ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/379537333

Algoritmi i strukture podataka

Book · April 2024

CITATIONS
0

READS
5

2 authors:



Aleksandar Stojanović
Zagreb University of Applied Sciences
9 PUBLICATIONS **13** CITATIONS
SEE PROFILE



Željko Kovačević
Polytechnic of Zagreb
20 PUBLICATIONS **37** CITATIONS
SEE PROFILE

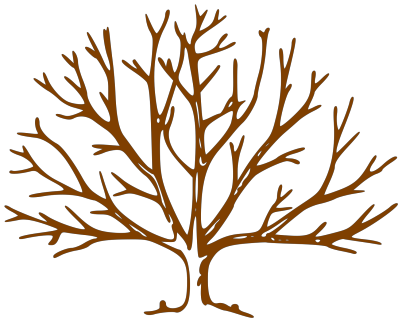
All content following this page was uploaded by Željko Kovačević on 03 April 2024.

The user has requested enhancement of the downloaded file.

TEHNIČKO VELEUČILIŠTE U ZAGREBU ZAGREB UNIVERSITY OF APPLIED SCIENCES

Algoritmi i strukture podataka

Aleksandar Stojanović
Željko Kovačević



Informatičko-računarski odjel
Tehničko veleučilište u Zagrebu
© 2024.

UDŽBENICI TEHNIČKOG VELEUČILIŠTA U ZAGREBU
MANUALIA POLYTECHNICI STUDIORUM ZAGRABIENSIS

dr. sc. Aleksandar Stojanović, v. pred.
doc. dr. sc. Željko Kovačević, v. pred.

ALGORITMI I STRUKTURE PODATAKA

Udžbenik na kolegiju Algoritmi i strukture podataka koji se održava u sklopu nastave na dodiplomskom stručnom studiju računarstva Tehničkog veleučilišta u Zagrebu

Zagreb, 2024.



izdavač	Tehničko veleučilište u Zagrebu Vrbik 8, Zagreb
za izdavača	prof. dr. sc. Jana Žiljak Gršić
autori	dr. sc. Aleksandar Stojanović, v. pred. doc. dr. sc. Željko Kovačević, v. pred.
recenzenti	dr. sc. Frane Urem, Veleučilište u Šibeniku doc. dr. sc. Domagoj Frank, Sveučilište Sjever
vrsta djela	udžbenik Objavljivanje je odobrilo Stručno vijeće Tehničkog veleučilišta u Zagrebu odlukom broj 6353-6/24 od 21.03.2024.
ISBN	978-953-8444-12-8

Sadržaj

1 Uvod **8**
1.1 Pojam algoritma i strukture podataka 8
1.2 Notacija za pseudokod 12

2 Algoritmi **14**
2.1 Analiza vremenske složenosti algoritma 15
2.1.1 Konstantna i linearna vremenska složenost 15
2.1.2 Kvadratna vremenska složenost 17
2.1.3 Stopa rasta funkcije, asimptotska analiza algoritama i asimptotska notacija 20
2.1.4 Grafički prikaz vremenske složenosti 26

3 Rekurzija **30**
3.1 Rekurzivne funkcije 30
3.2 Rekurzivno traženje i vraćanje istim putem 41
3.2.1 Primjer: Elementi u poretku s ograničenjima 45
3.3 Zadaci 48

4 Algoritmi za sortiranje **51**
4.1 Bubble sort 52
4.2 Selection sort 53
4.3 Insertion sort 54
4.4 Shell sort 56
4.5 Merge sort 58
4.6 Tim sort 59
4.7 Quick sort 60
4.8 Heap sort 61
4.8.1 Binarna gomila 62
4.8.2 Održavanje svojstva gomile 63
4.8.3 Izgradnja gomile 63
4.8.4 Algoritam *Heapsort* 66
4.8.5 Red prioriteta 68

5 Strukture podataka **72**
5.1 Tip podatka 72
5.2 Struktura podataka 74

5.3 Niz . . . 76
5.3.1 Indeksiranje niza . . . 76
5.3.2 Dodavanje i brisanje elemenata iz niza . . . 79
5.3.3 Dinamički niz . . . 81
5.3.4 Pretraživanje niza . . . 83
5.3.5 Načelo lokaliteta . . . 86
5.4 Povezana lista . . . 90
5.4.1 Jednostruko povezana lista . . . 90
5.4.2 Dvostruko povezana lista . . . 93
5.5 Stog i red . . . 96
5.5.1 Stog . . . 96
5.5.2 Red . . . 100
5.6 Stablo . . . 103
5.6.1 Binarno stablo . . . 104
5.6.2 Obilazak binarnog stabla . . . 107
5.6.3 Osnovne operacije nad stablom za binarno pretraživanje . . . 112
5.7 Tablica s izravnim adresiranjem i tablica raspršivanja . . . 119
5.7.1 Tablica s izravnim adresiranjem. . . . 119
5.7.2 Tablica raspršivanja . . . 121

Popis slika **131**

Popis algoritama **133**

Popis tablica **136**

Bibliografija **137**

Ključne riječi **139**

Predgovor

Pred vama je udžbenik pisan za potrebe studenata prijediplomskog stručnog studija računarstva na Tehničkom veleučilištu u Zagrebu. Sadržaj udžbenika obuhvaća sadržaj kolegija Algoritmi i strukture podataka, a namijenjen je svima koji žele usvojiti dodatna znanja iz područja algoritama za sortiranje i pretraživanje podataka te općepoznatih struktura za smještaj, organizaciju i obradu podataka.

Ovaj udžbenik predstavlja struktuirani pristup proučavanju algoritama, njihove analize i implementacije, kao i raznolikih struktura podataka koje služe kao osnove za organizaciju informacija. Algoritmi predstavljaju temelj svakog programa, precizno definirajući korake potrebne za riješavanje različitih problema. Kroz ovaj udžbenik studenti će razvijati sposobnost analize, optimizacije i implementacije algoritama kako bi postigli učinkovitost u rješavanju stvarnih problema u programiranju. Paralelno, razmatraju se različite strukture podataka koje omogućavaju organizaciju i upravljanje podacima. Raznolikost struktura podataka pruža mogućnost odabira optimalnog rešenja u skladu sa specifičnostima svakog zadatka. Duboko razumijevanje tih struktura podataka ključno je za razvoj učinkovitih, skalabilnih i održivih programskih sustava. Udžbenik sadrži strukturirane cjeline, praktične primjere te zadatke koji pomažu u primjeni stečenih znanja. Bez obzira na trenutno znanje i iskustvo u programiranju, cilj ovog udžbenika je pružiti studentima solidnu osnovu koja će ih pripremiti za rad u struci.

Ovim putem autori žele zahvaliti svima koji su svojim savjetima i primjedbama doprinijeli kvaliteti ovog udžbenika, a tu su bili mnogobrojni kolege iz struke, recenzenti te studenti Tehničkog veleučilišta u Zagrebu.

Autori,
Tehničko veleučilište u Zagrebu, travanj 2024.

