to n.

To obtain an efficient array-based implementation of a queue, we first notice that the problem would be easy if we had an infinite array a. We could maintain one index j that keeps track of the next element to remove and an integer n that counts the number of elements in the queue. The queue elements would always be stored in

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

Initially, both j and n would be set to 0. To add an element, we would place it in $a[j+n]$ and increment n. To remove an element, we would remove it from $a[j]$, increment j, and decrement n.

Of course, the problem with this solution is that it requires an infinite array. An ArrayQueue simulates this by using a finite array a and *modular arithmetic*. This is the kind of arithmetic used when we are talking about the time of day. For example 10:00 plus five hours gives 3:00. Formally, we say that

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

We read the latter part of this equation as "15 is congruent to 3 modulo 12." We can also treat mod as a binary operator, so that

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

More generally, for an integer a and positive integer m, a mod m is the unique integer $r \in \{0,\ldots,m-1\}$ such that $a = r + km$ for some integer k. Less formally, the value r is the remainder we get when we divide a by m. In many programming languages, including C++, the mod operator is represented using the % symbol.[2]

Modular arithmetic is useful for simulating an infinite array, since `i` mod `a.length` always gives a value in the range $0,\ldots,$ `a.length` -1. Using modular arithmetic we can store the queue elements at array locations

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

This treats the array `a` like a *circular array* in which array indices larger than `a.length` -1 "wrap around" to the beginning of the array.

The only remaining thing to worry about is taking care that the number of elements in the `ArrayQueue` does not exceed the size of `a`.

ArrayQueue

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

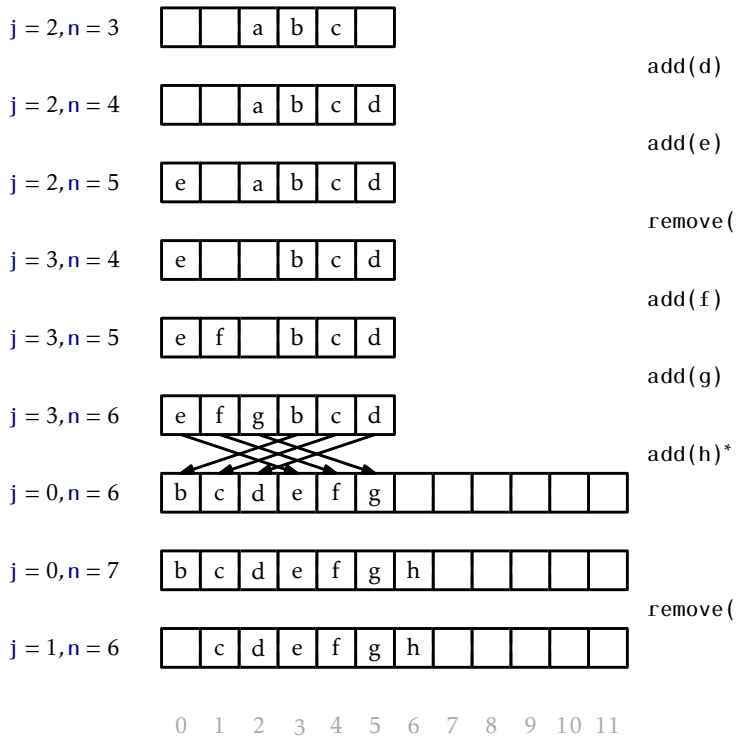
A sequence of `add(x)` and `remove()` operations on an `ArrayQueue` is illustrated in Figure 2.2. To implement `add(x)`, we first check if `a` is full and, if necessary, call `resize()` to increase the size of `a`. Next, we store `x` in `a[(j+n)%a.length]` and increment `n`.

ArrayQueue

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

To implement `remove()`, we first store `a[j]` so that we can return it later. Next, we decrement `n` and increment `j` (modulo `a.length`) by setting

[2] This is sometimes referred to as the *brain-dead* mod operator, since it does not correctly implement the mathematical mod operator when the first argument is negative.



Slika 2.2: A sequence of `add(x)` and `remove(i)` operations on an `ArrayQueue`. Arrows denote elements being copied. Operations that result in a call to `resize()` are marked with an asterisk.

`j = (j + 1) mod a.length`. Finally, we return the stored value of `a[j]`. If necessary, we may call `resize()` to decrease the size of `a`.

ArrayQueue

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

Finally, the `resize()` operation is very similar to the `resize()` operation of `ArrayStack`. It allocates a new array `b` of size `2n` and copies

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

onto

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

and sets `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 Summary

The following theorem summarizes the performance of the `ArrayQueue` data structure:

Theorem 2.2. *An `ArrayQueue` implements the (FIFO) `Queue` interface. Ignoring the cost of calls to `resize()`, an `ArrayQueue` supports the operations `add(x)` and `remove()` in $O(1)$ time per operation. Furthermore, beginning with an empty `ArrayQueue`, any sequence of m `add(i,x)` and `remove(i)` operations results in a total of $O(m)$ time spent during all calls to `resize()`.*

2.4 ArrayDeque: Fast Deque Operations Using an Array

The ArrayQueue from the previous section is a data structure for representing a sequence that allows us to efficiently add to one end of the sequence and remove from the other end. The ArrayDeque data structure allows for efficient addition and removal at both ends. This structure implements the List interface by using the same circular array technique used to represent an ArrayQueue.

ArrayDeque

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

The get(i) and set(i,x) operations on an ArrayDeque are straightforward. They get or set the array element a[(j + i) mod a.length].

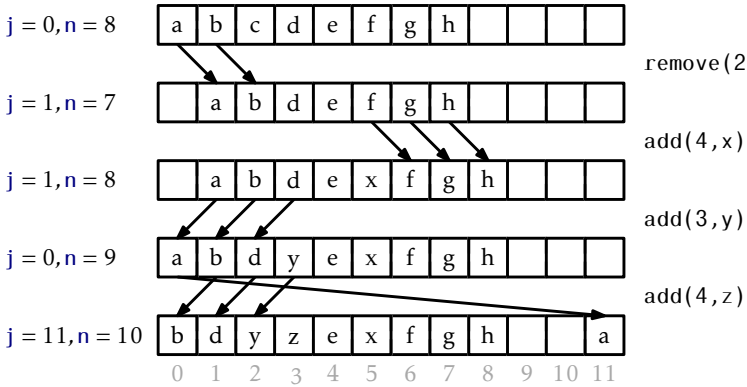
ArrayDeque

```
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;
}
```

The implementation of add(i,x) is a little more interesting. As usual, we first check if a is full and, if necessary, call resize() to resize a. Remember that we want this operation to be fast when i is small (close to 0) or when i is large (close to n). Therefore, we check if $i < n/2$. If so, we shift the elements $a[0],\ldots,a[i-1]$ left by one position. Otherwise ($i \geq n/2$), we shift the elements $a[i],\ldots,a[n-1]$ right by one position. See Figure 2.3 for an illustration of add(i,x) and remove(x) operations on an ArrayDeque.

ArrayDeque

```
void add(int i, T x) {
  if (n + 1 > a.length)  resize();
```



Slika 2.3: A sequence of add(i,x) and remove(i) operations on an ArrayDeque. Arrows denote elements being copied.

```
  if (i < n/2) { // shift a[0],..,a[i-1] left one position
    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++;
}
```

By doing the shifting in this way, we guarantee that add(i,x) never has to shift more than $\min\{i, n-i\}$ elements. Thus, the running time of the add(i,x) operation (ignoring the cost of a resize() operation) is $O(1+\min\{i, n-i\})$.

The implementation of the remove(i) operation is similar. It either shifts elements $a[0],\ldots,a[i-1]$ right by one position or shifts the elements $a[i+1],\ldots,a[n-1]$ left by one position depending on whether $i < n/2$. Again, this means that remove(i) never spends more than $O(1+\min\{i, n-i\})$ time to shift elements.

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 Summary

The following theorem summarizes the performance of the ArrayDeque data structure:

Theorem 2.3. *An ArrayDeque implements the* `List` *interface. Ignoring the cost of calls to* `resize()`*, an ArrayDeque supports the operations*

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

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

2.5 DualArrayDeque: Building a Deque from Two Stacks

Next, we present a data structure, the DualArrayDeque that achieves the same performance bounds as an ArrayDeque by using two ArrayStacks. Although the asymptotic performance of the DualArrayDeque is no better than that of the ArrayDeque, it is still worth studying, since it offers

a good example of how to make a sophisticated data structure by combining two simpler data structures.

A DualArrayDeque represents a list using two ArrayStacks. Recall that an ArrayStack is fast when the operations on it modify elements near the end. A DualArrayDeque places two ArrayStacks, called front and back, back-to-back so that operations are fast at either end.

DualArrayDeque

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

A DualArrayDeque does not explicitly store the number, n, of elements it contains. It doesn't need to, since it contains n = front.size() + back.size() elements. Nevertheless, when analyzing the DualArrayDeque we will still use n to denote the number of elements it contains.

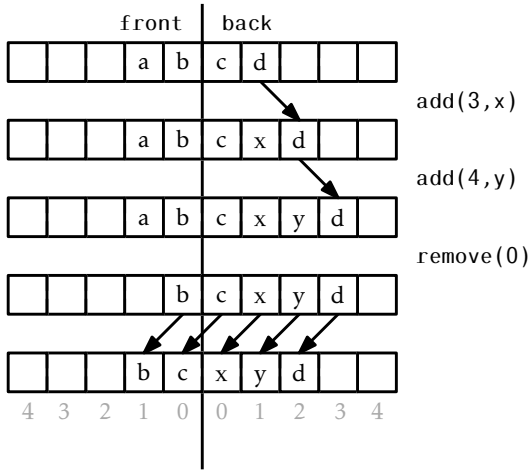
DualArrayDeque

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

The front ArrayStack stores the list elements that whose indices are $0,\ldots,$ front.size() -1, but stores them in reverse order. The back ArrayStack contains list elements with indices in front.size(),..., size() -1 in the normal order. In this way, get(i) and set(i,x) translate into appropriate calls to get(i) or set(i,x) on either front or back, which take $O(1)$ time per operation.

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);
```



Slika 2.4: A sequence of `add(i,x)` and `remove(i)` operations on a `DualArrayDeque`. Arrows denote elements being copied. Operations that result in a rebalancing by `balance()` are marked with an asterisk.

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

Note that if an index `i` < `front.size()`, then it corresponds to the element of `front` at position `front.size()`$-$`i`-1, since the elements of `front` are stored in reverse order.

Adding and removing elements from a `DualArrayDeque` is illustrated in Figure 2.4. The `add(i,x)` operation manipulates either `front` or `back`, as appropriate:

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();
```

```
}
```

The add(i,x) method performs rebalancing of the two ArrayStacks front and back, by calling the balance() method. The implementation of balance() is described below, but for now it is sufficient to know that balance() ensures that, unless size() < 2, front.size() and back.size() do not differ by more than a factor of 3. In particular, $3 \cdot$ front.size() $\geq$ back.size() and $3 \cdot$ back.size() $\geq$ front.size().

Next we analyze the cost of add(i,x), ignoring the cost of calls to balance(). If i < front.size(), then add(i,x) gets implemented by the call to front.add(front.size() − i − 1,x). Since front is an ArrayStack, the cost of this is

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

On the other hand, if i ≥ front.size(), then add(i,x) gets implemented as back.add(i − front.size(),x). The cost of this is

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

Notice that the first case (2.1) occurs when i < n/4. The second case (2.2) occurs when i ≥ 3n/4. When n/4 ≤ i < 3n/4, we cannot be sure whether the operation affects front or back, but in either case, the operation takes $O(\texttt{n}) = O(\texttt{i}) = O(\texttt{n} - \texttt{i})$ time, since i ≥ n/4 and n − i > n/4. Summarizing the situation, we have

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

Thus, the running time of add(i,x), if we ignore the cost of the call to balance(), is $O(1 + \min\{\texttt{i}, \texttt{n} - \texttt{i}\})$.

The remove(i) operation and its analysis resemble the add(i,x) operation and analysis.

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 Balancing

Finally, we turn to the `balance()` operation performed by `add(i,x)` and `remove(i)`. This operation ensures that neither `front` nor `back` becomes too big (or too small). It ensures that, unless there are fewer than two elements, each of `front` and `back` contain at least $n/4$ elements. If this is not the case, then it moves elements between them so that `front` and `back` contain exactly $\lfloor n/2 \rfloor$ elements and $\lceil n/2 \rceil$ elements, respectively.

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;
  }
}
```

Here there is little to analyze. If the `balance()` operation does rebalan-

cing, then it moves $O(n)$ elements and this takes $O(n)$ time. This is bad, since `balance()` is called with each call to `add(i,x)` and `remove(i)`. However, the following lemma shows that, on average, `balance()` only spends a constant amount of time per operation.

Lemma 2.2. *If an empty `DualArrayDeque` is created and any sequence of $m \geq 1$ calls to `add(i,x)` and `remove(i)` are performed, then the total time spent during all calls to `balance()` is $O(m)$.*

Dokaz. We will show that, if `balance()` is forced to shift elements, then the number of `add(i,x)` and `remove(i)` operations since the last time any elements were shifted by `balance()` is at least $n/2 - 1$. As in the proof of Lemma 2.1, this is sufficient to prove that the total time spent by `balance()` is $O(m)$.

We will perform our analysis using a technique knows as the *potential method*. Define the *potential*, Φ, of the `DualArrayDeque` as the difference in size between `front` and `back`:

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

The interesting thing about this potential is that a call to `add(i,x)` or `remove(i)` that does not do any balancing can increase the potential by at most 1.

Observe that, immediately after a call to `balance()` that shifts elements, the potential, Φ_0, is at most 1, since

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

Consider the situation immediately before a call to `balance()` that shifts elements and suppose, without loss of generality, that `balance()` is shifting elements because $3\texttt{front.size()} < \texttt{back.size()}$. Notice that, in this case,

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

Furthermore, the potential at this point in time is

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

Therefore, the number of calls to `add(i,x)` or `remove(i)` since the last time `balance()` shifted elements is at least $\Phi_1 - \Phi_0 > \texttt{n}/2 - 1$. This completes the proof. □

2.5.2 Summary

The following theorem summarizes the properties of a `DualArrayDeque`:

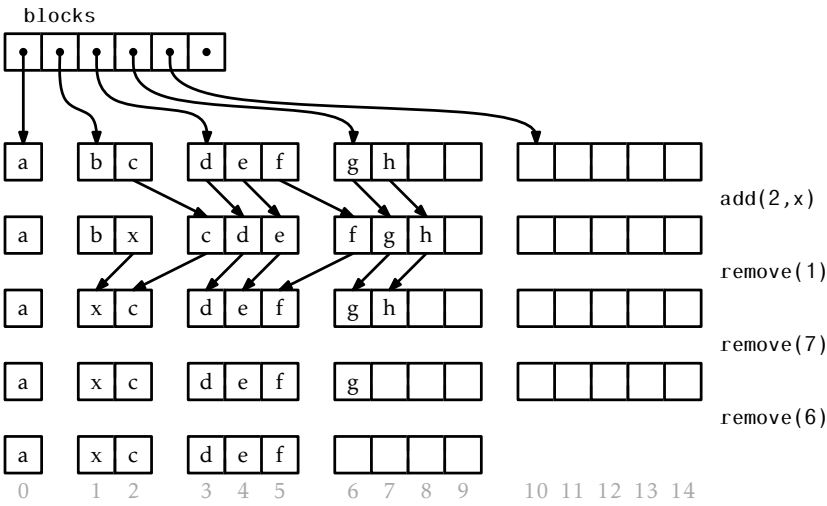
Theorem 2.4. *A `DualArrayDeque` implements the `List` interface. Ignoring the cost of calls to `resize()` and `balance()`, a `DualArrayDeque` supports the operations*

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

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

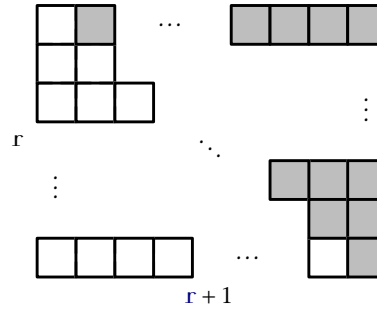
$$1 + 2 + 3 + \cdots + r = r(r+1)/2$$

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,...,b is $(b+1)(b+2)/2$. Therefore, b is the smallest integer such that

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

We can rewrite this equation as

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

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

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

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 **??**) 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

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

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 **??** for a discussion of how memory is measured.

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

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

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 [21, Section 2.2.2].

Brodnik *et al.* [5] 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 [15]. 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.a 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 **??**.)

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 **??** 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, tokrat z uporabo podatkovnih struktur, ki uporabljajo kazalce, namesto z uporabo 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 enojno povezane sezname, s katerimi lahko implementiramo Sklad in (FIFO) Vrste s konstantim časom na operacijo. Nato si bomo pogledali še dvojno povezani seznam, s katerim lahko implementiramo Deque operacije v konstantnem času (Deque - vrsta pri kateri lahko dodajamo ter odstanjujemo elemente iz začetka ali konca vrste).

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 skozi celoten 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: Enojno povezani seznam

Enojno 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 učinkovitost delovanja uporablja SLList spremnljivki head in tail za beleženje prvega ter zadnjega vozlišča. Za beleženje dolžine seznama, pa hrani celoštevilsko spremenljivko n:

SLList

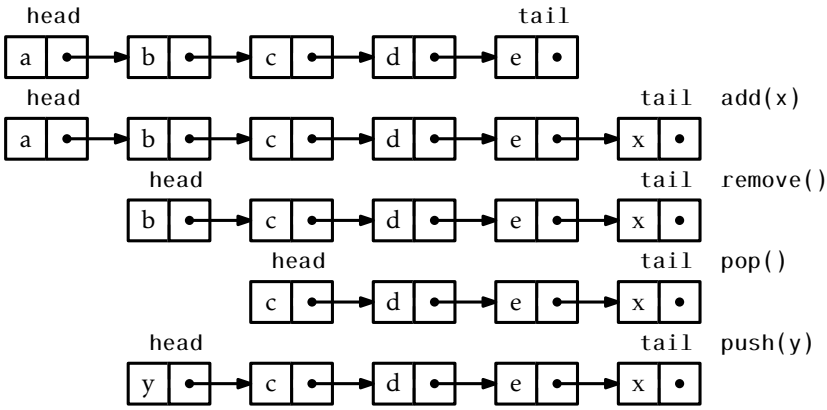
```
Node *head;
Node *tail;
int n;
```

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

Enojno povezani seznam lahko učinkovito implementira operaciji Skla to sta push(x) in pop(), s katerima dodajamo ter odstanjujemo elemente iz začetka seznama. Operacija push(x) kreira novo vozlišče u z vrednostjo x, u.next kaže na stari začetek seznama in u postane nov začetek seznama. Na koncu še povečamo vrednost n, saj se je velikost seznama povečala 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 (push(x) in pop()) nad enojno povezanim seznamom.

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

Operacija pop() preveri ali je enojno povezani seznam prazen. če ni prazen, odstrani začetno vozlišče, tako da spremeni vrednost vozlišča `head = head.next` in zmanjša spremenljivko `n` za 1. Poseben primer je, če odstranimo zadnje vozlišče, 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;
}
```

Obe od operacij push(x) in pop() imata časovno kompleksnost $O(1)$.