POGLAVLJE 1

Uvod

1.1 Pojam algoritma i strukture podataka

Algoritam je detaljan opis niza koraka kojima se dolazi do rješenja nekog zadanog problema. Algoritam prima ulazni podatak ili skup podataka i daje neki izlazni podatak ili skup podataka kao rezultat [1, 2, 3]. Algoritmima se rješavaju jasno definirani računalni problemi. Oni su jedno od temeljnih područja računarstva i koriste se u širokom rasponu aplikacija, od jednostavnih zadataka poput sortiranja popisa brojeva ili traženja određene stavke u bazi podataka, do složenijih operacija poput šifriranja podataka, obrade slika, pretraživanje weba, umjetne inteligencije, i mnogih drugih.

S druge strane, struktura podataka je specifičan način organiziranja i pohranjivanja podataka u računalu tako da im se može učinkovito pristupiti. Podatkovne strukture su ključne jer pružaju sredstva za sustavno upravljanje velikim količinama podataka, kao što su baze podataka ili usluge internetskog indeksiranja. Oni također igraju vitalnu ulogu u dizajnu i implementaciji algoritama, budući da izbor odgovarajuće strukture podataka može značajno utjecati na performanse algoritma.

Postoje različite vrste struktura podataka, od kojih je svaka osmišljena da učinkovito podržava određen skup operacija. Na primjer, nizovi i popisi idealni su za pohranjivanje sekvencijalnih podataka, stabla se koriste za upravljanje hijerarhijskim podacima, a hash tablice dizajnirane su za brzo dohvaćanje podataka. Odabir podatkovne strukture često ovisi o specifičnim zahtjevima, kao što je potreba za brzim pristupom putem brojčanog indeksa ili učinkovitost umetanja i brisanja podataka.

Ukratko, algoritmi i podatkovne strukture međusobno su povezani pojmovi u središtu računarstva. Algoritam definira korake potrebne za rješavanje problema, dok strukture podataka osiguravaju sredstva za pohranjivanje i organiziranje podataka uključenih u te procese. Oni zajedno omogućuju učinkovitu obradu podataka, rješavanje problema i razvoj složenih programskih sustava.

Primjer algoritma Jedan drevni i poznati primjer algoritma je tzv. *euklidov algoritam* za pronalaženje najvećeg zajedničkog djelitelja dvaju pozitivnih cijelih brojeva. Ovaj algoritam prikazan je u pseudokodu Algoritma 1.1.

Algoritam 1.1 Euklidov algoritam za pronalaženje najvećeg zajedničkog djelitelja dvaju cijelih brojeva.

Ulaz: Dva cijela broja, a i b.
Izlaz: Najveći zajednički djelitelj brojeva a i b.

1: Ako je $a = b$ vrati a
2: Ako je $a > b$ postavi a na $a - b$ i idi na korak 1
3: Ako je $a \leq b$ postavi b na $b - a$ i idi na korak 1

Algoritam 1.1 možemo "testirati" kroz rad s konkretnim vrijednostima. Neka je $a = 18$ i $b = 21$. Algoritam 1.2 prikazuje kako ovaj algoritam dolazi do rezultata. U zagradama su naznačene vrijednosti varijabli a i b kod svakog izvršenja dotičnog koraka.

Algoritam 1.2 Euklidov algoritam za pronalaženje najvećeg zajedničkog djelitelja dvaju cijelih brojeva.

Ulaz: $a = 18$ i $b = 21$.
Izlaz: Najveći zajednički djelitelj brojeva a i b.

1: Ako je $a = b$ vrati a (1. $a = 18$, $b = 21$; 2. $a = 18$, $b = 3$; 3. $a = 15$, $b = 3$; 4. $a = 12$, $b = 3$, 5. $a = 9$, $b = 3$, 6. $a = 6$, $b = 3$, 7. $a = 3 \leftarrow rezultat$, $b = 3$)
2: Ako je $a > b$ postavi a na $a - b$ i idi na korak 1 (1. $a = 15$; 2. 12, 3. $a = 9$, 4. $a = 6$ 5. $a = 3$)
3: Ako je $a \leq b$ postavi b na $b - a$ i idi na korak 1 (1. $b = 3$)

U ovom primjeru važno je istaknuti sljedeće:

- Svaki je korak ovog algoritma jasan i nedvosmislen. To ne znači da se jedan algoritam ne smije pozivati na neki drugi. Smije, pod uvjetom da je i taj drugi algoritam opisan na sličan način. Primjerice, neki algoritam može u jednom ili više svojih koraka navesti Euklidov algoritam kako je pokazano Algoritmom 1.3 koji vraća najmanji zajednički množitelj dvaju brojeva.

- Iako se izvodi iterativno, odnosno sadrži petlju (jer se vraća na korak 1), ovaj algoritam ima uvjet prekida (u redku 1) koji osigurava konačan broj operacija koji će izvršiti. Da bi se neki ovakav niz koraka smatrao algoritmom, njegovo se izvršavanje mora u nekom trenutku prekinuti. "Iterativno" izvođenje odnosi se na računalni proces ili metodu u kojoj se slijed operacija ponavlja kako bi se postupno dobio željeni rezultat. Ovaj se pristup često koristi u programiranju i algoritmima za rješavanje problema koji zahtijevaju dolazak do rješenja korak po korak. Svako ponavljanje procesa naziva se "iteracija", a sa svakom iteracijom rješenje je obično bliže rezultatu. U kontekstu programiranja, iterativno izvođenje često uključuje petlje, kao što su for-petlje ili while-petlje, koje izvršavaju blok koda više puta. Iterativni proces

se nastavlja sve dok se ne ispuni određeni uvjet, koji je definiran problemom koji se rješava. To može biti postizanje određenog broja ponavljanja, postizanje željene razine točnosti ili pronalaženje rješenja koje zadovoljava određene kriterije.

Algoritam 1.3 Najmanji zajednički množitelj dvaju brojeva.

Ulaz: Dva cijela broja, a i b.
Izlaz: Najmanji zajednički množitelj brojeva a i b.

1: Postavi gcd na rezultat euklidovog algoritma za a i b
2: Postavi m na apsolutnu vrijednost umnoška $a * b$
3: Vrati rezultat dijeljenja m/gcd

Pogledajmo sada Algoritam 1.4. Vidimo da ovdje jedan ili više koraka nisu jasno definirani, primjerice, kako radi pronalaženje inteligentnog odgovora na P. Ovaj primjer ne prikazuje nekakav algoritam nego samo jedan općeniti postupak. Problem s opisom ovakvog postupka, sa stajališta analize algoritama, je što je teško ili nemoguće analizirati njegovu vremensku ili prostornu složenost jer mu koraci ne sadrže dovoljno detalja potrebnih za takvu analizu. Upravo iz tog razloga u definiciji algoritma koju ćemo primjenjivati u ovom udžbeniku stoji da to mora biti *niz detaljnih koraka* uz pretpostavku da se ti koraci izvršavaju konačan broj puta. Primjerice, drugi korak Algoritma 1.1 kaže *postavi* a *na* $a - b$. Ovakvi su koraci dovoljno jasni da mogu biti dio opisa nekog algoritma.

Algoritam 1.4 Općeniti postupak, ali ne i algoritam.

Ulaz: Upit na engleskom jeziku, P.
Izlaz: Inteligentan odgovor na P.

1: Učitaj i obradi upit P.
2: Pronađi inteligentan odgovor na P.
3: Ispiši odgovor na engleskom jeziku.