3.1.1 Operaciji Vrste

Enojno povezani seznam lahko implementira tudi operaciji FIFO vrste, to sta `add(x)` in `remove()`, v konstantnem času. Odstanjujemo vozlišča iz začetka seznama, operacija je enaka kakor operacija `pop()`:

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;
}
```

Dodajamo pa vozlišča na konec seznama. V večini primerov to naredimo tako, da postavimo `tail.next = u`, kjer je `u` novo nastalo vozlišče in vsebuje vrednost `x`. Poseben primer je takrat, ko je `n = 0`, takrat je `tail = head = null`. V tem primeru oba `tail` in `head` 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 od operacij `add(x)` in `remove()` rabita konstanti čas.

3.1.2 Povzetek

Sledeči izrek povzame zmožnosti enojno povezanega seznama `SLList`:

Theorem 3.1. *Enojno povezani seznam* `SLList` *impelmentira vmesnike* `Sklad` *in (FIFO)* `Vrste`*. Operacije* `push(x)`, `pop()`, `add(x)` *in* `remove()` *potrebujejo* $O(1)$ *časa na operacijo.*

Enojno povezani seznam `SLList` implementira skoraj vse operacije `Degue` vrste. Edina manjkajoča operacija je odstranjevanje elementov s konca enojno povezanega seznama. Brisanje iz konca enojno povezanega seznama je težavno, saj moramo posodobiti vrednost `tail`, tako da kaže na vozlišče `w`, vozlišče `w` 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, začenjši z vozliščem `head`, za kar pa potrebujemo $n-2$ korakov.

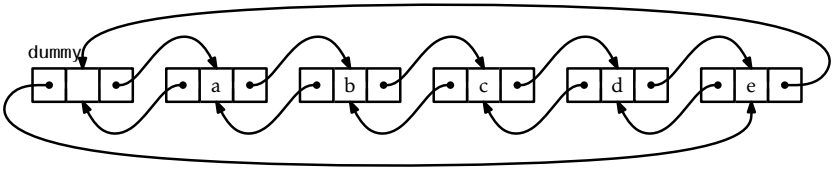
3.2 `DLList`: Dvojno povezan seznam

`DLList` (dvojno povezan seznam) je zelo podoben `SLList` le da ima vsako vozlišče `u` v `DLList` sklicevanja na obe 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 videli, da je bilo vedno več posebnih primerov za katere moramo skrbeti. Na primer, odstranjevanje zadinjega elementa iz `SLList`, ali pa dodajanje elementa v prazno `SLList` moramo biti pazljivi da se zagotovi, da sta `glava` in `rep` pravilno posodobljena. V `DLList` se število teh posebnih primerov znatno poveča. Morda najboljši način, da poskrbimo za vse te posebne primere v `DLList`, je uvesti `dummy` 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 vozlišče, ki sledi zadnjemu vozlišču n seznamu in da je pred prvim vozliščom v seznamu. Na ta način so vozlišča v seznamu (dvojno-) povezana v cikel, kot je prikazano naFigure 3.2.



Slika 3.2: A DLList containing a,b,c,d,e.

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 sedaj prav tako enostavni. Najprej smo našli 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 bi našli `i`-to vozlišče, in je zato $O(1+\min\{i,n-i\})$..

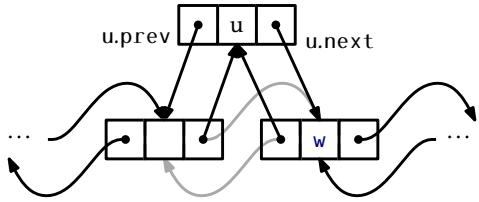
3.2.1 Dodajanje in odstranjevanje

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`, nato nastavimo `u.prev.next` in `u.next.prev`. (Glej Figure 3.3.) Zahvaljujoč dummy vozlišču, ni treba skrbeti za `w.prev` ali da `w.next` ne obstaja.

DLList

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

Operacija seznama `add(i,x` je trivialna za implementacio. Najdemo `i`-to vozlišče v `DLList` in vstavimo novo vozlišče `u`, ki vsebuje `x` tik pred njim.



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

DLList

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

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

Removing a node w from a DLList is easy. We only need to adjust pointers at w.next and w.prev so that they skip over w. Again, the use of the dummy node eliminates the need to consider any special cases:

Odstranjevanje vozlišča w iz DLList je enostavno. Potrebujemo samo postaviti kazalce na w.next in w.prev tako da preskočijo w. Uporaba dummy 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 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 `t`-tega vozlišča z uporabo `getNode`, `remove(i)` se izvede v času $O(1+\min\{i, n-i\})$.

3.2.2 Povzetek

Naslednji izrek povzema uspešnost `DLList`:

Theorem 3.2. *A `DLList` implements the `List` interface. In this implementation, the* `get(i)`, `set(i,x)`, `add(i,x)` *and* `remove(i)` *operations run in* $O(1+\min\{i, n-i\})$ *time per operation.*

Treba je omeniti, da če odmislimo ceno operacie `getNode(i)` se vse operacije v `DLList` izvedejo v konstantem času. Edini dragi del operacije v `DLList` je iskanje ustreznega vozlišča. Enkrat ko imamo ustrezno vozlišče, 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 od Chapter 2; v teh implementacijah, lahko ustrezen element najdemo v konstantnem času. Vendar, dodajanje ali odstranjevanje zahteva premikanje elementov v polju, ki na splošno ne potrebuje konsantnega časa.

Iz tega razloga so povezani seznami zelo primerni za uporabo kjer reference na vozlišča seznama dobimo od zunaj. Na primer kazalci na vozlišča povezanega seznama bi lahko bili shranjeni v `USet`. Za odstranitev element `x` iz povezanega seznama, lahko vozlišče, ki vsebuje `x`, hitro najdemo z uporabo `Uset` in vozlišče lahko odstranimo s seznama v konstantnem času.

Poglavje 4

Preskočni seznami

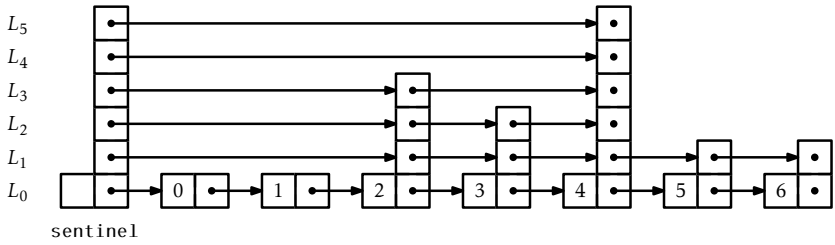
V tem poglavju bomo govorili o lepi podatkovni strukturi: preskočnem seznamu, ki ima veliko možnosti uporabe. Z uporabo preskočnega seznama lahko implementiramo `List`, ki ima $O(\log n)$ časovno implementacijo `get(i)`, `set(i,x)`, `add(i,x)`, and `remove(i)`. Prav tako lahko implementiramo `SSet`,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

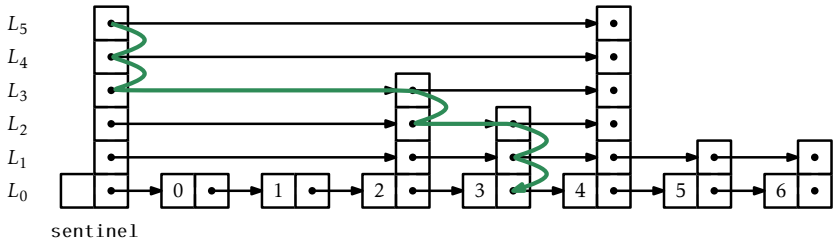
Na koncu vsakega seznama je posebno vozlišče, imanovano *stražar* od stražarja v L_h do vsakega vozlišča v L_0. Narediti pot iskanja za posamezno vozlišče uje preprosto (glej Figure 4.2) : Začnemo v zgornjem levem kotu preskočnega seznama (stražar je v L_h) in se premikamo desno toliko časa, dokler ne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

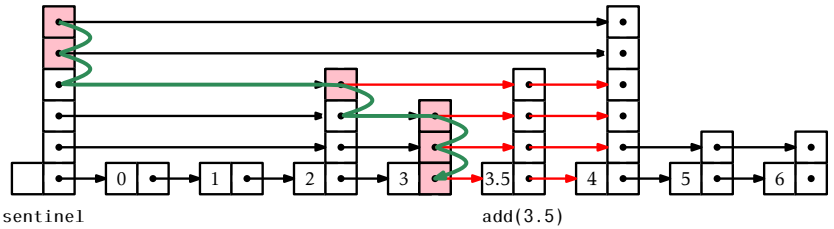
SkiplistSSet

```
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;
```

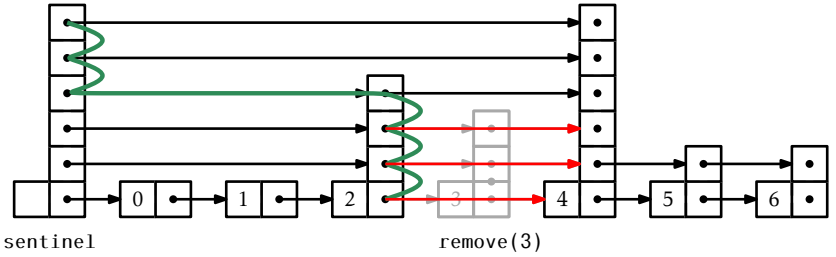


Slika 4.3: Dodajanje vozlišča 3.5 v preskočni seznam. Vozlišča shranjena v `sklad` so označena.

```
  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

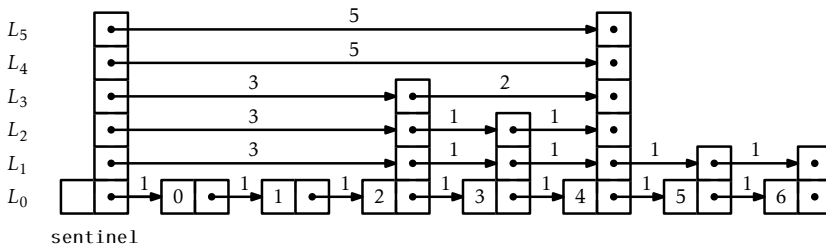
Sledeči teorem povzema uporabnost preskočnega seznama, ko ga uporabljamo pri implementaciji urejenih sklopov:

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`: An Efficient Random-Access `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 ith 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:

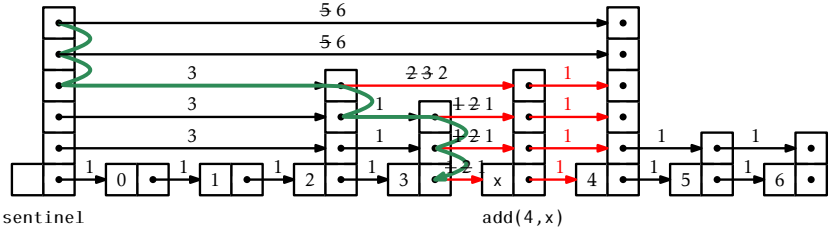
SkiplistList

```
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;
}
```

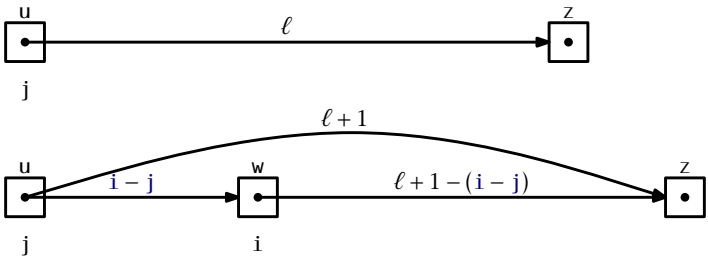
SkiplistList

```
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.

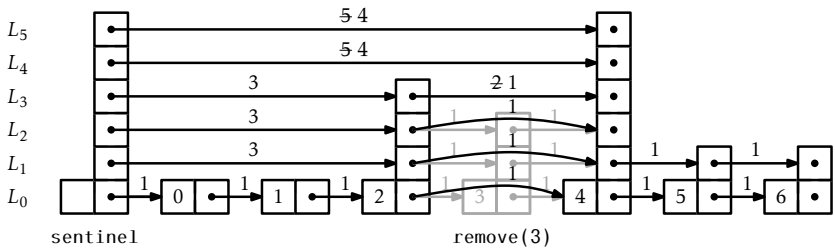
Postopek izgleda veliko bolj zakompliciran kot v resnici je. Koda je pravzaprav zelo enostavna:

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 Analysis of Skiplists

V sledečem delu bomo analizirali pričakovano višino, velikost ter dolžino Iskalne poti v preskočnem seznamu. Za razumevanje potrebujemo osnovn 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 1.2.4 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

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

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

□

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

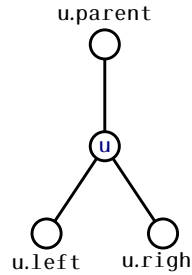
To poglavje predstavlja eno najbolj temeljnih struktur v računalništvu: dvojiška drevesa. Uporaba besede *drevo* prihaja iz dejstva, da ko jih rišemo, je risba podobna drevesom iz gozda. Obstaja veliko načinov definiranja binarnega drevesa. Matematično je *binarno drevo* povezan, neusmerjen, končni graf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

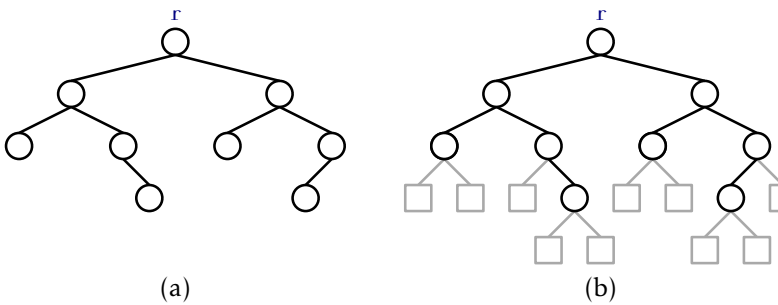
V ilustraciji, so dvojiška drevesa običajno sestavljena iz korena navzdol. Koren je na vrhu slike, ki ima levega in desnega otroka. Levi otrok je na levi strani, desni pa na desni strani (Figure 5.1). Na primer Figure 5.2. Kaže binarno drevo z devetimi vozlišči.

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`:

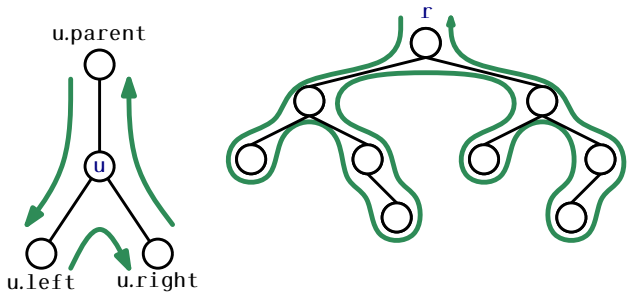
BinaryTree

```
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;
```

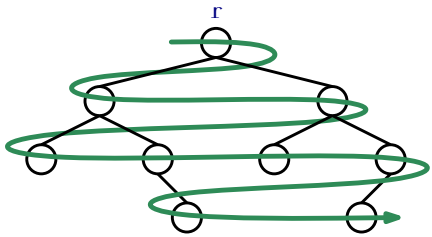


Slika 5.3: Trije primeri, ki se pojavijo na vozlišču u kadar obhodimo binarna drevesa,ki niso rekurzivna

```
      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.

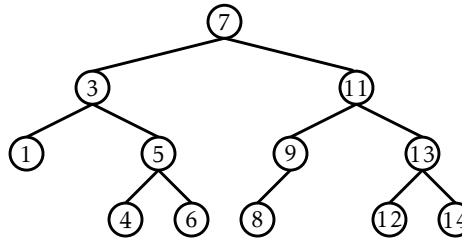
in dodamo `u.left` in `u.right` (če niso `nil`) v `q`:

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;
}
```

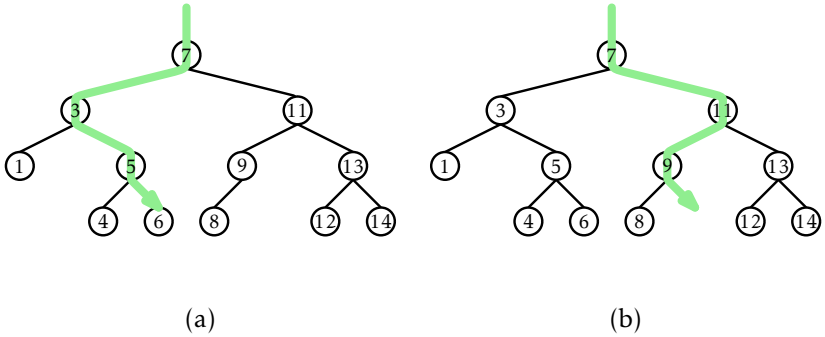
V Figure 5.6 sta prikazana dva primera iskanj v binarnem iskalnem drevesu. Drugi primer prikazuje, da tudi če ne najdemo x v drevesu, vseeno pridobimo nekaj pomembnih informacij. Če pogledamo zadnje vozlišče u pri katerem se je zgodila Možnost 1, vidimo, da je $u.x$ najmanjša vrednost v drevesu, ki je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

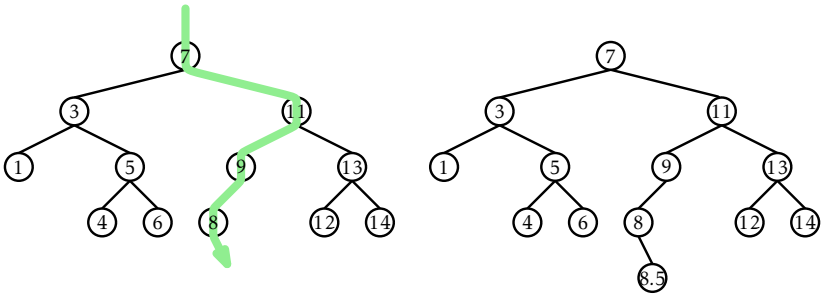
p, je odvisno od rezultata primerjave med x ter p.x.

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 x-a, ki porabi količino časa, ki je sorazmerna z višino novo vstavljenega vozlišča u. V najslabšem primeru je ta enaka višini BinarySearchTree.

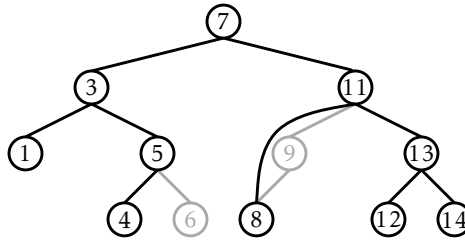
5.2.3 Brisanje

Brisanje vrednosti, ki jo hrani vozlišče `BinarySearchTree` `u`, je malce težje. Če je `u` list, potem preprosto odstranimo `u` od njegovega starša. Še bolje: če ima `u` samo enega otroka, potem lahko odstranimo `u` iz drevesa tako, da `u.parent` posvoji `u`-jevega otroka(glej Figure 5.8):

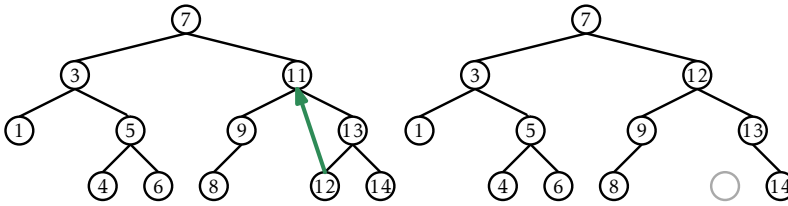
BinarySearchTree

```
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--;
}
```

Reči se zakomplicirajo, ko pa 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 iskalnega drevesa, mora biti vrednost `w.x` blizu vrednosti `u.x`. Na primer, če bi izbrali `w` tako, da je `w.x` najmanjša vrednost, ki je večja od `u.x`, bi delovalo. Iskanje primernega vozlišča `w` je preprosto; to je najmanjša vrednost, ki se nahaja v poddrevesu `u.right`. To vozlišče lahko brez skrbi odstranimo, ker nima levega otroka (glej Figure 5.9).



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.

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 `BinarySearchTree` vključuje sledenje neki poti od korena drevesa 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 (nič kaj poseben) 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 `SkiplistSSet` struktura lahko implementira `SSet` vmesnik z pričakovanim časom $O(\log n)$ na operacijo. Problem `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 neuravnoteženim binarnim iskalnim drevesom. 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.

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

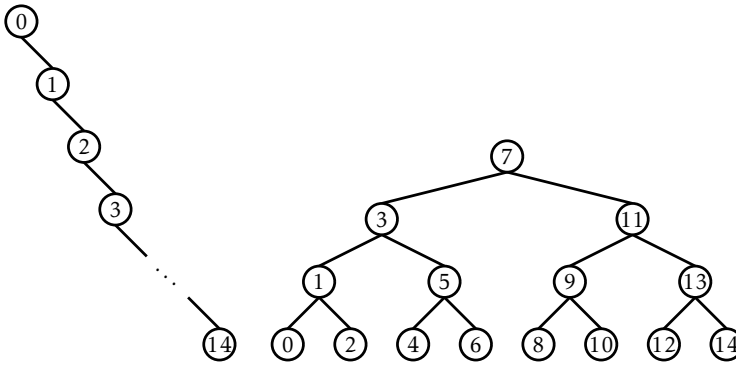
$$\langle 7, 3, 11, 1, 5, 9, 13, 0, 2, 4, 6, 8, 10, 12, 14 \rangle \ .$$

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 $0,\ldots,14$.

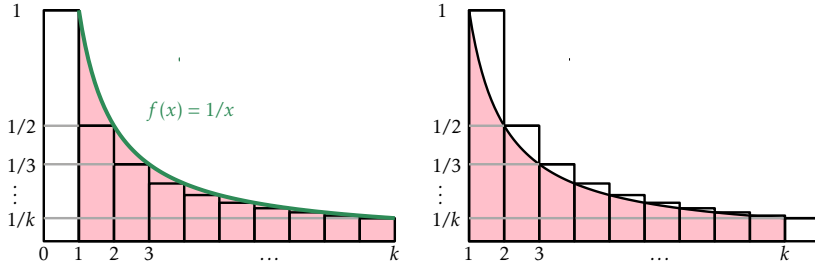
Dejstvo je, da obstaja $21,964,800$ dodatnih zaporedij, ki lahko ustvarijo drevo na desni strani in samo eno zaporedje, ki lahko ustvari drevo na levi strani.

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, upoštevajm 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

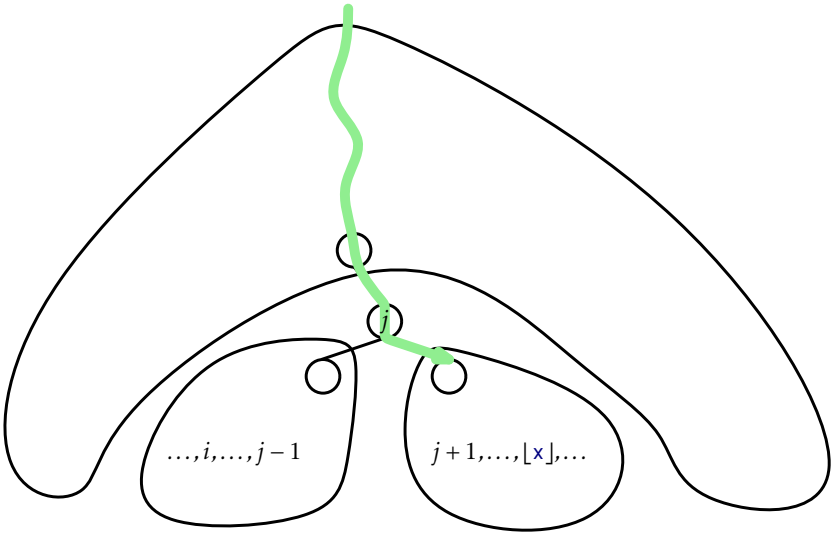
Ključna ugotovitev pri dokazovanju Lemma 6.1 je naslednja: Iskana pot za vrednost x v odprtem intervalu $(-1, n)$ v naključnem binarnem iskalnem drevesu, T, vsebuje vozlišče s ključem $i < x$ če, in samo če je naključna permutacija uporabljena za ustvarjanje T, i preden se pojavi katerakoli od $\{i+1,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

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.

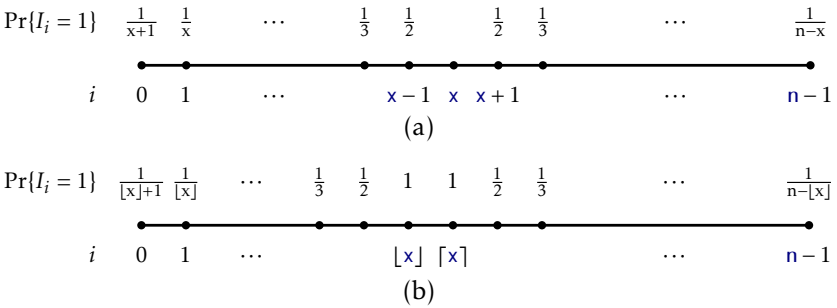
S tem opazovanjem, dokaz za Lemma 6.1 vključuje nekaj preprostih izračunov z harmonskimi števili:

Dokaz Lemma 6.1. Naj I_i bo pokazatelj naključne spremenljivke, ki je enak 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, $\texttt{find}(x)$ 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*:

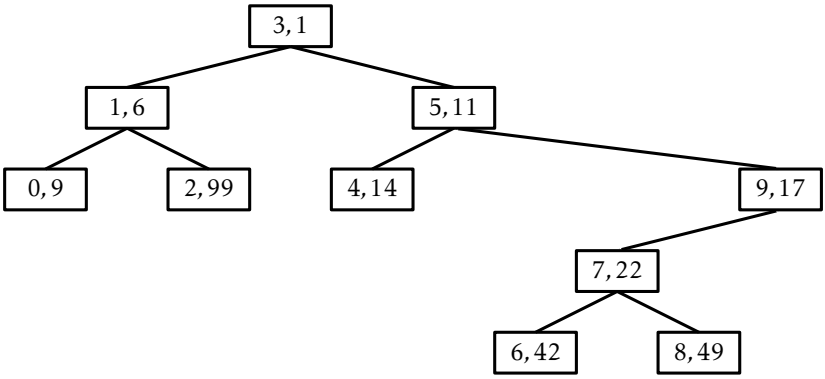
- (Lastnosti kopice) Pri vsakem vozlišču `u`, razen pri korenu, `u.parent.p` < `u.p`.

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.

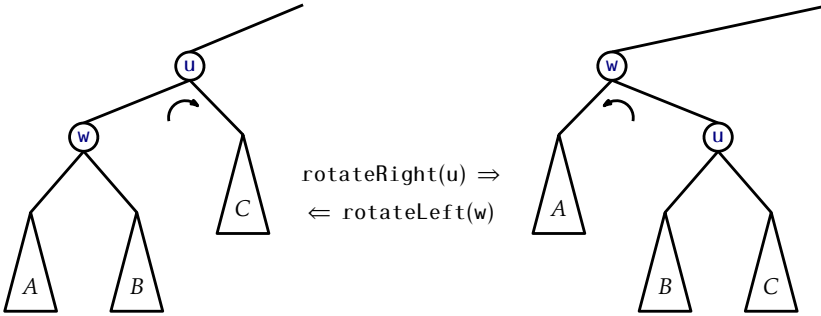
Ker so prioritete izbrane naključno, je to enako, če vzamemo naključno permutacijo ključev—v tem primeru permutacija je

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

—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,

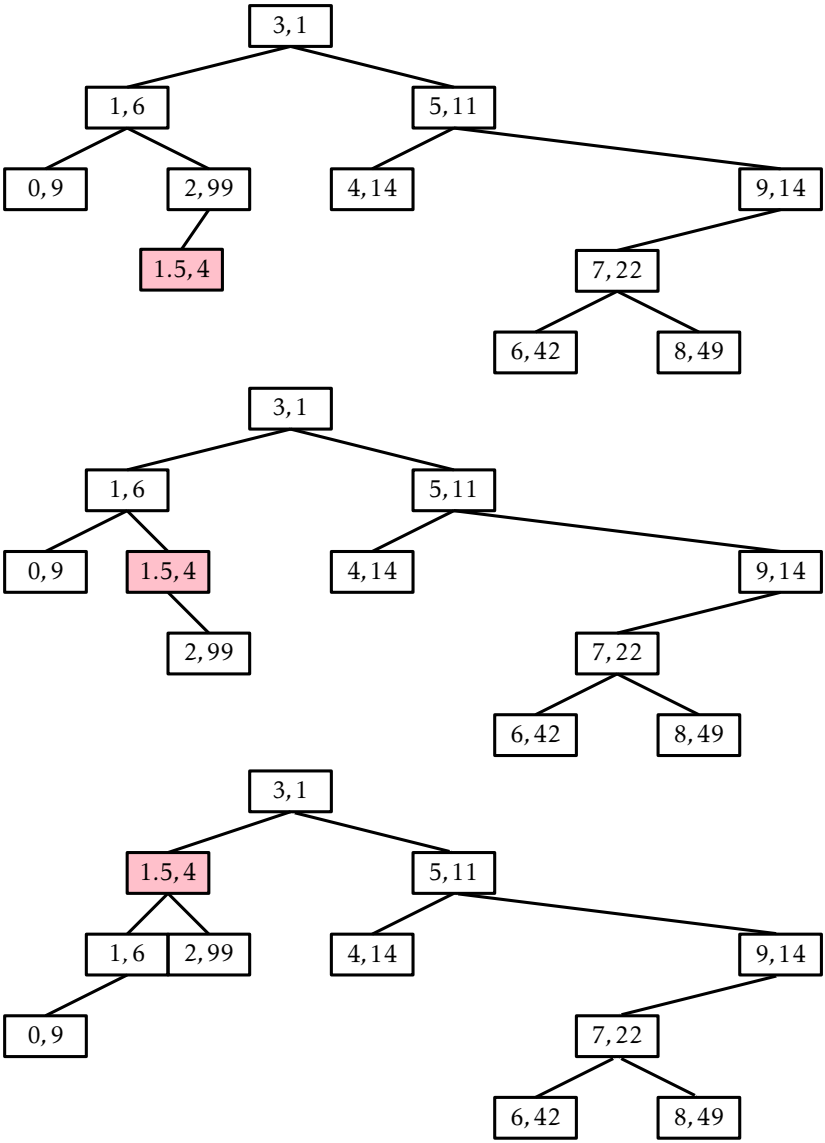
naše Treap drevo izpolnjuje lastnosti binarno iskalnega drevesa, vendar pa ni nujno, da izpolnjuje lastnosti kopice. Zlasti se lahko zgodi, da u.parent.p > u.p. Če se to zgodi, moramo izvesti rotacijo na vozlišču w=u.parent, tako da u postane starš w. Če u še naprej krši lastnosti kopice, bomo morali 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 sledi 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 $2\ lnn + O(1)$. Poleg tega, vsaka rotacija zmanjša globino u. To se ustavi, če u postane koren, tako da pričakovano število rotacij ne sme preseči predvidene dolžine iskalne poti. Zato je pričakovani čas izvajanja operacije add(x) v drevesu Treap, $O(\log n)$. (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:

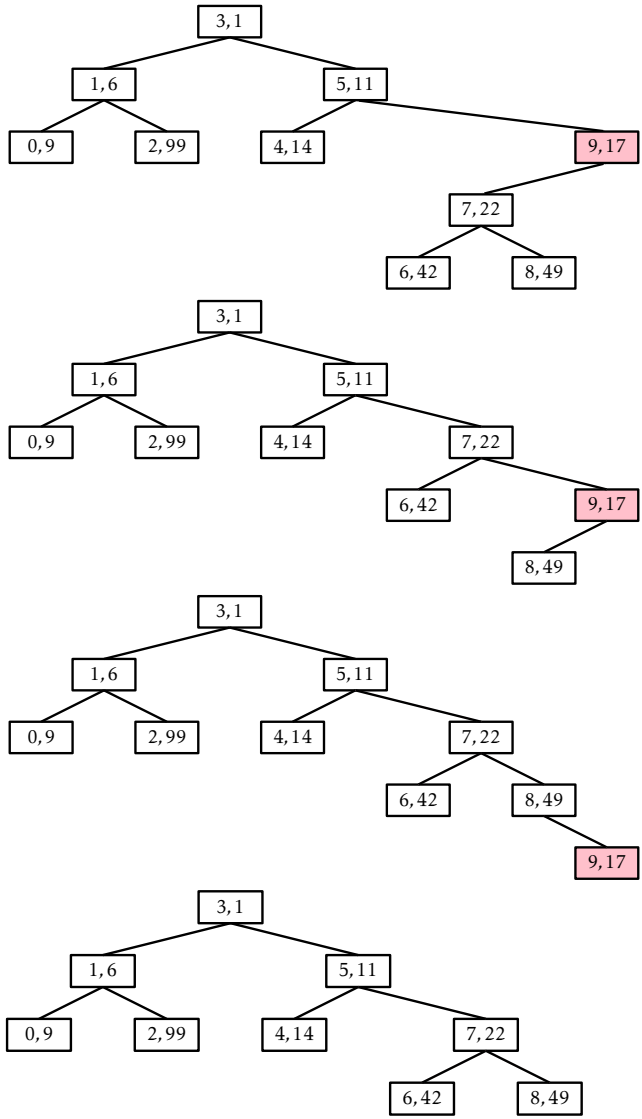
Theorem 6.2. *Treap implementira vmesnik SSet. Treap podpira operacije* add(x), remove(x) *in* find(x) *v pričakovanem času* $O(\log n)$ *za vsako operacijo.*

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

Red-Black Trees

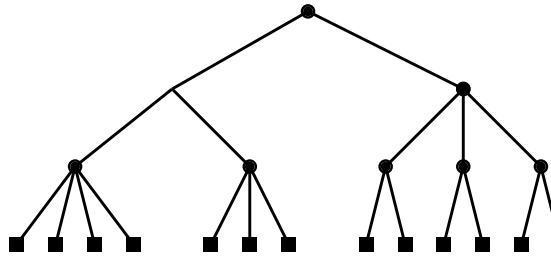
In this chapter, we present red-black trees, a version of binary search trees with logarithmic height. Red-black trees are one of the most widely used data structures. They appear as the primary search structure in many library implementations, including the Java Collections Framework and several implementations of the C++ Standard Template Library. They are also used within the Linux operating system kernel. There are several reasons for the popularity of red-black trees:

1. A red-black tree storing n values has height at most $2\log n$.
2. The `add(x)` and `remove(x)` operations on a red-black tree run in $O(\log n)$ *worst-case* time.
3. The amortized number of rotations performed during an `add(x)` or `remove(x)` operation is constant.

The first two of these properties already put red-black trees ahead of skiplists, treaps, and scapegoat trees. Skiplists and treaps rely on randomization and their $O(\log n)$ running times are only expected. Scapegoat trees have a guaranteed bound on their height, but `add(x)` and `remove(x)` only run in $O(\log n)$ amortized time. The third property is just icing on the cake. It tells us that that the time needed to add or remove an element x is dwarfed by the time it takes to find x.[1]

However, the nice properties of red-black trees come with a price: implementation complexity. Maintaining a bound of $2\log n$ on the height

[1] Note that skiplists and treaps also have this property in the expected sense. See Exercises **??** and **??**.



Slika 7.1: A 2-4 tree of height 3.

is not easy. It requires a careful analysis of a number of cases. We must ensure that the implementation does exactly the right thing in each case. One misplaced rotation or change of colour produces a bug that can be very difficult to understand and track down.

Rather than jumping directly into the implementation of red-black trees, we will first provide some background on a related data structure: 2-4 trees. This will give some insight into how red-black trees were discovered and why efficiently maintaining them is even possible.

7.1 2-4 Trees

A 2-4 tree is a rooted tree with the following properties:

Property 7.1 (height)**.** All leaves have the same depth.

Property 7.2 (degree)**.** Every internal node has 2, 3, or 4 children.

An example of a 2-4 tree is shown in Figure 7.1. The properties of 2-4 trees imply that their height is logarithmic in the number of leaves:

Lemma 7.1. *A 2-4 tree with* n *leaves has height at most* $\log n$.

Dokaz. The lower-bound of 2 on the number of children of an internal node implies that, if the height of a 2-4 tree is h, then it has at least 2^h leaves. In other words,

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

Taking logarithms on both sides of this inequality gives $h \leq \log n$. □

7.1.1 Adding a Leaf

Adding a leaf to a 2-4 tree is easy (see Figure 7.2). If we want to add a leaf u as the child of some node w on the second-last level, then we simply make u a child of w. This certainly maintains the height property, but could violate the degree property; if w had four children prior to adding u, then w now has five children. In this case, we *split* w into two nodes, w and w', having two and three children, respectively. But now w' has no parent, so we recursively make w' a child of w's parent. Again, this may cause w's parent to have too many children in which case we split it. This process goes on until we reach a node that has fewer than four children, or until we split the root, r, into two nodes r and r'. In the latter case, we make a new root that has r and r' as children. This simultaneously increases the depth of all leaves and so maintains the height property.

Since the height of the 2-4 tree is never more than $\log n$, the process of adding a leaf finishes after at most $\log n$ steps.

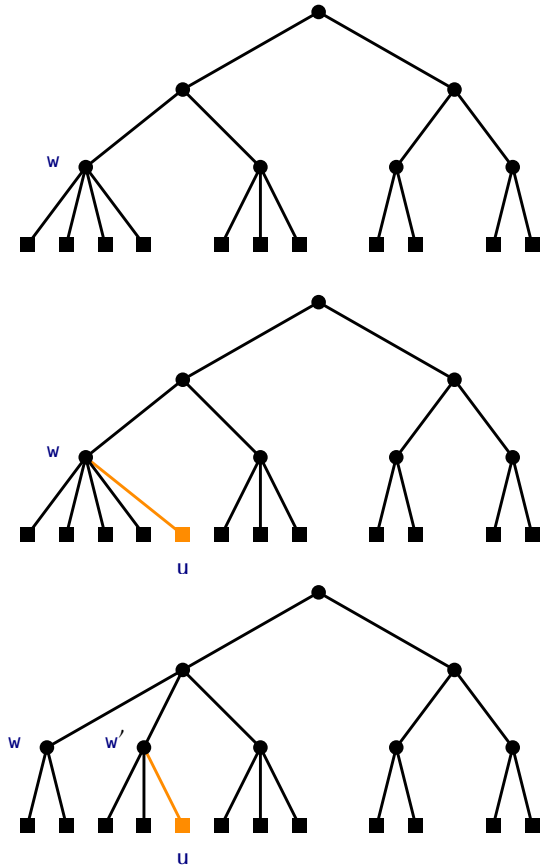
7.1.2 Removing a Leaf

Removing a leaf from a 2-4 tree is a little more tricky (see Figure 7.3). To remove a leaf u from its parent w, we just remove it. If w had only two children prior to the removal of u, then w is left with only one child and violates the degree property.

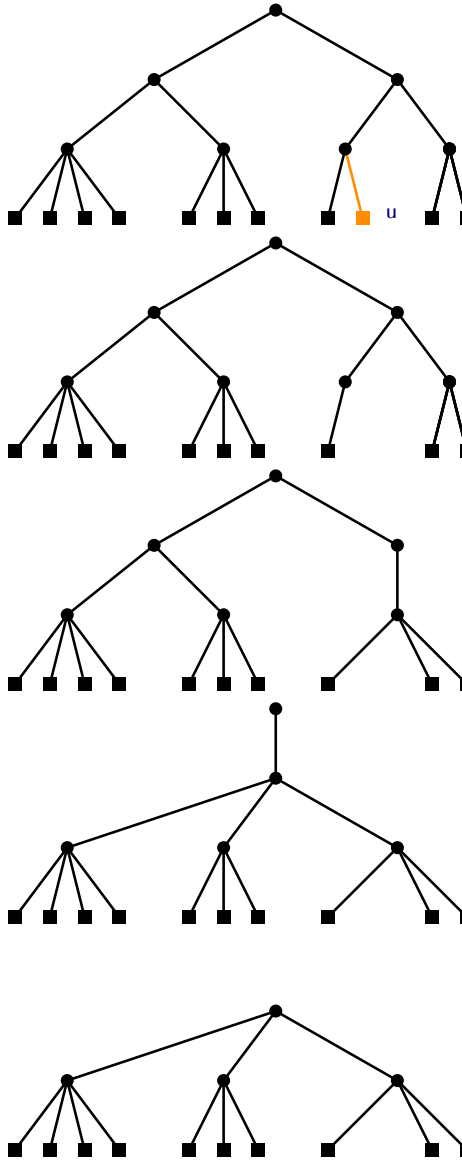
To correct this, we look at w's sibling, w'. The node w' is sure to exist since w's parent had at least two children. If w' has three or four children, then we take one of these children from w' and give it to w. Now w has two children and w' has two or three children and we are done.

On the other hand, if w' has only two children, then we *merge* w and w' into a single node, w, that has three children. Next we recursively remove w' from the parent of w'. This process ends when we reach a node, u, where u or its sibling has more than two children, or when we reach the root. In the latter case, if the root is left with only one child, then we delete the root and make its child the new root. Again, this simultaneously decreases the height of every leaf and therefore maintains the height property.

Again, since the height of the tree is never more than $\log n$, the process



Slika 7.2: Adding a leaf to a 2-4 Tree. This process stops after one split because w.parent has a degree of less than 4 before the addition.



Slika 7.3: Removing a leaf from a 2-4 Tree. This process goes all the way to the root because each of u's ancestors and their siblings have only two children.

of removing a leaf finishes after at most $\log n$ steps.

7.2 RedBlackTree: A Simulated 2-4 Tree

A red-black tree is a binary search tree in which each node, `u`, has a *colour* which is either *red* or *black*. Red is represented by the value 0 and black by the value 1.

RedBlackTree

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

Before and after any operation on a red-black tree, the following two properties are satisfied. Each property is defined both in terms of the colours red and black, and in terms of the numeric values 0 and 1.

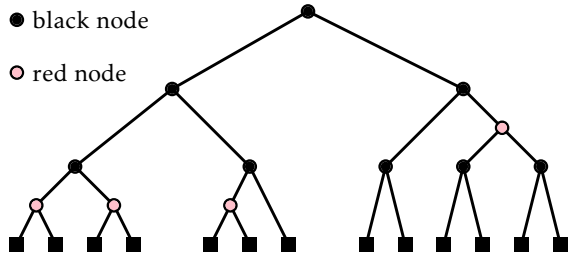
Property 7.3 (black-height)**.** There are the same number of black nodes on every root to leaf path. (The sum of the colours on any root to leaf path is the same.)

Property 7.4 (no-red-edge)**.** No two red nodes are adjacent. (For any node `u`, except the root, `u.colour` + `u.parent.colour` ≥ 1.)

Notice that we can always colour the root, `r`, of a red-black tree black without violating either of these two properties, so we will assume that the root is black, and the algorithms for updating a red-black tree will maintain this. Another trick that simplifies red-black trees is to treat the external nodes (represented by `nil`) as black nodes. This way, every real node, `u`, of a red-black tree has exactly two children, each with a well-defined colour. An example of a red-black tree is shown in Figure 7.4.

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



Slika 7.4: An example of a red-black tree with black-height 3. External (nil) nodes are drawn as squares.

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.

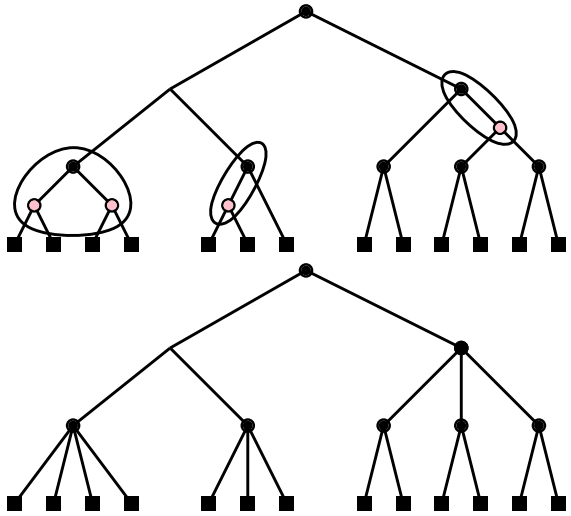
Nanašanje na Figure 7.5. Vzemimo, da ima katerokoli rdeče-črno drevo, T, n vozlišč in izvaja naslednje operacije: Zbriše vsako rdeče vozlišče n in poveže otroka vozlišča u direktno na (črnega) starša vozlišča u. Po teji spremembi imamo drevo T' z samo črnimi vozlišči.

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, 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:



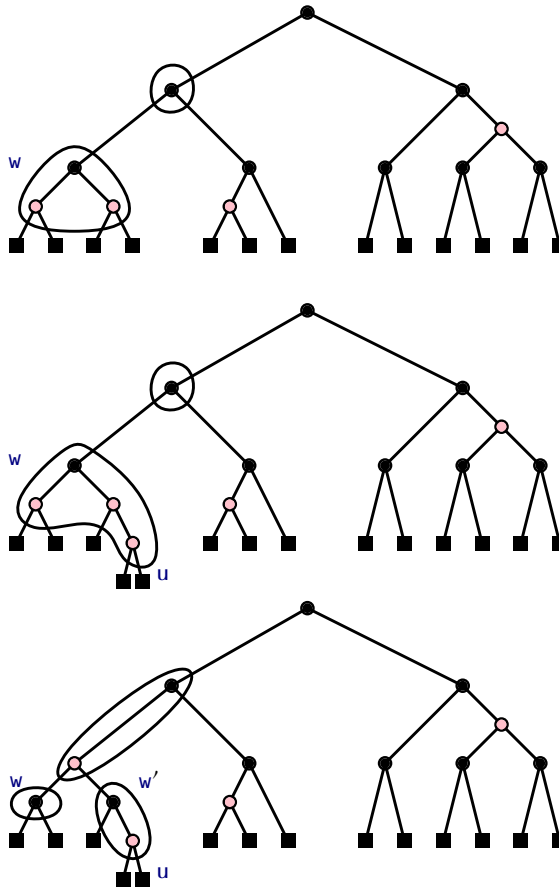
Slika 7.5: Vsako rdeče-črno drevo ima ustrezno 2-4 drevo.

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.

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.



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.)

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

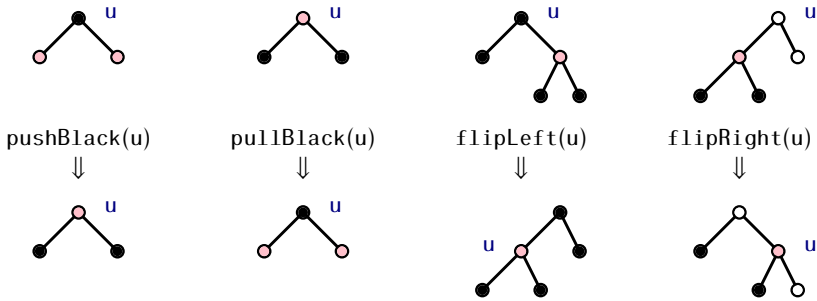
Ne obstaja nobena definicija rdeče-črnega drevesa. Namesto tega, je družin struktur, ki uspe ohranjati lastnosti black-height in no-red-edge med operacijama `add(x)` in `remove(x)`. Drugačne strukture to delajo na drugačne načine. Tukaj mi implementiramo podatkovno strukturo, ki jo kličemo `RdeceCrnoDrevo`.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



Slika 7.7: Flips, pulls and pushes

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 lastnosti 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

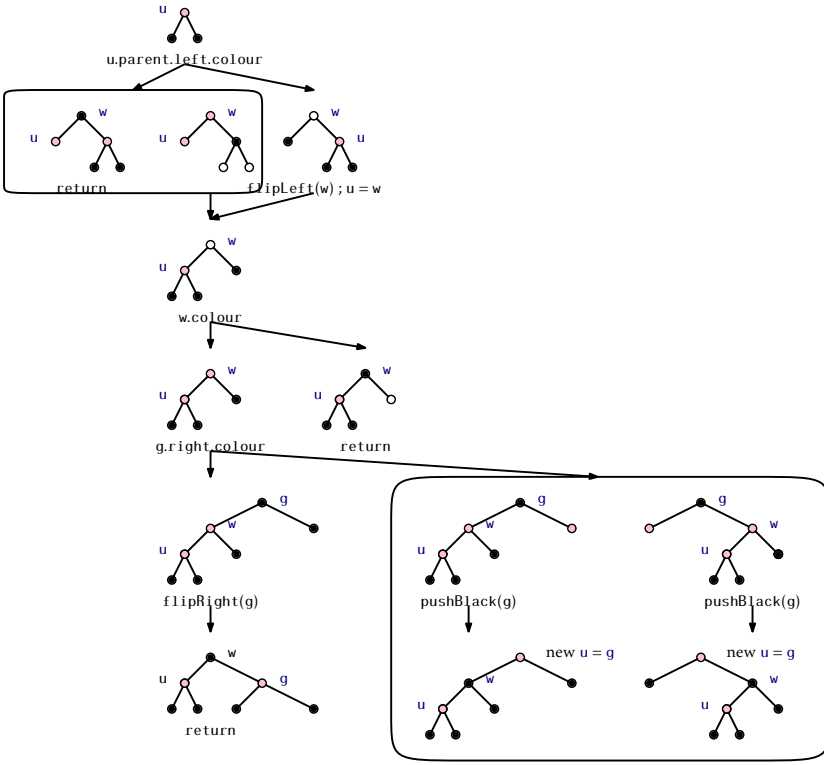
Za implementacijo add(x) v RdeceCrnoDrevo, izvedemo standardno BinarnoIskalnoDrevo vstavljanje za dodajanje novega lista, u, z u.x = x in nastavimo u.colour = red. Opomnimo, da to ne spremeni črne višine kateremukoli vozlišču, torej ne krši lastnosti black-height. To pa lahko krši levo-visečo lastnost (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 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



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

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 Removal

The `remove(x)` operation in a `RedBlackTree` is the most complicated to implement, and this is true of all known red-black tree variants. Just like the `remove(x)` operation in a `BinarySearchTree`, this operation boils down to finding a node `w` with only one child, `u`, and splicing `w` out of the tree by having `w.parent` adopt `u`.

The problem with this is that, if `w` is black, then the black-height property will now be violated at `w.parent`. We may avoid this problem, temporarily, by adding `w.colour` to `u.colour`. Of course, this introduces two other problems: (1) if `u` and `w` both started out black, then `u.colour` + `w.colour` = 2 (double black), which is an invalid colour. If `w` was red, then it is replaced by a black node `u`, which may violate the left-leaning property at `u.parent`. Both of these problems can be resolved with a call to the `removeFixup(u)` method.

RedBlackTree