Upotreba algoritama i struktura podataka u rješavanju problema Svaki je program implementacija nekog algoritma, bez obzira na to radi li se o nekom postojećem algoritmu ili algoritmu koji je smišljen specifično za neki trenutni problem koji rješavamo. Općenito, pri pisanju programa moežmo slijediti ove korake korake:

1. **Opis programskog problema** Jasna i koncizna izjava problema je ključna. Razmotrimo jednostavan primjer: "Napišite program koji pronalazi najveći broj u nizu cijelih brojeva."

2. **Opis algoritma u pseudokodu** Prije pisanja koda u nekom programskom jeziku korisno je skicirati ga u pseudokodu. Pseudokod za naš problem može izgledati ovako:

```
inicijaliziraj M s elementom niza na indeksu 0
```

```
za svaki indeks I u rasponu od 1..N, gdje je N posljednji indeks
     neka je X = element na indeksu I
     ako je X veći od M postavi M na X
vrati M
```

3. **Opis tipova podataka** Razumijevanje potrebnih vrsta podataka je ključno. Za naš problem, ulaz je popis (ili niz) cijelih brojeva, a izlaz jedan cijeli broj koji predstavlja najveći broj.

4. **Pisanje koda u programskom jeziku** Sljedeći korak je prevođenje pseudokoda u kôd nekog programskog jezika kao što je C:

```
#include <stdio.h>

int najveci_broj(int niz[], int br_elemenata) {
    int m = niz[0];
    for (int i = 1; i < br_elemenata; ++i)
        if (niz[i] > m) m = niz[i];

    return m;
}

int main() {
    int niz[] = {3, 50, 7, 2, 99, 23, 45, 67};
    int n = sizeof(niz) / sizeof(niz[0]);
    int najveci = najveci_broj(niz, n);
    printf("Najveći broj je: %d\n", najveci);
    return 0;
}
```

5. **Testiranje** Testiranje uključuje provjeru da program radi ispravno za različite slučajeve. Za naš primjer trebali bi testirati:

   - popis pozitivnih brojeva
   - popis koji uključuje negativne brojeve
   - popis sa svim brojevima koji su isti
   - prazan niz

6. **Rješenje** Rješenje obuhvaća ispravno napisan, testiran i verificiran kod koji rješava zadani problem. Za naš primjer, nakon temeljitog testiranja, rješenje bi bila konačna verzija funkcije *najveci_broj*.

Iako smo u ovom primjeru koristili jednostavan primjer, prethodno opisan postupak primjenjiv je i na puno složenije programe.

Primjena algoritama i struktura podataka Algoritmi i strukture podataka temeljni su elementi računalne znanosti koji nalaze primjenu u širokom rasponu područja, pružajući učinkovita rješenja za složene probleme. Na primjer, u web tražilicama kao što je Google, algoritmi kao što je PageRank, uz strukture podataka kao što su obrnuti indeksi i grafovi, koriste se za indeksiranje i rangiranje web stranica, omogućujući korisnicima brzo pronalaženje relevantnih informacija. Na platformama društvenih mreža kao što je Facebook, algoritmi se koriste za generiranje vijesti i preporuka prijatelja, oslanjajući se na podatkovne strukture kao što su stabla i tablice raspršivanja za učinkovito upravljanje i obradu ogromnih količina korisničkih podataka. U području zdravstvene zaštite, algoritmi strojnog učenja, koji koriste sofisticirane strukture podataka za rukovanje velikim skupovima podataka, primjenjuju se za predviđanje izbijanja bolesti, personaliziranje planova liječenja i analizu medicinskih slika. U financijskom sektoru algoritmi se koriste u visokofrekventnim sustavima trgovanja za izvršavanje naloga po optimalnim cijenama, koristeći podatkovne strukture poput redova čekanja i nizova za upravljanje protokom transakcija i tržišnih podataka. Dodatno, u području navigacijskih sustava kao što su Google Maps, algoritmi kao što su Dijkstra ili A* pretraga, zajedno sa strukturama podataka kao što su redovi i grafovi, koriste se za pronalaženje najkraćih putova i generiranje ažuriranja prometa u stvarnom vremenu. Ovi primjeri ilustriraju sveprisutni i transformativni utjecaj algoritama i struktura podataka u učinkovitom i inovativnom rješavanju problema iz stvarnog svijeta.

1.2 Notacija za pseudokod

Da bismo jasnije opisali razne algoritme bez upotrebe nekog konkretnog programskog jezika, koristit ćemo notaciju koja je uobičajena u literaturi i razumljiva je svakome s osnovnim iskustvom u nekom programskom jeziku kao što su C++, C#, Java i Python. Tako ćemo, primjerice, Algoritam 1.1 prikazati kao Algoritam 1.5.

Algoritam 1.5 Funkcija za pronalaženje najvećeg zajedničkog djelitelja dvaju pozitivnih cijelih brojeva prema Euklidovom algoritmu.

Ulaz: Dva pozitivna cijela broja, a i b.
Izlaz: Najveći zajednički djelitelj brojeva a i b.

```
1: function EUKLID(a, b)
2:     while a ≠ b do
3:         if a > b then
4:             a ← a − b
5:         else
6:             b ← b − a
7:         end if
8:     end while
9:     return a
10: end function
```

U ovom udžbeniku koristit ćemo *function* za pseudokod koji vraća neku vrijednost kao rezultat ili *procedure* za onaj koji ne vraća nikakav rezultat. Funkcije i procedure kao što

je EUKLID pisat ćemo tzv. malim velikim slovima (engl. *small caps*), a lokalne varijable kao što su a i b u Primjeru 1.5 malima. Za operator pridruživanja koristit ćemo simbol $\leftarrow$.

Tip varijable Na nekim mjestima u pseudokodu ponekad ćemo iza naziva varijable (što uključuje i polja) pisati dvotočku i tip vrijednosti koju ona sadrži. Primjerice, ako želimo istaknuti da varijabla x sadrži adresu instance tipa DATUM pisat ćemo x : DATUM kako je pokazano Algoritmom 1.6.

Algoritam 1.6 Primjer označavanja tipa varijable.

```
1: procedure NEKA-PROCEDURA(x : DATUM)
2:     ...
3: end procedure
```

Atributi U nekim slučajevima vrijednosti pridružene varijablama sadržavat će podatke o tim vrijednostima kao što su, primjerice, veličina niza ili tekstualne vrijednosti. Te podatke nazivat ćemo *atributima* i pisat ćemo ih za imena varijable odvojene točkom. Ovo je prikazano u Algoritmu 1.7.

Algoritam 1.7 Primjer korištenja atributa.

```
1: procedure NEKA-PROCEDURA(A : niz)
2:     if A.broj-elemenata > 0 then   ▷ broj-elemenata je atribut vrijednosti nizovnog tipa
3:         A.broj-elemenata ← 0
4:     end if
5: end procedure
```

U ovom primjeru praktičnije je koristiti atribut umjesto funkcije, kao BROJ-ELEMENATA(A), jer atributima možemo pridruživati vrijednosti.

Algoritam 1.8 prikazuje pseudokod funkcije koja vraća najveći element u zadanom nizu brojeva.

Algoritam 1.8 Pseudokod za pronalazak najvećeg elementa u nizu brojeva.

```
1: function NAJVEĆI(A : niz)
2:     m ← A[0]
3:     for i ← 1..A.broj-elemenata − 1 do
4:         if A[i] > m then
5:             m ← A[i]
6:         end if
7:     end for
8:     return m
9: end function
```

U redku 3, oznaka *1..A.broj-elemenata - 1* znači da će u svakoj iteraciji varijabli i biti pridružen idući broj intervala od 1 do *A.broj-elemenata - 1*. Petlju u redku 3 možemo čitati kao *za svaki "i" u intervalu 1 do A.broj-elemenata - 1 napravi ...*.

POGLAVLJE 2

Algoritmi

Mnogi popularni programski jezici isporučuju se sa standardnom bibliotekom (engl. *standard library*) koja se sastoji od velikog broja gotovih dijelova koda za razne potrebe. Svaka takva biblioteka obično sadrži i gotove dijelove koda za razne strukture podataka i algoritme kao što su oni za pretraživanje i sortiranje. U mnogim jezicima neke su od tih struktura podataka ugrađene u sam programski jezik, odnosno nisu dio standardne biblioteke nego imaju posebnu sintaksu za konstrukte namijenjene za rad s njima. Učinkovita upotreba tih struktura podataka zahtijeva razumijevanje njihovih karakteristika, kako sa strane funkcionalnosti tako i sa strane performansi. Primjerice, dok su neke od njih učinkovite za pristup podacima putem brojčanog indeksa, druge su učinkovitije za pristup podacima putem ključa. Isto tako, neke su učinkovitije za česta brisanja, a druge za česta dodavanja novih podataka.

Svaki se algoritam može promatrati iz dvaju gledišta: koliko mu je vremena potrebno da bi došao do rezultata te kolike su mu u tome potrebe za memorijom. Ovo prvo naziva se *vremenskom složenošću*, a ovo drugo *prostornom složenošću*. U ovom udžbeniku bavit ćemo se isključivo vremenskom složenošću, ali analiza prostorne složenosti izvodi se na isti način. Vremenska složenost algoritma odnosi se na njegovu brzinu izvršavanja, odnosno na količinu vremena potrebnu za pronalazak rješenja. Zašto nam je vremenska učinkovitost važna? Ovdje se je lako osloniti na brz razvoj računalnih komponenti kao što su procesori, memorija i grafičke kartice. Usporedo s razvojem računala drastično se povećala i količina podataka koja se danas njima obrađuje. Nekada se niz od 1.500 elemenata smatrao velikim dok je to danas zanemariva količina podataka. Međutim, ako smo zadovoljni vremenom potrebnim programu za sortiranje, recimo, 100.000 elemenata, hoćemo li biti zadovoljni i vremenom koje mu treba za sortiranje $1.000,000$ elemenata? Ili $10.000,000$? Samo zato što nam se čini da neki program radi prihvatljivo brzo s jednom količinom podataka ne znači da će tako raditi i s većim količinama. Analiza algoritama može nam dati odgovor upravo na to pitanje. Umjesto da se usredotočimo na vrijeme potrebno nekom programu da nešto napravi, analiza algoritama usredotočuje se na utvrđivanje odnosa vremena izvršavanja i količine podataka. Na taj način možemo predvidjeti kakve će biti performanse nekog algoritma za bilo koju količinu podataka.

Pored vremenske, važna je i energetska učinkovitost. Računala danas više nisu samo stacionarna i spojena u električnu mrežu, već su i mobilna te se napajaju baterijom. Ovdje je jasno da će program koji u dvije minute pronalazi optimalni put u gradu od točke A do točke B potrošiti više baterije nego onaj koji to napravi u dvije sekunde. Radi se o jednostavnom broju instrukcija koje računalo mora izvršiti da bi došlo do rezultata - više instrukcija zahtijeva više energije. Prema tome, cilj izrade učinkovitog postupka rješavanja nekog problema je taj problem riješiti izvršavanjem što manje instrukcija. Time će taj postupak biti ne samo vremenski nego i energetski učinkovitiji.

2.1 Analiza vremenske složenosti algoritma

Cilj je analize algoritma utvrditi njegovu *vremensku* ili *prostornu* složenost. Analiza algoritama često se u praksi koristi za uspoređivanje performansi dvaju ili više algoritama koji imaju istu namjenu (primjerice, usporedba dvaju algoritama sortiranja ili pretraživanja). Zašto je neki algoritam "brži" od drugoga i što to znači? S obzirom da je algoritam apstrakcija, a ne računalni program, za njegovu analizu moramo identificirati neke ključne operacije za čije ćemo izvođenje pretpostaviti da je potrebna jedna vremenska jedinica. Na taj način možemo reći, primjerice, da pridruživanje kao što je $x \leftarrow a + b * c$ treba jednu vremensku jedinicu. S tehničke strane, ovakvo pridruživanje može se sastojati od dvije do tri instrukcije procesora, zbrajanja i množenja (i možda spremanja rezultata negdje u memoriju ili registar). Kod analize algoritama takve detalje ne uzimamo u obzir jer ne utječu na karakteristike algoritma koje su nam važne. Važno nam je koliko se takvih operacija mora izvršiti da bi algoritam došao do rješenja.

Vremensku složenost algoritama možemo podijeliti u nekoliko kategorija ovisno o matematičkoj funkciji koja opisuje odnos količine ulaznih podataka ili *veličine ulaza* i vremena potrebnog algoritmu da dođe do rješenja. U nastavku su kroz primjere ilustrirane tri takve kategorije vremenske složenosti algoritama: konstantna, linearna i kvadratna te je prikazan način kako se došlo do njih. Iako su algoritmi u ovom dijelu prilično jednostavni, na isti se način analiziraju i puno složeniji algoritmi.

2.1.1 Konstantna i linearna vremenska složenost

Algoritam 2.1 daje zbroj cijelih brojeva u intervalu $[1, 10]$. Ovdje možemo identificirati dvije ključne operacije: prva je inicijalizacija $a \leftarrow 0$, a druga zbrajanje $a \leftarrow a + i$ koje se ponavlja 10 puta. Dakle, ovaj algoritam mora izvršiti 11 operacija da bi došao do rješenja.

Algoritam 2.1 Zbroj cijelih brojeva u intervalu $[1, 10]$.

```
1: function ZBROJ
2:     a ← 0
3:     for i ← 1..10 do
4:         a ← a + i
5:     end for
6:     return a
7: end function
```

Ovaj algoritam *uvijek* izvršava isti broj operacija jer zbraja brojeve u unaprijed postavljenom intervalu. Drugim riječima, broj operacija koje mora izvršiti ne ovisi o veličini ulaza. Za takve algoritme reći ćemo da imaju *konstantnu vremensku složenost* jer je za njihovo izvršavanje uvijek potreban isti broj koraka, neovisno o veličini ulaza. Pogledajmo sada jednu inačicu ovog algoritma prikazanu u Algoritmu 2.2.