```
bool remove(T x) {
  Node *u = findLast(x);
  if (u == nil || compare(u->x, x) != 0)
    return false;
  Node *w = u->right;
  if (w == nil) {
    w = u;
    u = w->left;
  } else {
    while (w->left != nil)
      w = w->left;
    u->x = w->x;
    u = w->right;
  }
  splice(w);
  u->colour += w->colour;
  u->parent = w->parent;
  delete w;
  removeFixup(u);
```

```
  return true;
}
```

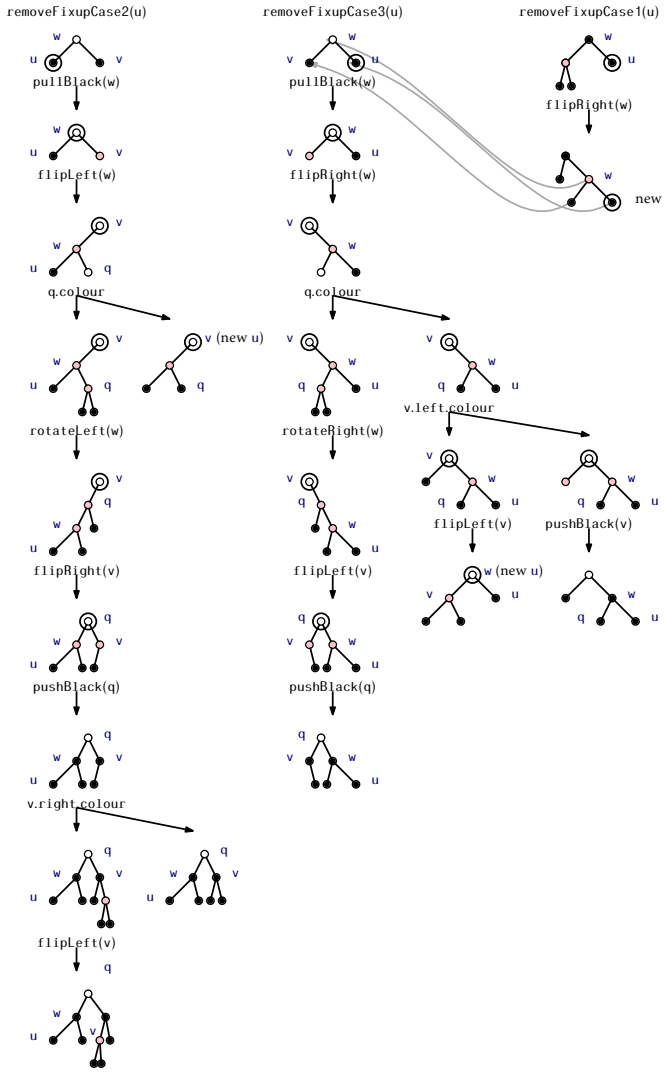
The `removeFixup(u)` method takes as its input a node `u` whose colour is black (1) or double-black (2). If `u` is double-black, then `removeFixup(u)` performs a series of rotations and recolouring operations that move the double-black node up the tree until it can be eliminated. During this process, the node `u` changes until, at the end of this process, `u` refers to the root of the subtree that has been changed. The root of this subtree may have changed colour. In particular, it may have gone from red to black, so the `removeFixup(u)` method finishes by checking if `u`'s parent violates the left-leaning property and, if so, fixing it.

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);
    }
  }
}
```

The `removeFixup(u)` method is illustrated in Figure 7.9. Again, the following text will be difficult, if not impossible, to follow without referring to Figure 7.9. Each iteration of the loop in `removeFixup(u)` processes the double-black node `u`, based on one of four cases:

Case 0: `u` is the root. This is the easiest case to treat. We recolour `u` to be black (this does not violate any of the red-black tree properties).



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

Case 1: u's sibling, v, is red. In this case, u's sibling is the 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. Note that this action causes w's parent to violate the left-leaning property and the depth of u to increase. However, it also implies that the next iteration will be in Case 3 with w coloured red. When examining Case 3 below, we will see that the process will stop during the next iteration.

RedBlackTree

```
Node* removeFixupCase1(Node *u) {
  flipRight(u->parent);
  return u;
}
```

Case 2: u's sibling, v, is black, and u is the left child of its parent, w. In this case, we call `pullBlack(w)`, making u black, v red, and darkening the colour of w to black or double-black. At this point, w does not satisfy the left-leaning property, so we call `flipLeft(w)` to fix this.

At this point, w is red and v is the root of the subtree with which we started. We need to check if w causes the no-red-edge property to be violated. We do this by inspecting w's right child, q. If q is black, then w satisfies the no-red-edge property and we can continue the next iteration with $u = v$.

Otherwise (q is red), so both the no-red-edge property and the left-leaning properties are violated at q and w, respectively. The left-leaning property is restored with a call to `rotateLeft(w)`, but the no-red-edge property is still violated. At this point, q is the left child of v, w is the left child of q, q and w are both red, and v is black or double-black. A `flipRight(v)` makes q the parent of both v and w. Following this up by a `pushBlack(q)` makes both v and w black and sets the colour of q back to the original colour of w.

At this point, the double-black node is has been eliminated and the no-red-edge and black-height properties are reestablished. Only one possible problem remains: the right child of v may be red, in which case the left-leaning property would be violated. We check this and perform a `flipLeft(v)` to correct it if necessary.

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

The following theorem summarizes the performance of the `RedBlackTree` data structure:

Theorem 7.1. *A `RedBlackTree` implements the `SSet` interface and supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in $O(\log n)$ worst-case time per operation.*

Not included in the above theorem is the following extra bonus:

Theorem 7.2. *Beginning with an empty `RedBlackTree`, any sequence of m* `add(x)` *and* `remove(x)` *operations results in a total of $O(m)$ time spent during all calls* `addFixup(u)` *and* `removeFixup(u)`.

We only sketch a proof of Theorem 7.2. By comparing `addFixup(u)` and `removeFixup(u)` with the algorithms for adding or removing a leaf in a 2-4 tree, we can convince ourselves that this property is inherited from a 2-4 tree. In particular, if we can show that the total time spent splitting, merging, and borrowing in a 2-4 tree is $O(m)$, then this implies Theorem 7.2.

The proof of this theorem for 2-4 trees uses the potential method of amortized analysis.[2] Define the potential of an internal node u in a 2-4 tree as

$$\Phi(u) = \begin{cases} 1 & \text{if } u \text{ has 2 children} \\ 0 & \text{if } u \text{ has 3 children} \\ 3 & \text{if } u \text{ has 4 children} \end{cases}$$

and the potential of a 2-4 tree as the sum of the potentials of its nodes. When a split occurs, it is because a node with four children becomes two nodes, with two and three children. This means that the overall potential drops by $3 - 1 - 0 = 2$. When a merge occurs, two nodes that used to have two children are replaced by one node with three children. The result is a drop in potential of $2 - 0 = 2$. Therefore, for every split or merge, the potential decreases by two.

Next notice that, if we ignore splitting and merging of nodes, there are only a constant number of nodes whose number of children is changed by the addition or removal of a leaf. When adding a node, one node has its number of children increase by one, increasing the potential by at most three. During the removal of a leaf, one node has its number of children decrease by one, increasing the potential by at most one, and two nodes may be involved in a borrowing operation, increasing their total potential by at most one.

To summarize, each merge and split causes the potential to drop by at least two. Ignoring merging and splitting, each addition or removal causes the potential to rise by at most three, and the potential is always non-negative. Therefore, the number of splits and merges caused by m additions or removals on an initially empty tree is at most $3m/2$. Theorem 7.2 is a consequence of this analysis and the correspondence between 2-4 trees and red-black trees.

[2] See the proofs of Lemma 2.2 and Lemma **??** for other applications of the potential method.

7.4 Discussion and Exercises

Red-black trees were first introduced by Guibas and Sedgewick [17]. Despite their high implementation complexity they are found in some of the most commonly used libraries and applications. Most algorithms and data structures textbooks discuss some variant of red-black trees.

Andersson [3] describes a left-leaning version of balanced trees that is similar to red-black trees but has the additional constraint that any node has at most one red child. This implies that these trees simulate 2-3 trees rather than 2-4 trees. They are significantly simpler, though, than the `RedBlackTree` structure presented in this chapter.

Sedgewick [27] describes two versions of left-leaning red-black trees. These use recursion along with a simulation of top-down splitting and merging in 2-4 trees. The combination of these two techniques makes for particularly short and elegant code.

A related, and older, data structure is the *AVL tree* [1]. AVL trees are *height-balanced*: At each node u, the height of the subtree rooted at `u.left` and the subtree rooted at `u.right` differ by at most one. It follows immediately that, if $F(h)$ is the minimum number of leaves in a tree of height h, then $F(h)$ obeys the Fibonacci recurrence

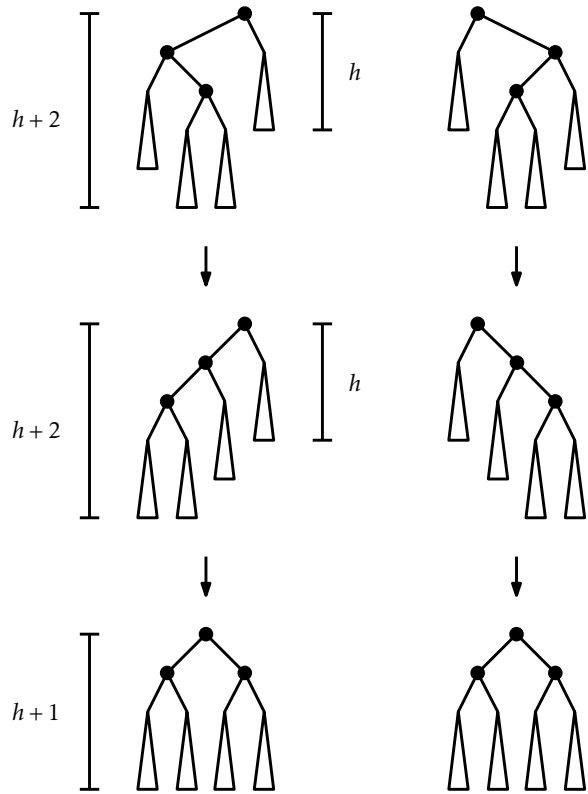
$$F(h) = F(h-1) + F(h-2)$$

with base cases $F(0) = 1$ and $F(1) = 1$. This means $F(h)$ is approximately $\varphi^h/\sqrt{5}$, where $\varphi = (1+\sqrt{5})/2 \approx 1.61803399$ is the *golden ratio*. (More precisely, $|\varphi^h/\sqrt{5} - F(h)| \leq 1/2$.) Arguing as in the proof of Lemma 7.1, this implies

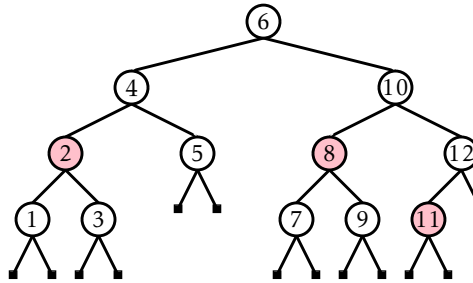
$$h \leq \log_\varphi n \approx 1.440420088 \log n \ ,$$

so AVL trees have smaller height than red-black trees. The height balancing can be maintained during `add(x)` and `remove(x)` operations by walking back up the path to the root and performing a rebalancing operation at each node `u` where the height of `u`'s left and right subtrees differ by two. See Figure 7.10.

Andersson's variant of red-black trees, Sedgewick's variant of red-black trees, and AVL trees are all simpler to implement than the `RedBlackTree` structure defined here. Unfortunately, none of them can gua-



Slika 7.10: Rebalancing in an AVL tree. At most two rotations are required to convert a node whose subtrees have a height of h and $h + 2$ into a node whose subtrees each have a height of at most $h + 1$.



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

rantee that the amortized time spent rebalancing is $O(1)$ per update. In particular, there is no analogue of Theorem 7.2 for those structures.

Exercise 7.1. Illustrate the 2-4 tree that corresponds to the `RedBlackTree` in Figure 7.11.

Exercise 7.2. Illustrate the addition of 13, then 3.5, then 3.3 on the `RedBlackTree` in Figure 7.11.

Exercise 7.3. Illustrate the removal of 11, then 9, then 5 on the `RedBlackTree` in Figure 7.11.

Exercise 7.4. Show that, for arbitrarily large values of `n`, there are red-black trees with `n` nodes that have height $2\log n - O(1)$.

Exercise 7.5. Consider the operations `pushBlack(u)` and `pullBlack(u)`. What do these operations do to the underlying 2-4 tree that is being simulated by the red-black tree?

Exercise 7.6. Show that, for arbitrarily large values of `n`, there exist sequences of `add(x)` and `remove(x)` operations that lead to red-black trees with `n` nodes that have height $2\log n - O(1)$.