Algoritam 2.2 Zbroj cijelih brojeva u intervalu [1, n].

```
1: function ZBROJ(n)
2:     a ← 0
3:     for i ← 1..n do
4:         a ← a + i
5:     end for
6:     return a
7: end function
```

S obzirom da ovdje ne zbrajamo brojeve u nekom unaprijed postavljenom intervalu nego je taj interval zadan kao parametar n, broj koraka ovog algoritma ovisi o tom parametru. Parametar n predstavlja veličinu ulaza, što je u ovom primjeru veličina intervala unutar kojeg treba zbrojiti vrijednosti. Što je n veći to je veći i broj koraka koje ovaj algoritam mora izvršiti, a time i vrijeme potrebno da dođe do rezultata. Ako sada želimo ustanoviti vremensku složenost ovog algoritma, moramo utvrditi na koji način njegov broj koraka ovisi o parametru n. Drugim riječima, ako, primjerice, udvostručimo n, hoće li se i broj koraka udvostručiti? Ovdje je jednostavno utvrditi odnos veličine n i broja koraka jer se to jasno vidi u petlji u redcima $3..5$. Vidimo da se operacija $a \leftarrow a+i$ izvršava jednom za svaku vrijednost u zadanom intervalu pa je jasno da ako n povećamo x puta, proporcionalno tome povećat će se i broj operacija. Ovo možemo izraziti preciznije: ako je n veličina ulaza, a c broj konstantnih operacija koje se moraju izvršiti n puta, ukupan broj operacija koje ovaj algoritam mora izvršiti bit će cn. Zbog toga kažemo da ovakav algoritam ima *linearnu vremensku složenost* [4, 5].

Algoritam kao što je 2.2 nije najučinkovitiji algoritam za zbrajanje brojeva u intervalu $1..n$. Isti problem možemo riješiti i kako je prikazano u Primjeru 2.3.

Algoritam 2.3 Brži zbroj cijelih brojeva u intervalu [1, n].

```
1: function ZBROJ(n)
2:     return n * (n + 1)/2
3: end function
```

Iako ovaj algoritam također ovisi o parametru n, broj operacija koje on mora izvršiti ne ovisi o n, odnosno ovdje n ne predstavlja veličinu ulaza. Očito je da i ovaj algoritam ima konstantnu vremensku složenost kao i Algoritam 2.1.

Algoritam 2.4 prikazuje još jedan algoritam s linearnom vremenskom složenošću. Ovaj algoritam traži zadanu vrijednost u neuređenom nizu. Da bismo saznali postoji li neki element u takvom nizu, moramo u najjednostavnijem slučaju ispitati svaki indeks po redu dok ne naiđemo na traženi element ili dok ne dođemo do kraja niza. Jasno je, prema tome,

da vrijeme potrebno za takav algoritam ovisi o veličini niza. Ovakvo se pretraživanje zove *linearno pretraživanje* i najjednostavnija je tehnika pretraživanja niza [6].

Algoritam 2.4 Linearno pretraživanje.

Parametri: A - niz koji pretražujemo; x - vrijednost koju tražimo.
Rezultat: indeks vrijednosti koju tražimo ili -1 ako ona ne postoji.

```
1: function LINEARNO-PRETRAŽIVANJE(A, x)
2:     for i ← 1..A.veličina-niza do
3:         if A[i] = x then
4:             return i                          ▷ vrati indeks traženog elementa
5:         end if
6:     end for
7:     return -1                                 ▷ traženi element ne postoji
8: end function
```

Za razliku od Algoritma 2.2, ovaj algoritam ne mora nužno proći sve elemente niza. Ako se, primjerice, traženi element nalazi na početku niza, algoritam mora proći samo jedan element. Kod ovog algoritma možemo razlikovati dva osnovna slučaja:

1. najbolji slučaj – traženi element je prvi u nizu
2. najgori slučaj – traženi element se ne nalazi u nizu.

Čini se da je kod ovog algoritma teže zaključiti koja je njegova vremenska složenost. U jednom je slučaju konstantna (najbolji slučaj), u drugom linearna (najgori slučaj), a u ostalima nešto između. Ovdje se moramo pitati što nam je važno. Ako nam je važan najgori slučaj, isto tako bit će nam važno u kojoj mjeri je taj slučaj moguć. Ako govorimo o ovom konkretnom primjeru, hoće li se traženi element najčešće nalaziti odmah na početku niza ili ga možda najčešće neće biti u nizu? Hoće li se možda često nalaziti blizu kraja niza ili negdje u sredini? Ovo su važna pitanja na koja odgovor uglavnom ovisi o konkretnoj aplikaciji. S obzirom da ovdje nemamo konkretan problem za koji je ovakvo pretraživanje niza dio rješenja, pretpostavit ćemo da je za svaki element jednako vjerojatno da se nađe bilo gdje u nizu. Iz tog se razloga u analizi algoritama često usredotočujemo na najgori slučaj.

2.1.2 Kvadratna vremenska složenost

Algoritam 2.5 brojeve iz zadanog niza preraspoređuje tako da se nalaze u uzlaznom sortiranom redoslijedu.
Koje su ključne operacije ovog algoritma? S obzirom da se radi o algoritmu sortiranja jasno je da se on zasniva na uspoređivanju elemenata i njihovom razmiještanju unutar niza. Slika 2.1 ilustrira rad ovog algoritma. U prvom prolasku kroz niz, počevši od indeksa 0, pronađe se najmanji element segmenta $[0..(A.\textit{veličina-niza} - 1)]$ i taj se element na kraju tog prolaska smješta na indeks 1. U drugom prolasku, počevši od indeksa 1, opet se provodi isti postupak čime se najmanji element segmenta $[1..(A.\textit{veličina-niza} - 1)]$ smješta na indeks 1. Ovaj se postupak ponavlja za segmente $[2..(A.\textit{veličina-niza} - 1)]$,

Algoritam 2.5 Algoritam za sortiranje niza.

Parametri: A - niz brojeva.

```
1: procedure SORTIRANJE(A)
2:     for i ← 0..A.veličina-niza − 2 do
3:         m ← i
4:         for j ← (i + 1)..A.veličina-niza − 1 do
5:             if A[j] < A[m] then
6:                 m ← j            ▷ element na indeksu m manji je od onog na indeksu j
7:             end if
8:         end for
9:         zamijeni A[i] i A[m]
10:    end for
11: end procedure
```

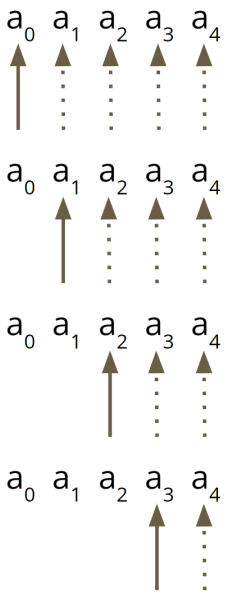
$[3..(A.\textit{veličina-niza} - 1)]$, ..., $[(A.\textit{veličina-niza} - 2)..(A.\textit{veličina-niza} - 1)]$, nakon čega je niz sortiran.

Sada kada poznajemo način rada ovog algoritma možemo analizirati koliko uspoređivanja mora provesti za neki niz vrijednosti. Za početak možemo pogledati koliko bi uspoređivanja bilo potrebno za neki konkretan broj vrijednosti, recimo 5. U prvom prolasku kroz niz prvi element uspoređujemo s četiri preostala elementa, drugi element s tri, treći s dva i četvrti s jednim. Ukupan broj uspoređivanja je $4 + 3 + 2 + 1 = 10$. Možemo li iz ovoga zaključiti da bi nam za dvostruko veći niz, odnosno niz od 10 elemenata, trebalo dvostruko više uspoređivanja, odnosno 20? Na ovo pitanje možemo odgovoriti ako generaliziramo broj elemenata na n. Tada prvi element uspoređujemo s $n - 1$ elemenata, drugi s $n - 2$, treći s $n - 3$, itd. Prema tome, za niz od n elemenata vrijedi

$$(n-1) + (n-2) + ... + 3 + 2 + 1 = \frac{n(n+1)}{2} - n = \frac{1}{2}n^2 - \frac{1}{2}n \tag{2.1}$$

Vidimo da će nam za 10 elemenata biti potrebno 45 usporedbi, a ne 20, dok bi nam, primjerice, za 100 elemenata bilo potrebno 4950 usporedbi. Isto tako, vidimo da gornja formula predstavlja kvadratnu funkciju, odnosno funkciju oblika $f(x) = ax^2 + bx + c$ gdje je $a = \frac{1}{2}$, $b = -\frac{1}{2}$ i $c = 0$. Iz tog razloga ovakva se vremenska složenost zove *kvadratna vremenska složenost* i karakteristična je za mnoge algoritme.

Broj usporedbi koje ovaj algoritam izvodi ne ovisi o poretku elemenata ulaznog niza. Ako je, primjerice, ulazni niz već sortiran ovaj će algoritam napraviti isti broj prolazaka kroz niz kao i da nije. Prema tome, jedno poboljšanje Algoritma 2.5 bilo bi da prekinemo njegovo izvršavanje ako je zadani niz već sortiran, kao što je prikazano Algoritmom 2.6. Funkcija za provjeru sortiranosti niza prikazana je u Algoritmu 2.7. Ova funkcija mora proći cijeli niz jednom pa je njena vremenska složenost linearna. Kolika je sada vremenska složenost Algoritma 2.6? Iako se na prvi pogled može činiti da smo ovim usporili polazni Algoritam 2.5, ipak će u ovoj novoj inačici najveći dio vremena biti potrošen na redke $5..13$. U nastavku ćemo vidjeti koja je vremenska složenost Algoritma 2.6, ali prije toga je neophodno upoznati se s asimptotskom analizom algoritama.



Slika 2.1: Način rada algoritma u primjeru 2.5.

Algoritam 2.6 Poboljšani algoritam sortiranja 2.5.

Parametri: A - niz vrijednosti.

```
1:  procedure SORTIRANJE(A)
2:      if SORTIRAN(A) then
3:          return
4:      end if
5:      for i ← 0..A.veličina-niza − 2 do
6:          m ← i
7:          for j ← (i + 1)..A.veličina-niza − 1 do
8:              if A[j] < A[m] then
9:                  m ← j              ▷ element na indeksu m manji je od onog na indeksu j
10:             end if
11:         end for
12:         zamijeni A[i] i A[m]
13:     end for
14: end procedure
```

Algoritam 2.7 Provjera sortiranosti niza.

Parametri: A - niz vrijednosti.
Rezultat: $true$ ako je A sortiran, inače $false$.

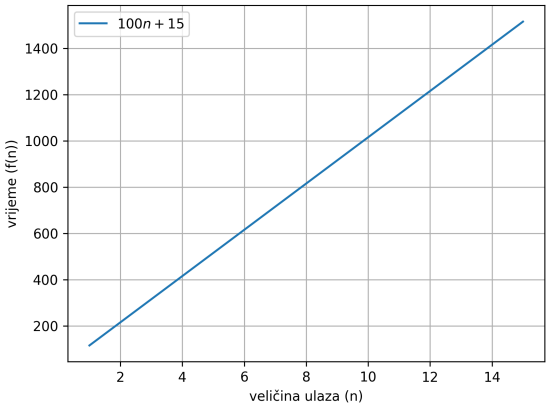
```
1: function SORTIRAN(A)
2:     for i ← 2..length(A) do
3:         if A[i] < A[i − 1] then
4:             return false
5:         end if
6:     end for
7:     return true
8: end function
```

2.1.3 Stopa rasta funkcije, asimptotska analiza algoritama i asimptotska notacija

U ovom poglavlju bavimo se *asimptotskom analizom algoritama* (engl. *asymptotic algorithm analysis*) [7, 8, 9] . Prethodno smo rekli da nas kod analize vremenske složenosti algoritma zanima kako veličina ulaza utječe na njegovo vrijeme izvršavanja, ali to se može definirati preciznije: *vremenska složenost algoritma matematička je funkcija koja opisuje odnos između veličine ulaza i vremena izvršavanja algoritma* [1, 2, 10, 11]. Pod veličinom ulaza podrazumijevamo podatak ili skupinu podataka, kao što je niz. U ovom udžbeniku najčešće radimo sa skupinama podataka, ali postoji velik broj algoritma koji rade s malim brojem podataka gdje određene karakteristike tih podataka određuju vremensku složenost algoritma (primjerice, dva broja za koje treba naći najveći zajednički djelitelj).

Za opis vremenske složenosti algoritma koristit ćemo matematičke funkcije kao one koje smo vidjeli u prethodnom dijelu. Jedna od prednosti takvog opisa vremenske složenosti je što omogućuje prikaz grafa funkcije kojom je definirana, kao što je prikazano Slikom 2.2. U ovom kontekstu, kada govorimo o vremenu, ne mislimo na konkretne vremenske jedinice kao što su sekunde i minute. Isto tako, kada govorimo o veličini ulaza, ne mislimo na megabajte, gigabajte i sl. To je zato jer nas zanima odnos vremena i veličine ulaza. Primjerice, ako iz vremenske složenosti nekog algoritma vidimo da se vrijeme udvostruči kada se veličina ulaza udvostruči, nisu nam važne mjerne jedinice. Još je jedna prednost opisa vremenske složenosti matematičkom funkcijom, i ona najvažnija, da nam omogućava predviđanje performansi algoritama matematičkim metodama. Iz istog se razloga matematičke funkcije koriste u znanosti.

Zašto se ovakva analiza algoritama zove *asimptotska* analiza? U matematici, asimptotska analiza metoda je opisa graničnog rasta funkcije. Uzmimo za primjer funkciju $f(x) = x^2 + 2x + 1$. Član koji ima najveći utjecaj na stopu rasta ove funkcije je x^2. To možemo vidjeti u Tablici 2.1. U posljednjem stupcu prikazan je kao postotak odnos veličine $2x + 1$ prema x^2. Kako x raste, $2x + 1$ ima sve manji utjecaj na ukupni rezultat. Preciznije, rekli bismo da je $f(x)$ *asimptotski jednaka* x^2 kada x raste u beskonačnost. U analizi algoritama često ćemo opisivati vremensku složenost ovakvim polinomom pa će



Slika 2.2: Primjer grafa funkcije koja predstavlja vremensku složenost.