Exercise 7.7. Why does the method `remove(x)` in the `RedBlackTree` implementation perform the assignment `u.parent = w.parent`? Shouldn't this already be done by the call to `splice(w)`?

Exercise 7.8. Suppose a 2-4 tree, T, has n_ℓ leaves and n_i internal nodes.

1. What is the minimum value of n_i, as a function of n_ℓ?

2. What is the maximum value of n_i, as a function of n_ℓ?

3. If T' is a red-black tree that represents T, then how many red nodes does T' have?

Exercise 7.9. Suppose you are given a binary search tree with n nodes and a height of at most $2\log n - 2$. 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. Suppose you have two red-black trees T_1 and T_2 that have the same black height, h, and such that the largest key in T_1 is smaller than the smallest key in T_2. Show how to merge T_1 and T_2 into a single red-black tree in $O(h)$ time.

Exercise 7.11. Extend your solution to Exercise 7.10 to the case where the two trees T_1 and T_2 have different black heights, $h_1 \neq h_2$. The running-time should be $O(\max\{h_1, h_2\})$.

Exercise 7.12. Prove that, during an `add(x)` operation, an AVL tree must perform at most one rebalancing operation (that involves at most two rotations; see Figure 7.10). Give an example of an AVL tree and a `remove(x)` operation on that tree that requires on the order of $\log n$ rebalancing operations.

Exercise 7.13. Implement an `AVLTree` class that implements AVL trees as described above. Compare its performance to that of the `RedBlackTree` implementation. Which implementation has a faster `find(x)` operation?

Exercise 7.14. Design and implement a series of experiments that compare the relative performance of `find(x)`, `add(x)`, and `remove(x)` for the `SSet` implemeentations `SkiplistSSet`, `ScapegoatTree`, `Treap`, and `RedBlackTree`. Be sure to include multiple test scenarios, including cases where the data is random, already sorted, is removed in random order, is removed in sorted order, and so on.

Poglavje 8

Kopice

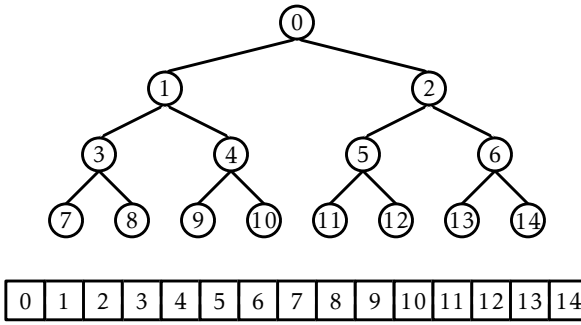
V tem poglavju si bomo pogledali 2 implementacije zelo uporabne podatkovne strukture `Polje` s prednostjo. Obe od teh dveh struktur sta posebne oblike Binarnega drevesa imenovani *Kopica*, kar pomeni "neorganizirana kopica". To je v nasprotju z binarnimi iskalnimi drevesi pri katerih pomislimo na zelo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

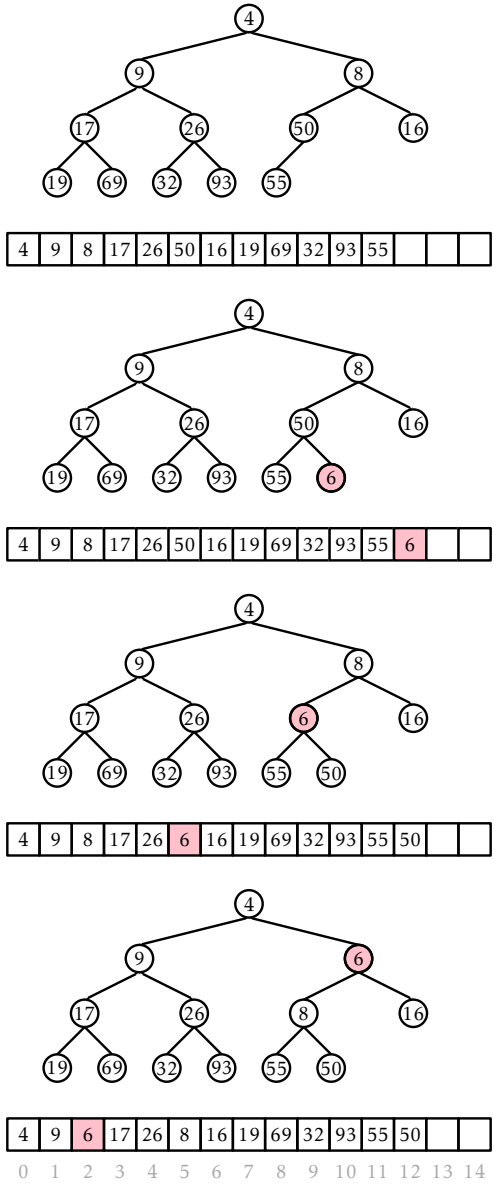
```
bool add(T x) {
  if (n + 1 > a.length) resize();
  a[n++] = x;
  bubbleUp(n-1);
  return true;
}
void bubbleUp(int i) {
  int p = parent(i);
  while (i > 0 && compare(a[i], a[p]) < 0) {
    a.swap(i,p);
    i = p;
    p = parent(i);
  }
}
```

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:

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

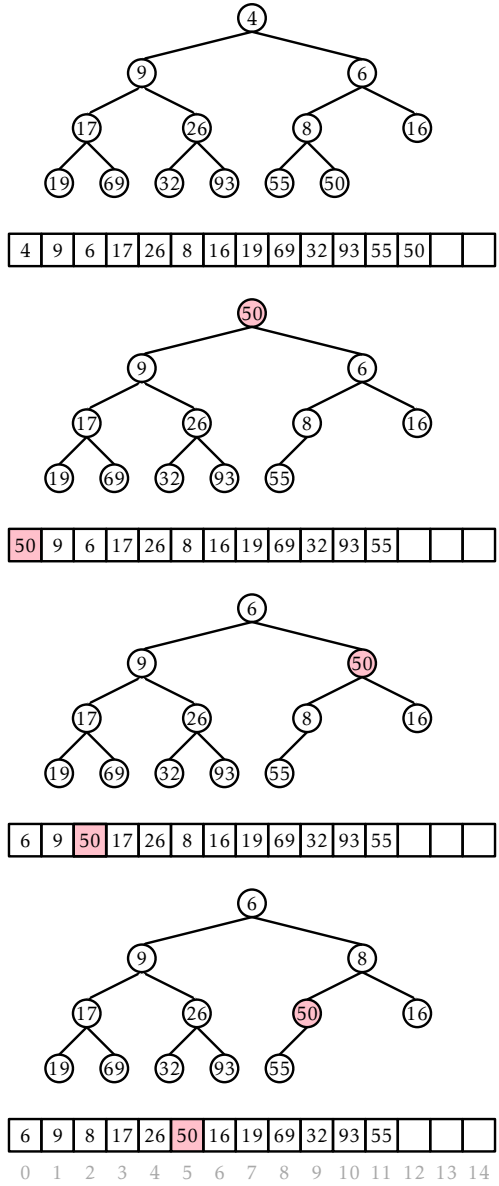
Algoritem da na obeh straneh enacbe

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

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 povecaresize(), 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

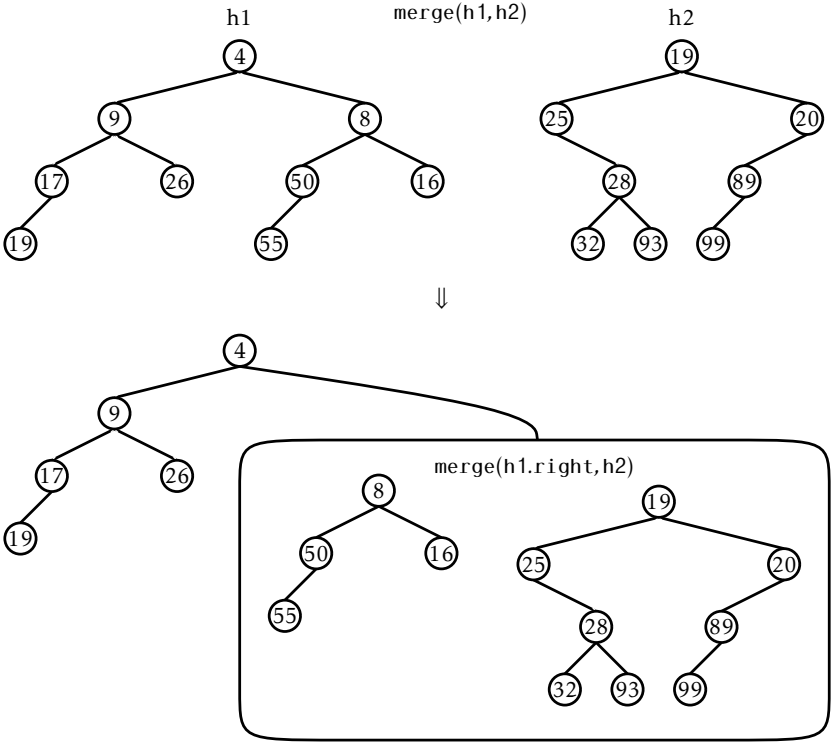
In this section, we describe the MeldableHeap, a priority Queue implementation in which the underlying structure is also a heap-ordered binary tree. However, unlike a BinaryHeap in which the underlying binary tree is completely defined by the number of elements, there are no restrictions on the shape 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 [10]. He used this representation in books containing pedigree family trees of noble families. The BinaryHeap data structure described here was first introduced by Williams [32].

The randomized MeldableHeap data structure described here appears to have first been proposed by Gambin and Malinowski [14]. Other meldable heap implementations exist, including leftist heaps [7, 22, Section 5.3.2], binomial heaps [30], Fibonacci heaps [12], pairing heaps [11], and skew heaps [28], 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 [9]. This more efficient decreaseKey(u, y) operation has applications in speeding up several graph algorithms, including Dijkstra's shortest path algorithm [12].

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

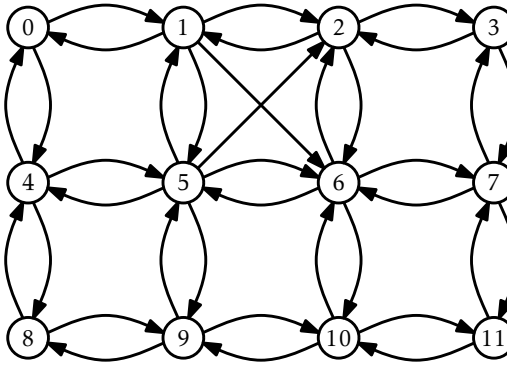
Graphs

In this chapter, we study two representations of graphs and basic algorithms that use these representations.

Mathematically, a *(directed) graph* is a pair $G = (V, E)$ where V is a set of *vertices* and E is a set of ordered pairs of vertices called *edges*. An edge (`i`, `j`) is *directed* from `i` to `j`; `i` is called the *source* of the edge and `j` is called the *target*. A *path* in G is a sequence of vertices $v_0, \ldots, v_k$ such that, for every $i \in \{1, \ldots, k\}$, the edge (v_{i-1}, v_i) is in E. A path $v_0, \ldots, v_k$ is a *cycle* if, additionally, the edge (v_k, v_0) is in E. A path (or cycle) is *simple* if all of its vertices are unique. If there is a path from some vertex v_i to some vertex v_j then we say that v_j is *reachable* from v_i. An example of a graph is shown in Figure 9.1.

Due to their ability to model so many phenomena, graphs have an enormous number of applications. There are many obvious examples. Computer networks can be modelled as graphs, with vertices corresponding to computers and edges corresponding to (directed) communication links between those computers. City streets can be modelled as graphs, with vertices representing intersections and edges representing streets joining consecutive intersections.

Less obvious examples occur as soon as we realize that graphs can model any pairwise relationships within a set. For example, in a university setting we might have a timetable *conflict graph* whose vertices represent courses offered in the university and in which the edge (`i`, `j`) is present if and only if there is at least one student that is taking both class `i` and class `j`. Thus, an edge indicates that the exam for class `i` should not be



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.

scheduled at the same time as the exam for class `j`.

Throughout this section, we will use `n` to denote the number of vertices of G and `m` to denote the number of edges of G. That is, `n` $= |V|$ and `m` $= |E|$. Furthermore, we will assume that $V = \{0, \ldots, \texttt{n} - 1\}$. Any other data that we would like to associate with the elements of V can be stored in an array of length `n`.

Some typical operations performed on graphs are:

- `addEdge(i, j)`: Add the edge `(i, j)` to E.
- `removeEdge(i, j)`: Remove the edge `(i, j)` from E.
- `hasEdge(i, j)`: Check if the edge `(i, j)` $\in E$
- `outEdges(i)`: Return a `List` of all integers `j` such that `(i, j)` $\in E$
- `inEdges(i)`: Return a `List` of all integers `j` such that `(j, i)` $\in E$

Note that these operations are not terribly difficult to implement efficiently. For example, the first three operations can be implemented directly by using a `USet`, so they can be implemented in constant expected time using the hash tables discussed in Chapter **??**. The last two operations can be implemented in constant time by storing, for each vertex, a list of its adjacent vertices.

However, different applications of graphs have different performance requirements for these operations and, ideally, we can use the simplest implementation that satisfies all the application's requirements. For this reason, we discuss two broad categories of graph representations.

9.1 AdjacencyMatrix: Representing a Graph by a Matrix

An *adjacency matrix* is a way of representing an n vertex graph $G = (V, E)$ by an $n \times n$ matrix, a, whose entries are boolean values.

AdjacencyMatrix

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

The matrix entry a[i][j] is defined as

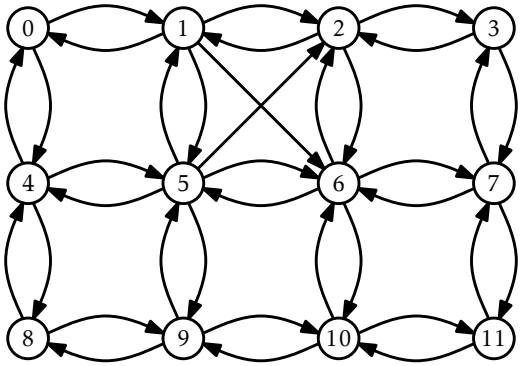
$$\texttt{a[i][j]} = \begin{cases} \texttt{true} & \text{if } (\texttt{i}, \texttt{j}) \in E \\ \texttt{false} & \text{otherwise} \end{cases}$$

The adjacency matrix for the graph in Figure 9.1 is shown in Figure 9.2. In this representation, the operations addEdge(i, j), removeEdge(i, j), and hasEdge(i, j) just involve setting or reading the matrix entry a[i][j]:

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];
}
```

These operations clearly take constant time per operation.
Where the adjacency matrix performs poorly is with the outEdges(i) and inEdges(i) operations. To implement these, we must scan all n entries in the corresponding row or column of a and gather up all the indices, j, where a[i][j], respectively a[j][i], is true.



	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.

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);
}
```

These operations clearly take $O(n)$ time per operation.
Another drawback of the adjacency matrix representation is that it is large. It stores an $n \times n$ boolean matrix, so it requires at least n^2 bits of memory. The implementation here uses a matrix of `bool` values so it actually uses on the order of n^2 bytes of memory. A more careful implementation, which packs w boolean values into each word of memory, could reduce this space usage to $O(n^2/w)$ words of memory.

Theorem 9.1. *The `AdjacencyMatrix` data structure implements the `Graph` interface. An `AdjacencyMatrix` supports the operations*

- `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 $O(n^2)$.

Despite its high memory requirements and poor performance of the `inEdges(i)` and `outEdges(i)` operations, an `AdjacencyMatrix` can still be useful for some applications. In particular, when the graph G is *dense*, i.e., it has close to n^2 edges, then a memory usage of n^2 may be acceptable.
The `AdjacencyMatrix` data structure is also commonly used because algebraic operations on the matrix `a` can be used to efficiently compute properties of the graph G. This is a topic for a course on algorithms, but we point out one such property here: If we treat the entries of `a` as integers (1 for `true` and 0 for `false`) and multiply `a` by itself using matrix multiplication then we get the matrix a^2. Recall, from the

definition of matrix multiplication, that

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

Interpreting this sum in terms of the graph G, this formula counts the number of vertices, `k`, such that G contains both edges (`i`, `k`) and (`k`, `j`). That is, it counts the number of paths from `i` to `j` (through intermediate vertices, `k`) whose length is exactly two. This observation is the foundation of an algorithm that computes the shortest paths between all pairs of vertices in G using only $O(\log n)$ matrix multiplications.

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

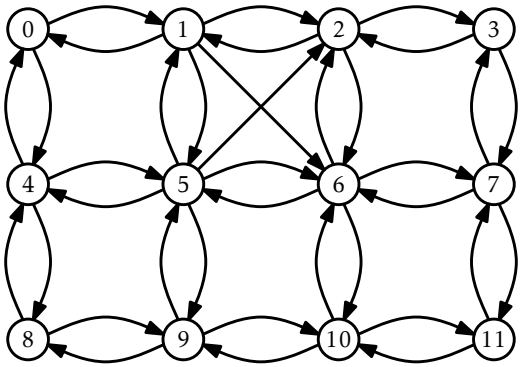
```
int n;
List *adj;
```

(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);
}
```

To se izvede v konstantem času.



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

Operacija `removeEdge(i, j)` pregleda seznam `adj[i]` dokler ne najde `j` in ga odstrani iz seznama:

AdjacencyLists
```
void removeEdge(int i, int j) {
  for (int k = 0; k < adj[i].size(); k++) {
    if (adj[i].get(k) == j) {
      adj[i].remove(k);
      return;
    }
  }
}
```

To se izvede v $O(\deg(i))$ času, kjer deg(i) (*stopnja* 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 $O(\deg(i))$ času.

Operacija `outEdges(i)` je zelo preprosta;

AdjacencyLists
```
void outEdges(int i, ListT &edges) {
  for (int k = 0; k < adj[i].size(); k++)
    edges.add(adj[i].get(k));
}
```

To se očitno izvede v $O(\deg(i))$ času.

Operacija `inEdges(i)` 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

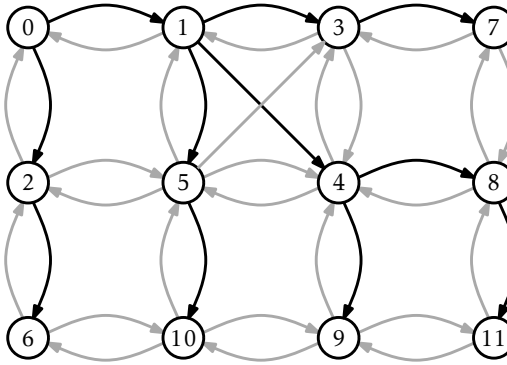
In this section we present two algorithms for exploring a graph, starting at one of its vertices, `i`, and finding all vertices that are reachable from `i`. Both of these algorithms are best suited to graphs represented using an adjacency list representation. Therefore, when analyzing these algorithms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

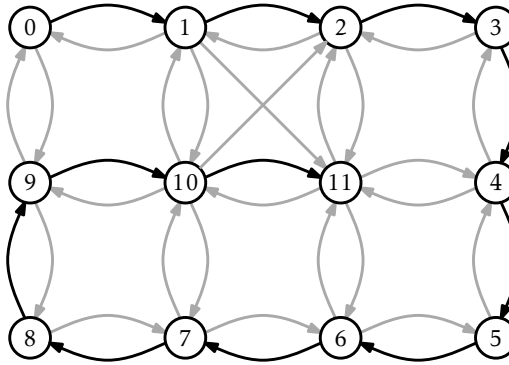
A breadth-first traversal has some very special properties. Calling `bfs(g,r)` will eventually enqueue (and eventually dequeue) every vertex `j` such that there is a directed path from `r` to `j`. Moreover, the vertices at distance 0 from `r` (`r` itself) will enter `q` before the vertices at distance 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;
```

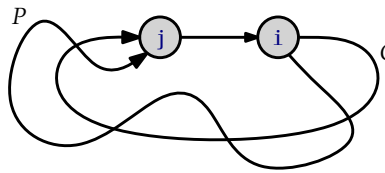
An example of the execution of this algorithm is shown in Figure 9.5. Although depth-first-search may best be thought of as a recursive algorithm, recursion is not the best way to implement it. Indeed, the code given above will fail for many large graphs by causing a stack overflow. An alternative implementation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

```
 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.

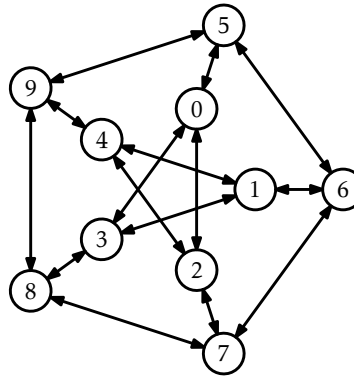
can be proven by contradiction, by considering any path P from `i` to `j`.) One application of this property is the detection of cycles. Refer to Figure 9.6. Consider some cycle, C, that can be reached from `r`. Let `i` be the first node of C that is coloured `grey`, and let `j` be the node that precedes `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 [25] and Lee [23] in the contexts of maze exploration and circuit routing, respectively.

Adjacency-list representations of graphs were presented by Hopcroft and Tarjan [18] 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 [19].

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

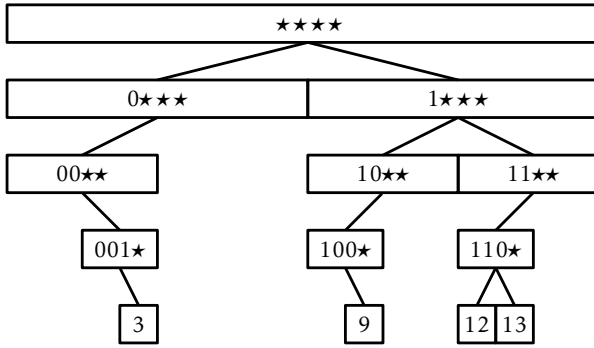
V tem poglavju se bomo vrnili k problemu implementiranja SSet-a. Razlika v implementaciji je ta, da zdaj privzamemo, da so elementi shranjeni v SSet-u, w-bitna cela števila. To pomeni da hočemo implementirati metode add(x), remove(x) in find(x), kjer velja da $x \in \{0, \ldots, 2^w - 1\}$. Če malo pomislimo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



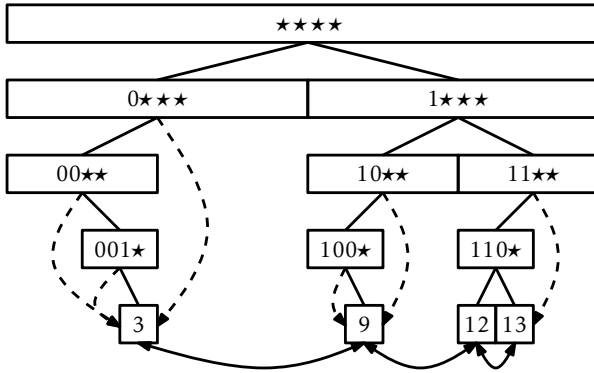
Slika 10.1: Cela števila shranjena v binary trie so zakodirana kot poti od korena do lista.

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).

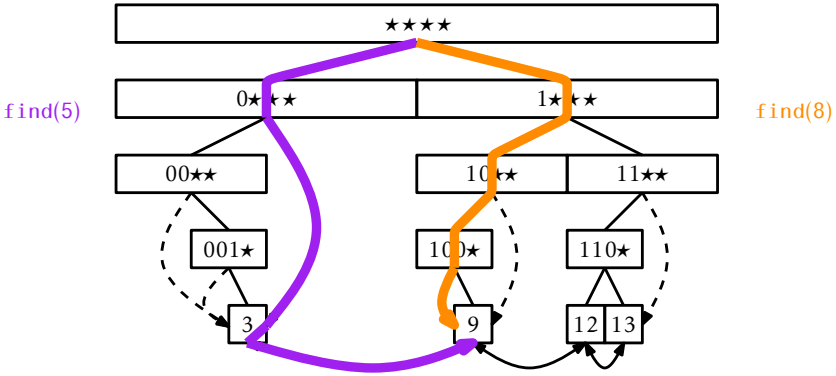
Ker iskalna pot za vrednost `x` odvisi od bitov `x`-a, nam bo koristilo, če otroka vozlišča poimenujemo `u`, `u.child`[0] (`left`) in `u.child`[1] (`right`). Tile kazalci na otroke bodo pravzaprav služili dvema namenoma. Ker listi v binary trie nimajo nobenega otroka, so kazalci uporabljeni za povezavo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

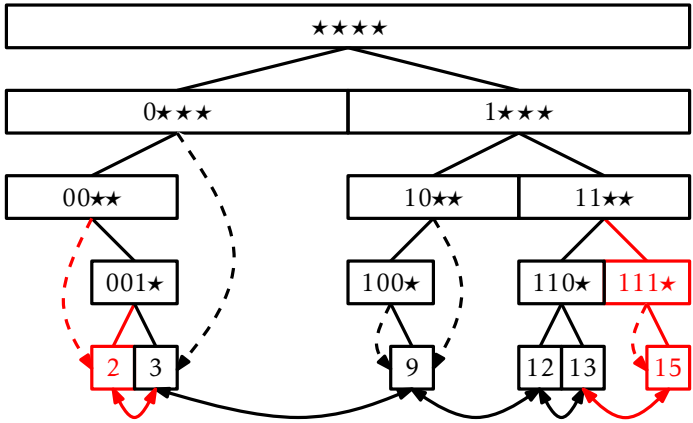
BinaryTrie

```
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.

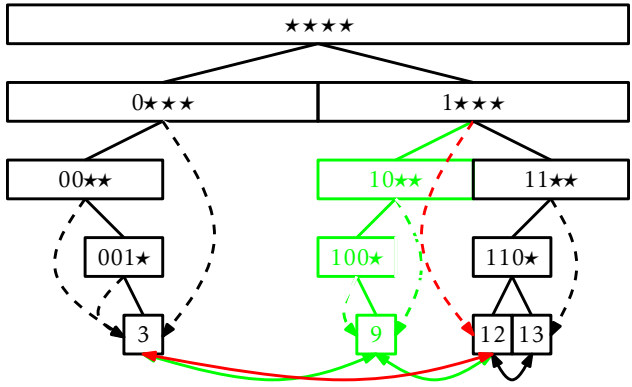
BinaryTrie

```
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;
}
```

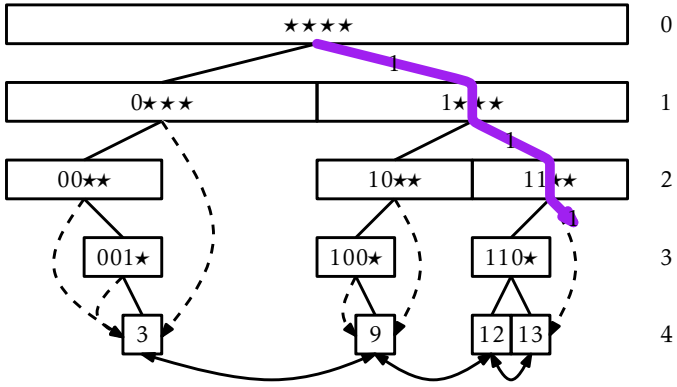
Theorem 10.1. *`BinaryTrie` implementira `SSet` vmesnik za `w`-bitna cela števila. `BinaryTrie` podpira operacije* `add(x)`, `remove(x)` *in* `find(x)` *v časovni kompleksnosti $O(w)$ na operacijo. Prostor, ki ga `BinaryTrie` uporablja za shranjevanje `n` vrednosti je $O(n \cdot w)$.*

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]`,…, `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 $t[0],\ldots,t[w]$. 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. *`AXFastTrie` 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: A Doubly-Logarithmic Time SSet

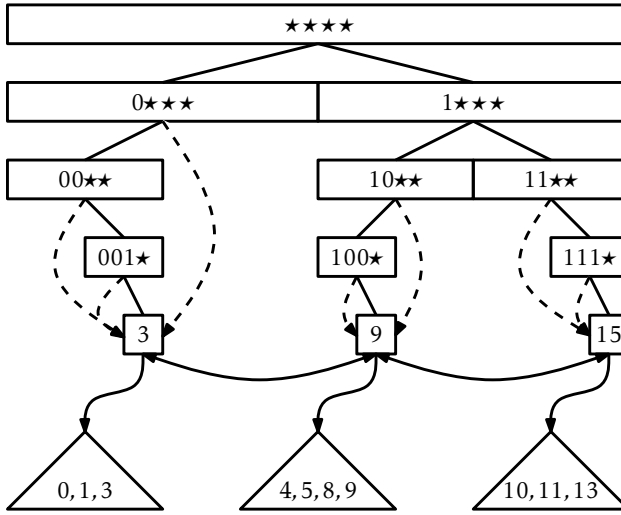
The `XFastTrie` is a vast—even exponential—improvement over the `BinaryTrie` in terms of query time, but the `add(x)` and `remove(x)` operations are still not terribly fast. Furthermore, the space usage, $O(n \cdot w)$, is higher than the other `SSet` implementations described in this book, which all use $O(n)$ space. These two problems are related; if `n` `add(x)` operations build a structure of size $n \cdot w$, then the `add(x)` operation requires at least on the order of `w` time (and space) per operation.

The `YFastTrie`, discussed next, simultaneously improves the space and speed of `XFastTries`. A `YFastTrie` uses an `XFastTrie`, `xft`, but only stores $O(n/w)$ values in `xft`. In this way, the total space used by `xft` is only $O(n)$. Furthermore, only one out of every `w` `add(x)` or `remove(x)` operations in the `YFastTrie` results in an `add(x)` or `remove(x)` operation in `xft`. By doing this, the average cost incurred by calls to `xft`'s `add(x)` and `remove(x)` operations is only constant.

The obvious question becomes: If `xft` only stores n/w elements, where do the remaining $n(1 - 1/w)$ elements go? These elements move into *secondary structures*, in this case an extended version of treaps (Section 6.2). There are roughly n/w of these secondary structures so, on average, each of them stores $O(w)$ items. Treaps support logarithmic time `SSet` operations, so the operations on these treaps will run in $O(\log w)$ time, as required.

More concretely, a `YFastTrie` contains an `XFastTrie`, `xft`, that contains a random sample of the data, where each element appears in the sample independently with probability $1/w$. For convenience, the value $2^w - 1$, is always contained in `xft`. Let $x_0 < x_1 < \cdots < x_{k-1}$ denote the elements stored in `xft`. Associated with each element, x_i, is a treap, t_i, that stores all values in the range $x_{i-1} + 1, \ldots, x_i$. This is illustrated in Figure 10.7.

The `find(x)` operation in a `YFastTrie` is fairly easy. We search for `x` in `xft` and find some value x_i associated with the treap t_i. We then use the



Slika 10.7: A YFastTrie containing the values 0, 1, 3, 4, 6, 8, 9, 10, 11, and 13.

treap find(x) method on t_i to answer the query. The entire method is a one-liner:

YFastTrie

```
T find(T x) {
  return xft.find(YPair<T>(intValue(x))).t->find(x);
}
```

The first find(x) operation (on xft) takes $O(\log w)$ time. The second find(x) operation (on a treap) takes $O(\log r)$ time, where r is the size of the treap. Later in this section, we will show that the expected size of the treap is $O(w)$ so that this operation takes $O(\log w)$ time.[1]

Adding an element to a YFastTrie is also fairly simple—most of the time. The add(x) method calls xft.find(x) to locate the treap, t, into which x should be inserted. It then calls t.add(x) to add x to t. At this point, it tosses a biased coin that comes up as heads with probability $1/w$ and as tails with probability $1 - 1/w$. If this coin comes up heads, then x will be added to xft.

This is where things get a little more complicated. When x is added to

[1] This is an application of *Jensen's Inequality*: If $\mathrm{E}[r] = w$, then $\mathrm{E}[\log r] \le \log w$.

xft, the treap t needs to be split into two treaps, t1 and t′. The treap t1 contains all the values less than or equal to x; t′ is the original treap, t, with the elements of t1 removed. Once this is done, we add the pair (x, t1) to xft. Figure 10.8 shows an example.

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;
}
```

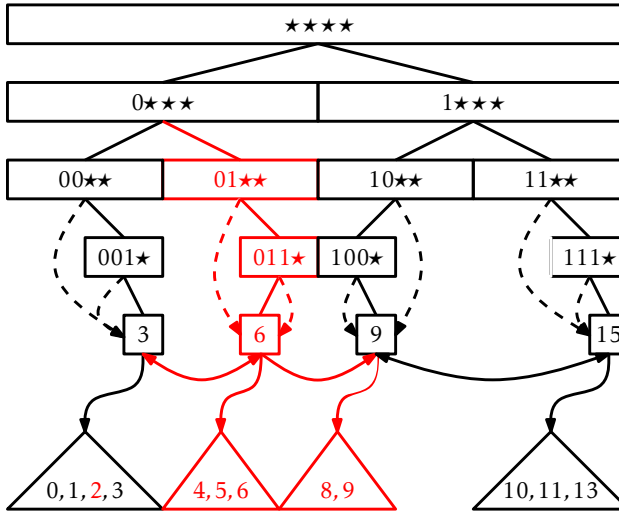
Adding x to t takes $O(\log w)$ time. Exercise **??** shows that splitting t into t1 and t′ can also be done in $O(\log w)$ expected time. Adding the pair (x,t1) to xft takes $O(w)$ time, but only happens with probability $1/w$. Therefore, the expected running time of the add(x) operation is

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

The remove(x) method undoes the work performed by add(x). We use xft to find the leaf, u, in xft that contains the answer to xft.find(x). From u, we get the treap, t, containing x and remove x from t. If x was also stored in xft (and x is not equal to $2^w - 1$) then we remove x from xft and add the elements from x's treap to the treap, t2, that is stored by u's successor in the linked list. This is illustrated in 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);
  if (ret) n--;
```

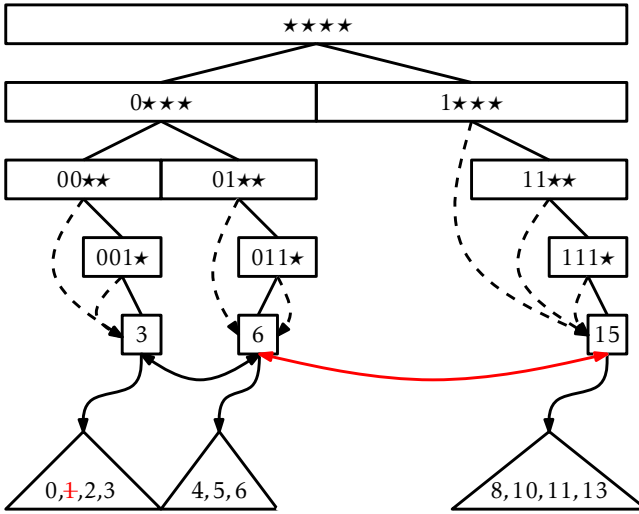


Slika 10.8: Adding the values 2 and 6 to a YFastTrie. The coin toss for 6 came up heads, so 6 was added to xft and the treap containing 4, 5, 6, 8, 9 was split.

```
  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;
}
```

Finding the node u in xft takes $O(\log w)$ expected time. Removing x from t takes $O(\log w)$ expected time. Again, Exercise **??** shows that merging all the elements of t into t2 can be done in $O(\log w)$ time. If necessary, removing x from xft takes $O(w)$ time, but x is only contained in xft with probability $1/w$. Therefore, the expected time to remove an element from a YFastTrie is $O(\log w)$.
Earlier in the discussion, we delayed arguing about the sizes of treaps in this structure until later. Before finishing this chapter, we prove the result we need.

Lemma 10.1. *Let* x *be an integer stored in a* YFastTrie *and let* n_x *denote*



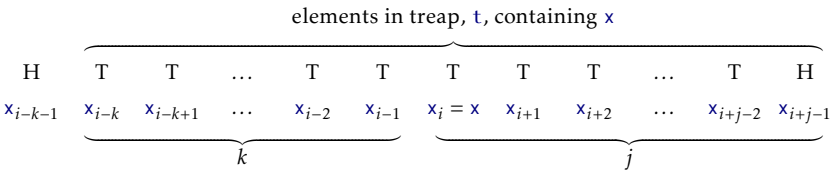
Slika 10.9: Removing the values 1 and 9 from a `YFastTrie` in Figure 10.8.

the number of elements in the treap, `t`*, that contains* `x`*. Then* $\mathrm{E}[\mathtt{n_x}] \le 2\mathtt{w} - 1$.

Dokaz. Refer to Figure 10.10. Let $\mathtt{x}_1 < \mathtt{x}_2 < \cdots < \mathtt{x}_i = \mathtt{x} < \mathtt{x}_{i+1} < \cdots < \mathtt{x}_n$ denote the elements stored in the `YFastTrie`. The treap `t` contains some elements greater than or equal to `x`. These are $\mathtt{x}_i, \mathtt{x}_{i+1}, \ldots, \mathtt{x}_{i+j-1}$, where $\mathtt{x}_{i+j-1}$ is the only one of these elements in which the biased coin toss performed in the `add(x)` method turned up as heads. In other words, $\mathrm{E}[j]$ is equal to the expected number of biased coin tosses required to obtain the first heads.[2] Each coin toss is independent and turns up as heads with probability $1/\mathtt{w}$, so $\mathrm{E}[j] \le \mathtt{w}$. (See Lemma **??** for an analysis of this for the case $\mathtt{w} = 2$.)
Similarly, the elements of `t` smaller than `x` are $\mathtt{x}_{i-1}, \ldots, \mathtt{x}_{i-k}$ where all these k coin tosses turn up as tails and the coin toss for $\mathtt{x}_{i-k-1}$ turns up as heads. Therefore, $\mathrm{E}[k] \le \mathtt{w} - 1$, since this is the same coin tossing experiment considered in the preceding paragraph, but one in which the last toss is not counted. In summary, $\mathtt{n_x} = j + k$, so

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

[2] This analysis ignores the fact that j never exceeds $\mathtt{n} - i + 1$. However, this only decreases $\mathrm{E}[j]$, so the upper bound still holds.



Slika 10.10: The number of elements in the treap `t` containing `x` is determined by two coin tossing experiments.

Lemma 10.1 was the last piece in the proof of the following theorem, which summarizes the performance of the `YFastTrie`:

Theorem 10.3. *A `YFastTrie` implements the `SSet` interface for `w`-bit integers. A `YFastTrie` supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in $O(\log w)$ expected time per operation. The space used by a `YFastTrie` that stores `n` values is $O(n+w)$.*

The `w` term in the space requirement comes from the fact that `xft` always stores the value $2^w - 1$. The implementation could be modified (at the expense of adding some extra cases to the code) so that it is unnecessary to store this value. In this case, the space requirement in the theorem becomes $O(n)$.

10.4 Discussion and Exercises

The first data structure to provide $O(\log w)$ time `add(x)`, `remove(x)`, and `find(x)` operations was proposed by van Emde Boas and has since become known as the *van Emde Boas* (or *stratified*) *tree* [29]. The original van Emde Boas structure had size 2^w, making it impractical for large integers.

The `XFastTrie` and `YFastTrie` data structures were discovered by Willard [31]. The `XFastTrie` structure is closely related to van Emde Boas trees; for instance, the hash tables in an `XFastTrie` replace arrays in a van Emde Boas tree. That is, instead of storing the hash table `t[i]`, a van Emde Boas tree stores an array of length 2^i.

Another structure for storing integers is Fredman and Willard's fusion

trees [13]. This structure can store n w-bit integers in $O(n)$ space so that the `find(x)` operation runs in $O((\log n)/(\log w))$ time. By using a fusion tree when $\log w > \sqrt{\log n}$ and a `YFastTrie` when $\log w \le \sqrt{\log n}$, one obtains an $O(n)$ space data structure that can implement the `find(x)` operation in $O(\sqrt{\log n})$ time. Recent lower-bound results of Pătraşcu and Thorup [26] show that these results are more or less optimal, at least for structures that use only $O(n)$ space.

Exercise 10.1. Design and implement a simplified version of a `BinaryTrie` that does not have a linked list or jump pointers, but for which `find(x)`
still runs in $O(w)$ time.

Exercise 10.2. Design and implement a simplified implementation of an `XFastTrie` that doesn't use a binary trie at all. Instead, your implementation should store everything in a doubly-linked list and $w+1$ hash tables.

Exercise 10.3. We can think of a `BinaryTrie` as a structure that stores bit strings of length w in such a way that each bitstring is represented as a root to leaf path. Extend this idea into an `SSet` implementation that stores variable-length strings and implements `add(s)`, `remove(s)`, and `find(s)` in time proporitional to the length of `s`.
Hint: Each node in your data structure should store a hash table that is indexed by character values.

Exercise 10.4. For an integer $x \in \{0, \ldots 2^w - 1\}$, let $d(x)$ denote the difference between x and the value returned by `find(x)` [if `find(x)` returns `null`, then define $d(x)$ as 2^w]. For example, if `find(23)` returns 43, then $d(23) = 20$.

1. Design and implement a modified version of the `find(x)` operation in an `XFastTrie` that runs in $O(1 + \log d(x))$ expected time. Hint: The hash table $t[w]$ contains all the values, x, such that $d(x) = 0$, so that would be a good place to start.

2. Design and implement a modified version of the `find(x)` operation in an `XFastTrie` that runs in $O(1 + \log\log d(x))$ expected time.