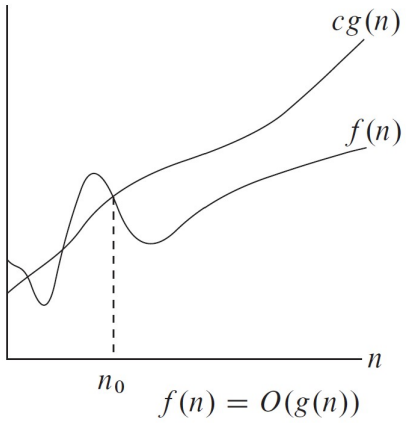
x	x^2	$2x+1$	$2x+1$ prema x^2
1	1	3	300.0%
10	100	21	21.0%
50	2500	101	4.04%
100	10000	201	2.01%
500	250000	1001	0.4%
1000	1000000	2001	0.2%
5000	25000000	10001	0.04%
10000	100000000	20001	0.02%
100000	10000000000	200001	0.002%
500000	250000000000	1000001	0.0004%
1000000	1000000000000	2000001	0.0002%
∞	...	...	0%

Tablica 2.1: Utjecaj člana x^2 na ukupni rezultat funkcije $f(x) = x^2 + 2x + 1$ kada x raste u beskonačnost.

najveći utjecaj na stopu rasta takve funkcije imati član s najvećim eksponentom.

Za algoritam sortiranja (2.5) rekli smo da mu je vremenska složenost kvadratna jer graf matematičke funkcije koja određuje koliko mu je koraka potrebno za ulazni niz veličine n po obliku odgovara kvadratnoj funkciji. Iz sličnog razloga za algoritam linearnog pretraživanja (2.4) rekli smo da mu je vremenska složenost linearna. Na taj smo način izostavili puno detalja samog algoritma, ali smo se usredotočili na ono što nam je najvažnije: odnos broja koraka i veličine ulaznog niza.

Za opis vremenske složenosti koristi se tzv. *asimptotska notacija* (engl. *asymptotic notation)*. Ova se notacija odnosi na funkcije. Primjerice, za Algoritam 2.4 (linearno pretraživanje) rekli smo da mu je potrebno kn koraka u najgorem slučaju. Upotrebom asimptotske notacije rekli bismo da je vremenska složenost tog algoritma $\Theta(n)$ u najgorem slučaju. Slično, za Algoritam 2.6 (sortiranje) rekli bismo da mu je vremenska složenost $\Theta(n^2)$ u na-



Slika 2.3: Odnos $f(n)$ i $g(n)$ kod O-notacije [1].

jgorem slučaju. Za opis vremenske složenosti upotrebom asimptotske notacije navodimo samo one članove koji imaju najveći eksponent, s tim da izostavljamo konstante. Za kn to je n, a za $\frac{1}{2}n^2 - \frac{1}{2}n$ to je n^2. Ako se prisjetimo dijela o asimptotskoj analizi, članovi s najvećim eksponentom najviše utječu na rast funkcije pa su nam oni i najvažniji za opis vremenske složenosti. Primjerice, kod vremenske složenosti algoritma sortiranja najveći utjecaj na rast funkcije $\frac{1}{2}n^2 - \frac{1}{2}n$ imat će n^2.

Asimptotska notacija sastoji se od nekoliko simbola. Da bismo te simbole mogli ispravno koristiti i razumijeti moramo ih precizno definirati.

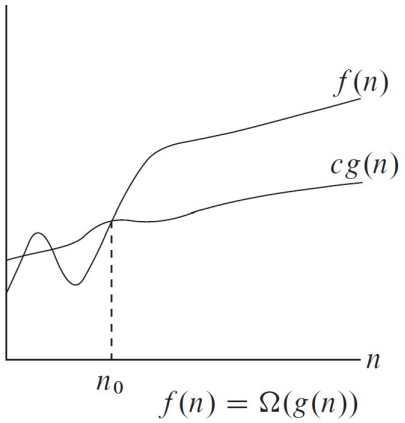
2.1.3.1 O-notacija

O-notaciju koristimo kada želimo specificirati *asimptotsko ograničenje odozgo* [1]. Ovom notacijom kažemo da neka funkcija vremenske složenosti ne raste brže od navedene stope rasta. Primjerice, stopa rasta za funkciju $4n^2 + 17n + 3$ je n^2 zato što je član s najvećim eksponentom $4n^2$. Stoga stopu rasta ove funkcije možemo specificirati s $O(n^2)$ čime smo rekli da ona ne raste brže od n^2. Isto tako, možemo reći da ne raste brže od n^3, n^4, n^5, itd. Zbog toga tehnički je ispravno napisati da je ta funkcija $O(n^3)$, $O(n^4)$, $O(n^5)$, itd. Drugim riječima, ako neka funkcija ne raste brže od n^2 ona ne može rasti brže niti od n^3, n^4 ili bilo koji drugi veći eksponent. Ovo možemo generalizirati i reći da je ova funkcija $O(n^c)$ za bilo koju konstantu $c \geq 2$.

Formalno [1], $O(g(n))$ je skup funkcija

$$O(g(n)) = \{\ f(n) : \text{postoji konstanta } c \text{ i } n_0 \text{ takva da je } 0 \leq f(n) \leq cg(n) \text{ za svaki } n \geq n_0\ \}.$$

Na Slici 2.3 ilustrirana je O-notacija. Kažemo da je $f(n) = O(g(n))$ ako postoje pozitivne konstante n_0 i c takve da na točki n_0 ili desno od nje vrijednost funkcije $f(n)$ uvijek leži na ili ispod $cg(n)$. Odnosno, $f(n)$ pripada skupu $O(g(n))$ ako postoji nenegativna konstanta c za koju je $f(n) \leq cg(n)$ za svaki $n \geq n_0$.



Slika 2.4: Odnos $f(n)$ i $g(n)$ kod Ω-notacije [1].

Ovu definiciju možemo iskoristiti da bismo pokazali da je neka funkcija $f = O(g(n))$. Primjerice, $f(n) = 3n^2 - 92n + 5 = O(n^2)$ jer za $c = 3$, $3n^2 > f(n)$. Isto tako možemo pokazati da je $f(n) = O(n^3)$ time što je $c = 1$, $n^3 > f(n)$. Ovu definiciju možemo koristiti i da pokažemo da $f(n) \neq O(n)$. Razlog je taj da za svaku konstantu $c > 0$, $cn < f(n)$ kada je $n > (c + 92)/3$.

Ono što O-notaciju čini praktičnom je činjenica da gornju granicu nekog algoritma često možemo lako zaključiti analizom strukture tog algoritma. Primjerice, algoritam sortiranja 2.5 sastoji se od dviju petlji gdje se jedna nalazi unutar druge. Vidimo da obje petlje ovise o veličini ulaznog niza $n = length(A)$ pa možemo zaključiti da je ograničenje odozgo ovog algoritma $O(n^2)$.

Ova se notacija često upotrebljava neformalno. Što znači kada kažemo da algoritam sortiranja 2.5 ima vremensku složenost $O(n^2)$? Iako ovdje nije navedeno na koji se to slučaj odnosi, najčešće se podrazumijeva najgori slučaj pa bi to značilo da mu je ograničenje odozgo n^2 u najgorem slučaju. Nadalje, O-notaciju možemo tumačiti i ovako: kada $O(f(n))$ koristimo za opis najgoreg slučaja, time smo obuhvatili *sve* slučajeve jer kakvi god da su ulazni podaci vrijeme izvršavanja neće rasti brže od $f(n)$.