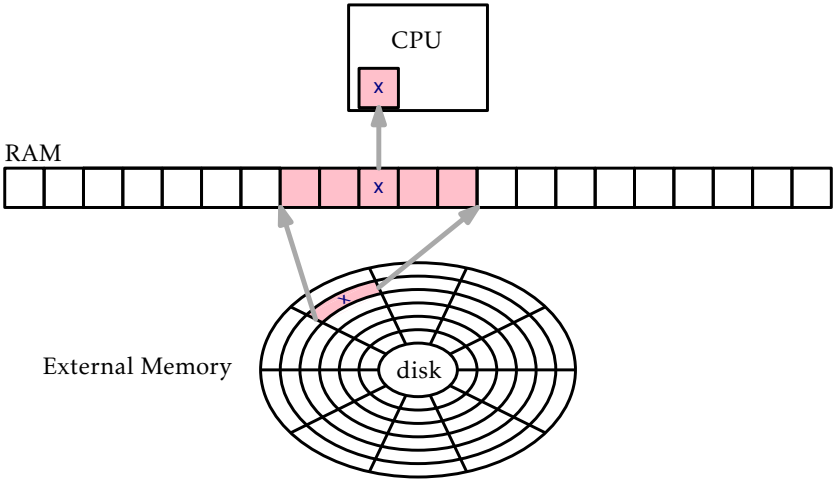
Poglavje 11

Iskanje v zunanjem pomnilniku

Skozi knjigo smo uporabljali w-bitni besedni-RAM model računanja, katerega smo opredelili v Section **??**. 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:

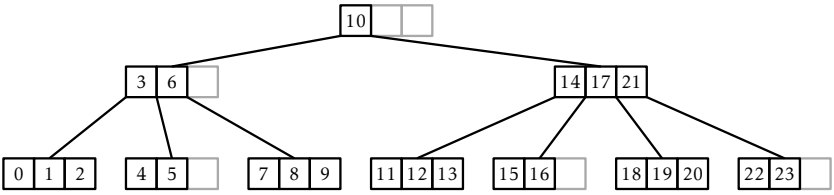
$$\begin{aligned} h &\leq \frac{\log \ell - 1}{\log B} + 1 \\ &\leq \frac{\log \ell}{\log B} + 1 \\ &= \log_B \ell + 1 \ . \end{aligned}$$

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` $+ 1]$.

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



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

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

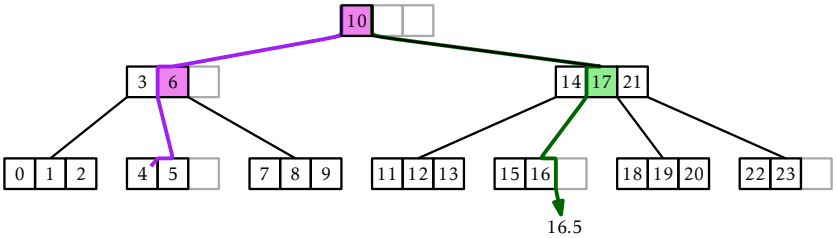
bajtov podatkov. To bi bila odlična vrednost B za trdi disk ali pogon SSD (predstavljen v uvodu tega poglavja), kateri ima velikost bloka 4096 bajtov.
`BTree` razred, kateri implementira B-drevo, vsebuje `BlockStore`, `bs`,ki vsebuje `BTree` vozlišča in prav tako indeks, `ri`, korena. Kot ponavadi, število `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



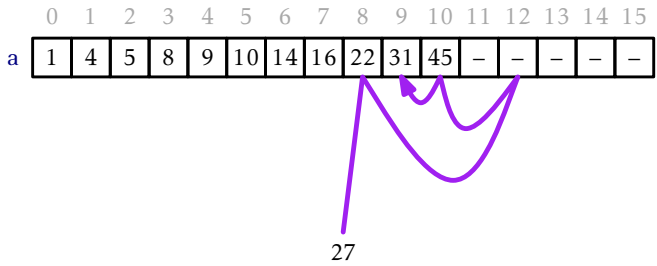
Slika 11.3: Uspešno iskanje (vrednosti 4) in neuspešno iskanje (za vrednost 16.5) v *B*-drevesu. Obarvana vozlišča predstavljajo, kje se je vrednost med iskanjem `z`ja spremenila.

`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 $a[0],\ldots,a[k-1]$ 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 `najdIt(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 `najdiIt(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 `najdIt(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

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

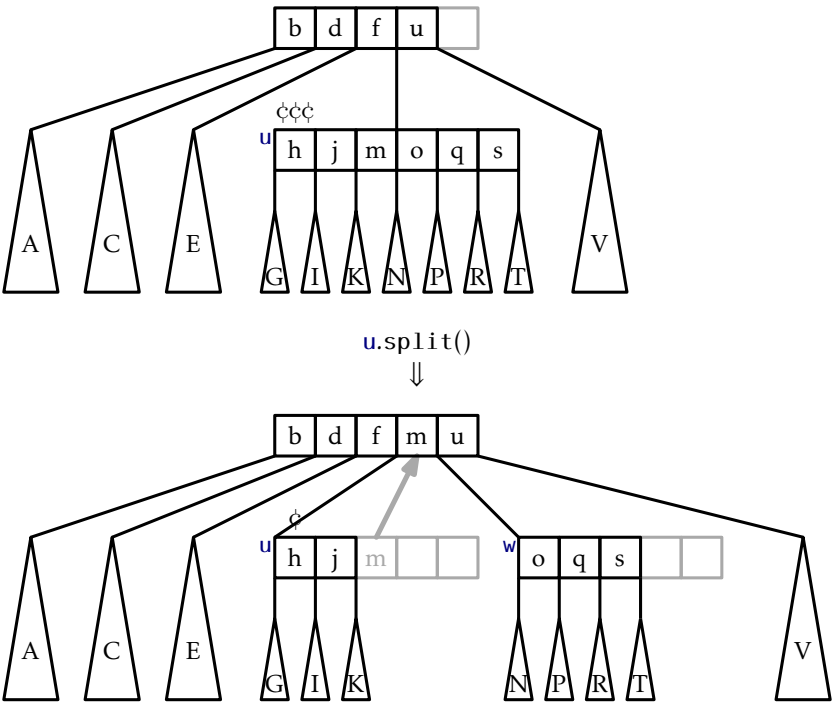
11.2.2 Addition

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.

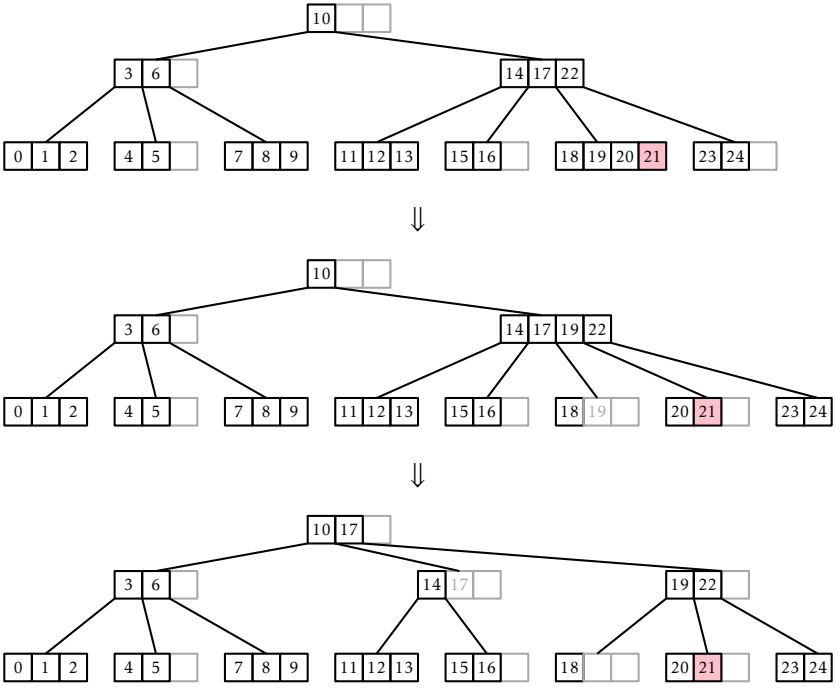
Like all balanced search trees, some form of rebalancing is required during an `add(x)` operation. In a B-tree, this is done by *splitting* nodes. Refer to Figure 12.5 for what follows. Although splitting takes place across two levels of recursion, it is best understood as an operation that takes a node `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*.

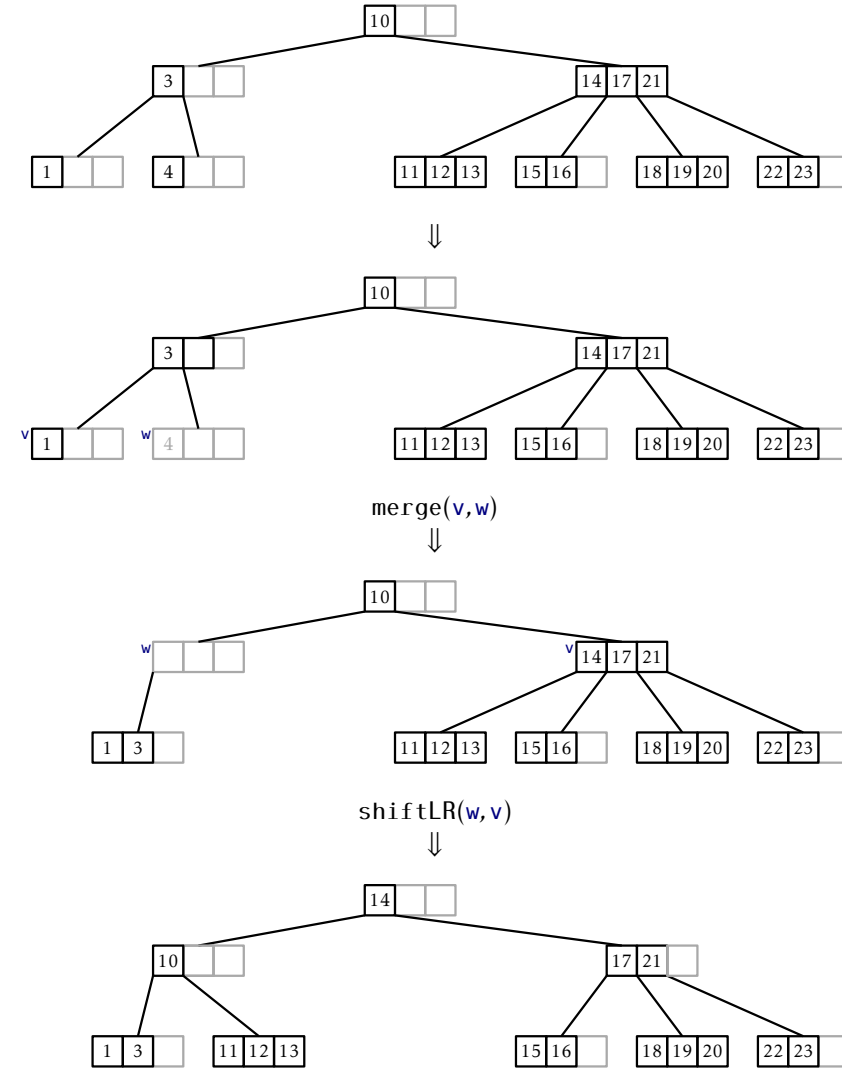
When an underflow occurs, u either borrows keys from, or is merged with, one of its siblings. If u is merged with a sibling, then u's parent will now have one less child and one less key, which can cause u's parent to underflow; this is again corrected by borrowing or merging, but merging may cause u's 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.

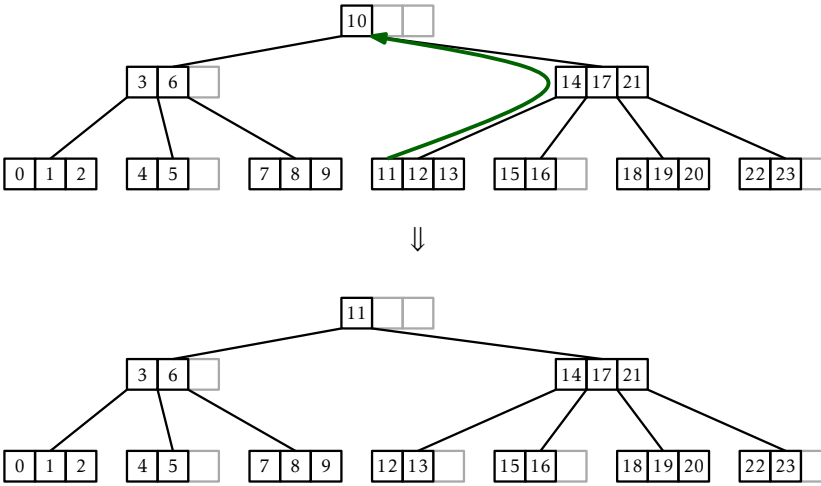
The removeRecursive(x,ui) method is a recursive implementation of the preceding algorithm:

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);
}
```

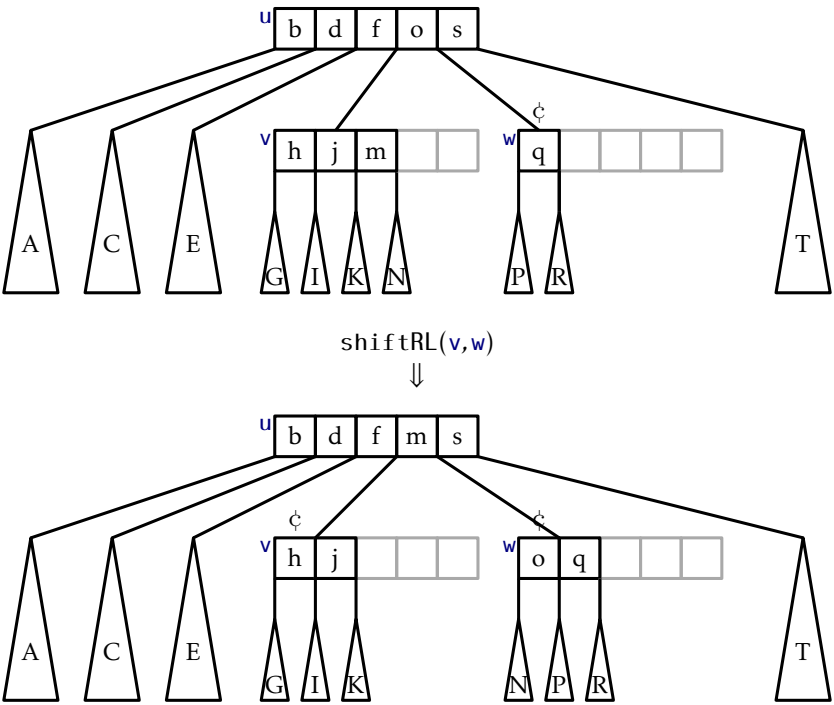
In the following, we focus on the case when $i \neq 0$ so that any underflow at the ith child of u will be corrected with the help of the $(i-1)$st child of u. The case $i=0$ is similar and the details can be found in the accompanying source code.
To fix an underflow at node w, we need to find more keys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

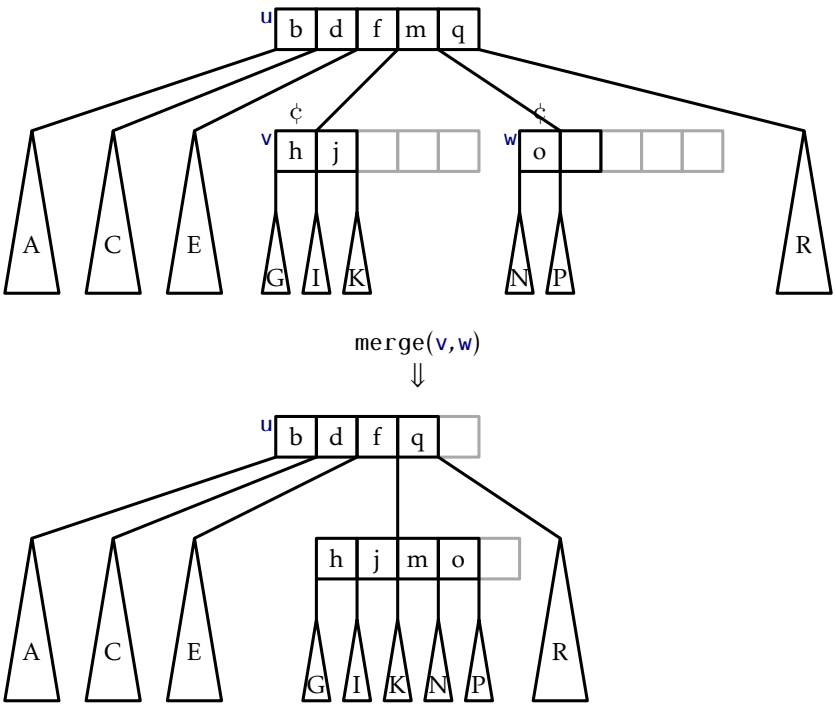
$$B-2+\mathtt{size}(\mathtt{w}) \geq 2B-2$$

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 [2]. 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 [4] in 1970 and, less than ten years later, the title of Comer's ACM Computing Surveys article referred to them as ubiquitous [6]. 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 [16] 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 [20]. 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:

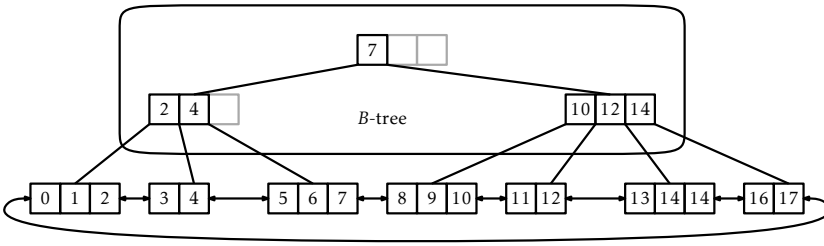
   (a) keys can be redistributed among u, v, and w; or

   (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.

=======

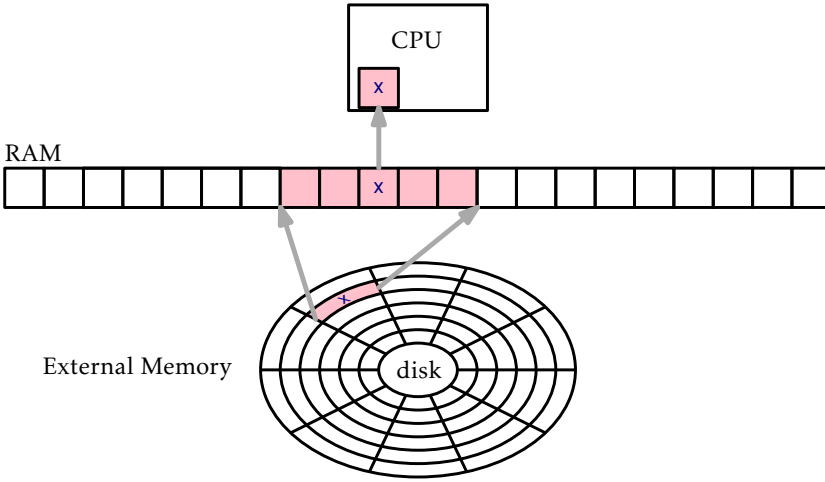
Poglavje 12

External Memory Searching

Throughout this book, we have been using the w-bit word-RAM model of computation defined in Section **??**. 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

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

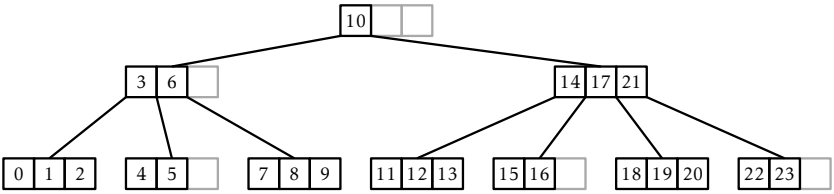
Vzamemo logaritem iz prve neenakosti in preuredimo. Dobimo:

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

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,

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

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



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]`.

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

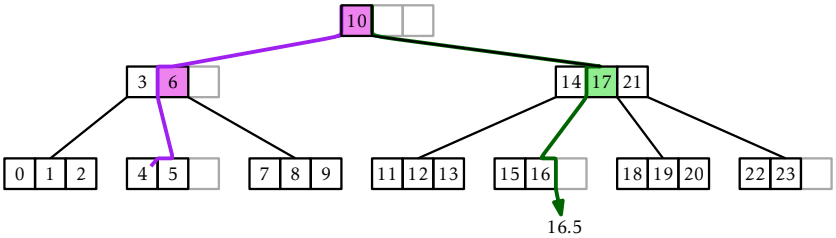
Primer B-drevesa z $B = 2$ je prikazan na sliki Figure **??**.
Upoštevajte, da so podatki shranjeni v vozliščih B-drevesa velikosti $O(B)$. Zato, je v nastavitvah zunanjega spomina, vrednost B v B-drevesu določena tako, da celotno vozlišče lahko ustreza enemu zunanje spominskemu bloku. V tem primeru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

BTree

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



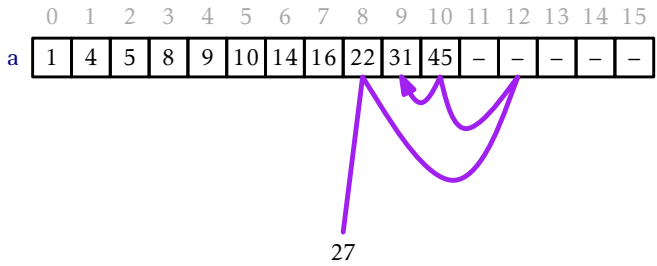
Slika 12.3: Uspešno iskanje (vrednosti 4) in neuspešno iskanje (za vrednost 16.5) v *B*-drevesu. Obarvana vozlišča predstavljajo, kje se je vrednost med iskanjem `z` spremenila.

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 $O(\log_B n)$. 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

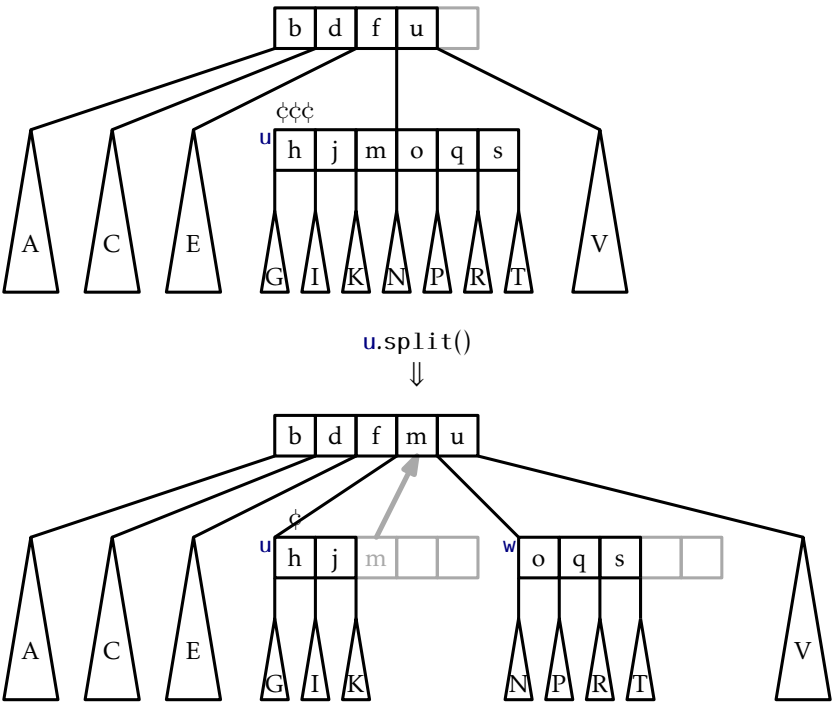
$$O(\log_B n) \times O(\log B) = O(\log n) \ .$$

12.2.2 Addition

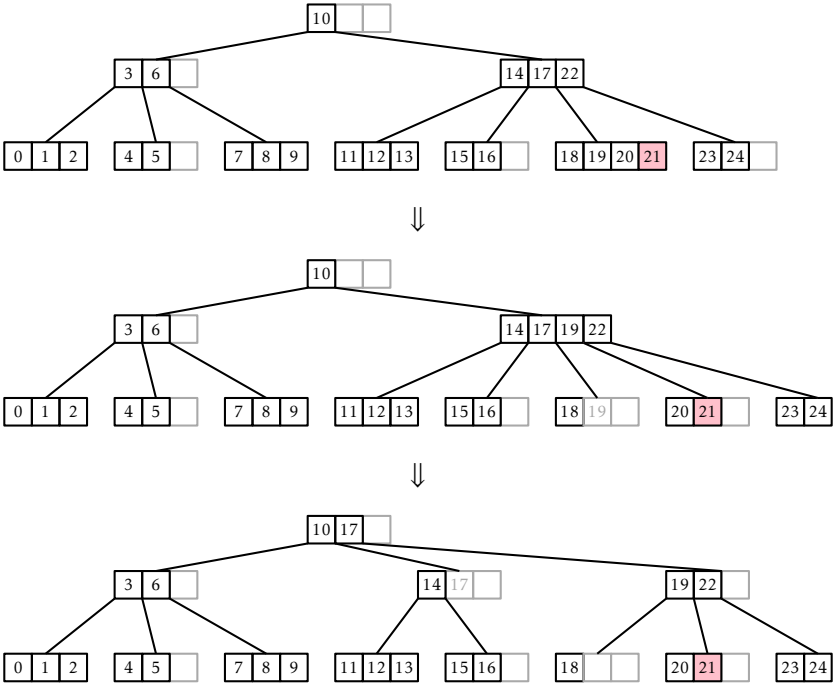
One important difference between B-trees and the `BinarySearchTree` data structure from Section 5.2 is that the nodes of a B-tree do not store pointers to their parents. The reason for this will be explained shortly. The lack of parent pointers means that the `add(x)` and `remove(x)` operations on B-trees 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*.

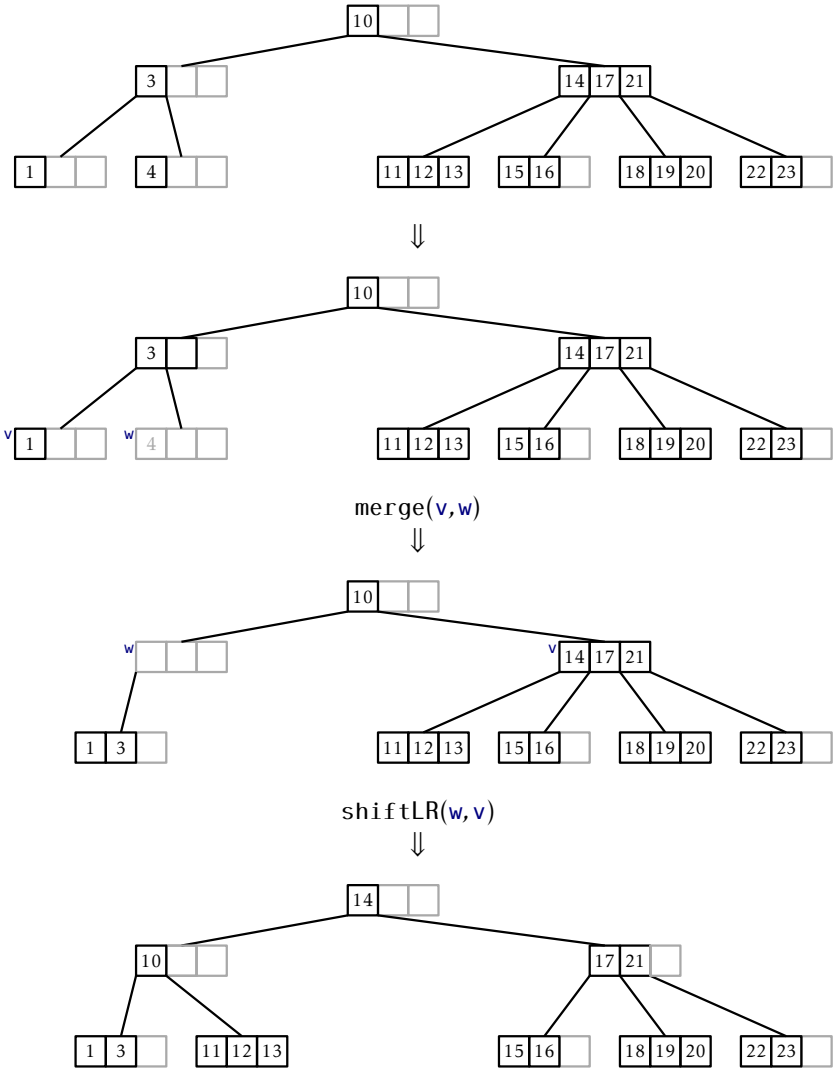
When an underflow occurs, u either borrows keys from, or is merged with, one of its siblings. If u is merged with a sibling, then u's parent will now have one less child and one less key, which can cause u's parent to underflow; this is again corrected by borrowing or merging, but merging may cause u's 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.

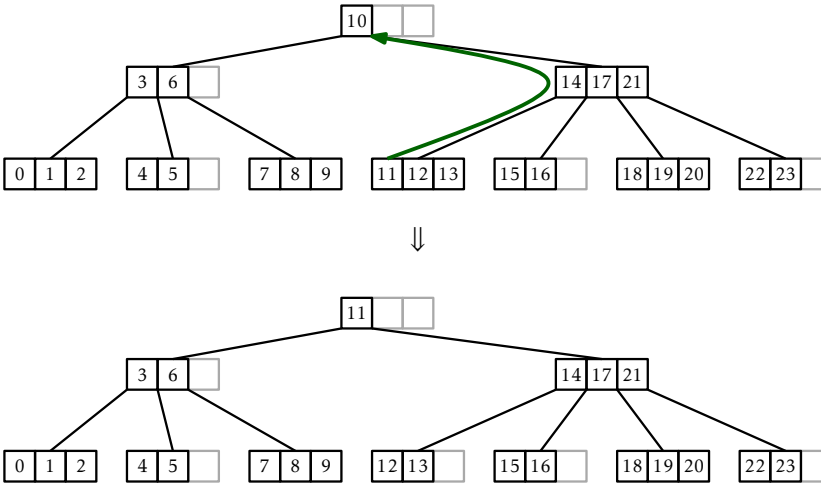
The removeRecursive(x,ui) method is a recursive implementation of the preceding algorithm:

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);
}
```

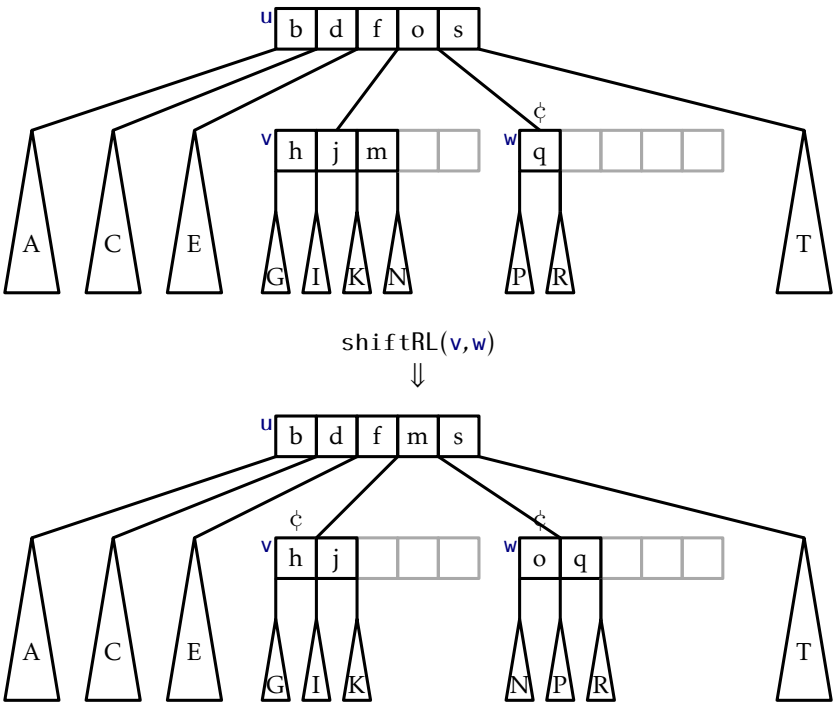
In the following, we focus on the case when $i \neq 0$ so that any underflow at the ith child of u will be corrected with the help of the $(i-1)$st child of u. The case $i=0$ is similar and the details can be found in the accompanying source code.

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:

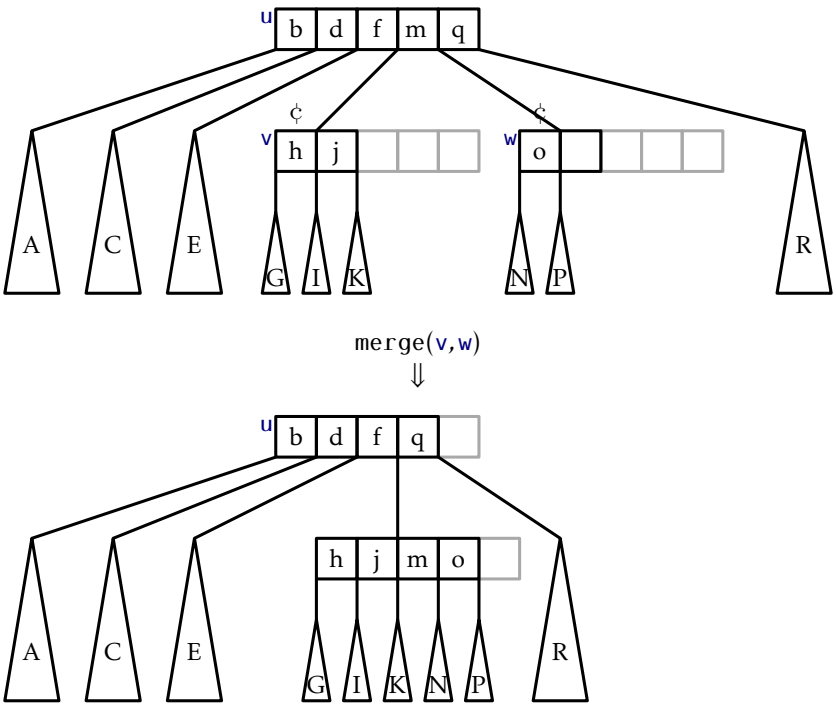
Borrowing: If w has a sibling, v, with more than $B-1$ keys, then w can borrow some keys (and possibly also children) from v. More specifically, if v stores size(v) keys, then between them, v and w have a total of

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

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 [2]. 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 [4] in 1970 and, less than ten years later, the title of Comer's ACM Computing Surveys article referred to them as ubiquitous [6]. 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 [16] 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 [20]. 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 $\alpha > 0$.

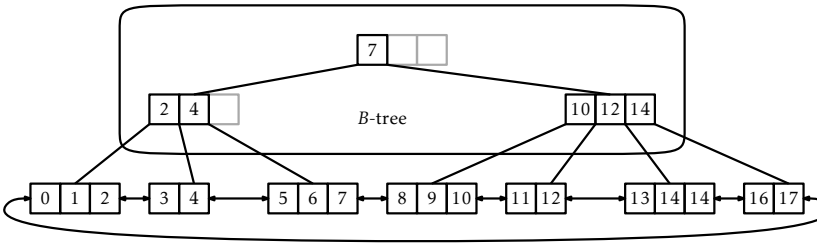
2. Let u be an underfull node and let v and w be siblings of u There are two ways to fix the underflow at u:

   (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. [8], 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. [8], pages 205–216.

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

[17] 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.

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

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

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

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

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

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

[24] 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].

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

[26] 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.

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

[28] 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.

[29] 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.

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

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

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

Stvarno kazalo

številke za klic v sili, 2

abstract data type, *glej* interface
adjacency list, 156
adjacency matrix, 153
ancestor, 79
array
    circular, 28
ArrayDeque, 31
ArrayQueue, 27
arrays, 19
ArrayStack, 21
asymptotic notation, 6
AVL tree, 130

B^*-tree, 208, 234
B^+-tree, 208, 234
B-tree, 190, 216
big-Oh notation, 6
Binarna Kopica, 135
BinarnaKopica, 135
Binarno drevo
    Popolno, 139
binary logarithm, 5
binary search, 176, 193, 219
binary search tree, 85
    height balanced, 130
    random, 94
    red-black, 109
binary search tree property, 85
binary tree, 79
    heap-ordered, 136
    search, 85
binary-tree traversal, 82
BinarySearchTree, 85
BinaryTree, 81
BinaryTrie, 170
binomial heap, 146
binomski koeficient, 6
black node, 114
black-height property, 114
block, 213, 214
block store, 189, 215
BlockStore, 189, 215
blok, 187, 188
borrow, 202, 228
BPlusTree, 211, 237
breadth-first traversal, 84
breadth-first-search, 160

celebrity, *glej* universal sink
child, 79
    left, 79
    right, 79
circular array, 28
coin toss, 12, 74

colour, 114
compare(x,y), 17
conflict graph, 151
connected components, 167
connected graph, 167
conted B-tree, 208, 234
credit invariant, 206, 232
credit scheme, 206, 232
CubishArrayStack, 51
cycle, 151
cycle detection, 165

DaryHeap, 147
datotečni sistem, 1
decreaseKey(u,y), 146
degree, 158
depth, 79
depth-first-search, 162
deque, 15
descendant, 79
dictionary, 17
directed edge, 151
directed graph, 151
disk access model, 208, 234
DLList, 57
doubly-linked list, 57
DualArrayDeque, 33
dummy node, 57

e (Euler's constant), 5
edge, 151
eksponenti, 4
Euler's constant, 5
expected running time, 11
expected value, 11
Ext4, 208, 234
external memory, 213
external memory hashing, 208, 235
external memory model, 214
external storage, 213
Eytzingerjeva metoda, 135

factorial, 5
FastArrayStack, 26
Fibonacci heap, 146
FIFO queue, 14
fusion tree, 185

Google, 3
graph, 151
    connected, 167
    strongly-connected, 167
    undirected, 166

H_k (harmonic number), 95
hard disk, 213
harmonic number, 95
heap
    binomial, 146
    Fibonacci, 146
    leftist, 146
    pairing, 146
    skew, 146
heap order, 136
heap property, 99
heap-ordered binary tree, 136
height
    in a tree, 79
    of a skiplist, 63
    of a tree, 79
height-balanced, 130
HFS+, 208, 234

I/O model, 208, 234
imenik, 1
incidence matrix, 166
indicator random variable, 11
interface, 13

Kopica, 135
    Binarna, 135

leaf, 79
left child, 79
left rotation, 101
left-leaning property, 118
left-leaning red-black tree, 118
leftist heap, 146
LIFO queue, *glej tudi* stack, 14
linearity of expectation, 11
linked list, 53
    doubly-, 57
    singly-, 54
`List`, 15
logarithm
    binary, 5
    natural, 5
logaritm, 4

map, 17
`MeldableHeap`, 141
`memcpy(d,s,n)`, 26
merge, 111, 202, 230
model zunanjega pomnilnika, 188
modular arithmetic, 27

natural logarithm, 5
no-red-edge property, 114
NTFS, 208, 234

O notation, 6
ordered tree, 79

pair, 17
pairing heap, 146
parent, 79
path, 151
pedigree family tree, 146
permutacija, 6
permutation
    random, 94
planarity testing, 166
podporno polje, 19
Popolno Binrano drevo, 139
potential, 38
potential method, 38, 129
preskočni seznam, 63
priority queue, *glej tudi* heap, 14
probability, 11

queue
    FIFO, 14
    LIFO, 14
    priority, 14

random binary search tree, 94
random permutation, 94
randomization, 11
randomized algorithm, 11
randomized data structure, 11
`RandomQueue`, 50
reachable vertex, 151
recursive algorithm, 82
red node, 114
red-black tree, 109, 118
`RedBlackTree`, 118
right child, 79
right rotation, 101

rooted tree, 79
RootishArrayStack, 39
rotation, 101
running time
    expected, 11

search path
    in a BinaryTrie, 170
    in a binary search tree, 86
    in a skiplist, 64
secondary structure, 179
sentinel node, 64
simple path/cycle, 151
singly-linked list, 54
skew heap, 146
SkiplistList, 69
SkiplistSSet, 65
SLList, 54
socialna omrežja, 1
solid-state drive, 213
source, 151
spanning forest, 167
spletno iskanje, 1
split, 111, 194, 220
square roots, 46
SSD disk, 187
SSet, 17
stack, 14
std :: copy(a0, a1, b), 26
Stirlingov približek, 6
stratified tree, 184
strongly-connected graph, 167
successor search, 17
System.arraycopy(s, i, d, j, n), 26

target, 151
tiered-vector, 49
traversal
    breadth-first, 84
    of a binary tree, 82
trdi disk, 187
Treap, 99
tree, 79
    d-ary, 146
    binary, 79
    ordered, 79
    rooted, 79
tree traversal, 82
Treque, 50

underflow, 199, 225
undirected graph, 166
universal sink, 167
USet, 16

van Emde Boas tree, 184
vertex, 151

wasted space, 44

XFastTrie, 176

YFastTrie, 179

zunanji pomnilnik, 187
zunanji spomin, 187