2.1.3.2 Ω-notacija

Ako želimo izraziti *asimptotsko ograničenje odozdo* koristimo Ω-notaciju [1]. Formalno [1], Ω se definira kao

$$\Omega(g(n)) = \{\ f(n) : \text{postoji konstanta } c \text{ i } n_0 \text{ takva da je } 0 \leq cg(n) \leq f(n) \text{ za svaki } n \geq n_0\ \}.$$

Na Slici 2.4 ilustrirana je Ω-notacija. Ovdje na sličan način kao i kod O-notacije možemo utvrditi je li neka funkcija $f(n) = \Omega(g(n))$. Kada napišemo da je $f(n) = \Omega(g(n))$ to znači da $f(n)$ ne raste *sporije* od $cg(n)$ za neku nenegativnu konstantu c.

2.1.3.3 Θ-notacija

Θ-notacija predstavlja asimptotsko *tijesno* ograničenje odozgo i odozdo [1]. Za razliku od Θ-notacije, O-notacija ne mora nužno (ali može) označavati tijesno ograničenje. Primjerice, $n \neq \Theta(n^2)$, ali $n = O(n^2)$. Isto tako, možemo reći da je vremenska složenost algoritma za linearno pretraživanje, primjerice, $O(n^3)$ u najgorem slučaju, ali ne i da je $\Theta(n^3)$. Ako je $f(n) = \Theta(g(n))$ onda je $f(n) = O(g(n))$ jer Θ predstavlja čvršće ograničenje od O.

Formalno [1], Θ-notacija definirana je ovako:

$$\Theta(g(n)) = \{\ f(n) : \text{postoje konstante } c_1, c_2 \text{ i } n_0 \text{ takve da je } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ za svaki } n \geq n_0\ \}.$$

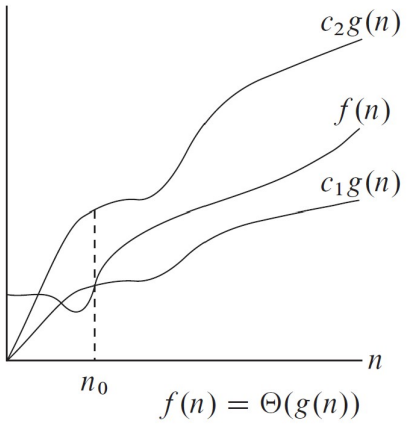
Sada možemo pokazati da je $\frac{1}{2}n^2 - \frac{1}{2}n = \Theta(n^2)$. Prema gornjoj definiciji, konstante c_1 i c_2 množimo s n^2, odnosno $g(n) = n^2$. To znači da moramo pronaći konstante c_1 i c_2 takve da je $c_1 n^2 \leq \frac{1}{2}n^2 - \frac{1}{2}n \leq c_2 n^2$ za svaki $n \geq n_0$. Ako sve podijelimo s n^2 dobit ćemo $f(n) = \frac{1}{2} - \frac{1}{2n}$. Prema tome, tražimo konstante c_1 i c_2 takve da je $c_1 \leq \frac{1}{2} - \frac{1}{2n} \leq c_2$. Ako riješimo lijevu stranu ove nejednakosti, možemo uzeti $c_1 = \frac{1}{4}$ za $n \geq 2$. Slično, za desnu stranu možemo uzeti $c_2 = \frac{1}{2}$ za $n \geq 1$. Ako uzmemo da je $n_0 = 2$ (jer ovo obuhvaća i $n \geq 1$) onda je $\frac{1}{2}n^2 - \frac{1}{2}n = \Theta(n^2)$.

Na sličan način možemo pokazati da je algoritam za linearno pretraživanje (2.4) koji treba kn koraka $\Theta(n)$. Ako, primjerice, zamijenimo k nekom konkretnom vrijednošću, recimo $k = 5$, želimo pokazati da je $5n = \Theta(n)$. Ovdje lako možemo odabrati konstante c_1 i c_2 takve da je $c_1 n \leq 5n \leq c_2 n$. Primjerice, $c_1 = 4$ i $c_2 = 6$ za $n_0 = 1$ zadovoljavaju ovu nejednakost.

U ovim smo primjerima mogli odabrati i drugačije konstante pod uvjetom da one zadovoljavaju zadanu nejednakost. Takve će konstante ovisiti o konkretnoj funkciji, ali da bi se moglo pokazati da je neka funkcija $f(n)$ u skupu $\Theta(g(n))$ važno je da one *postoje*.

Kao i s definicijama za O i Ω, definiciju Θ-notacije možemo koristiti i da bismo pokazali da neka funkcija nije u $\Theta(g(n))$. Primjerice, $9n \neq \Theta(n^2)$. Da bismo to pokazali možemo pretpostaviti da postoje n_0 i c_2 takvi da je $9n \leq c_2 n^2$ za svaki $n \geq n_0$. Ako ovo podijelimo s n^2 dobit ćemo $\frac{9}{n} \leq c_2$, odnosno $n \leq \frac{9}{c_2}$. S obzirom da nas zanima arbitrarno veliki n, ova nejednakost ne može biti ispravna jer je $\frac{9}{c_2}$ konstanta. Slično, $4n^3 \neq \Theta(n^2)$. Ovo možemo pokazati na isti način tako da pretpostavimo da postoji n_0 i c_2 takav da je $4n^3 \leq c_2 n^2$ za svaki $n \geq n_0$. Ako sve podijelimo s n^2 dobit ćemo $n \leq \frac{c_2}{4}$, što ne važi za arbitrarno veliki n jer je $\frac{c_2}{4}$ konstanta.

Mnogi algoritmi imaju vremensku složenost koja proizlazi iz nekakvog polinoma kao što je $\frac{1}{2}n^2 - \frac{1}{2}n$ ili kn. Pravilo glasi da za svaki polinom $p(n) = \sum_{i=0}^{d} a_i n^i$, gdje su a_i konstante i $a_d > 0$, vrijedi $p(n) = \Theta(n^d)$. Iz ovoga proizlazi da se konstantne funkcije mogu izraziti kao $\Theta(n^0)$ ili $\Theta(1)$. Prema tome, vremensku složenost ograničenu pozitivnim konstantama odozgo i odozdo možemo označavati s $\Theta(1)$.



Slika 2.5: Odnos $f(n)$ i $g(n)$ kod Θ-notacije [1].

Za neki skup funkcija $\Theta(g(n))$ kaže se da je $g(n)$ *asimptotski tijesno ograničenje* (engl. *asymptotically tight bound*) [1]. Slika 2.5 ilustrira odnos funkcija $f(n)$ i $g(n)$ gdje je $f(n) = \Theta(g(n))$. Na toj slici vidimo da za sve vrijednosti za koje je $n \geq n_0$, vrijednosti od $f(n)$ nalaze se između (ili su jednake) $c_1g(n)$ i $c_2g(n)$. Funkcija $c_1g(n)$ predstavlja donje ograničenje, a funkcija $c_2g(n)$ gornje. Nadalje, vrijedi sljedeći teorem [1]:

Teorem 1 *Za svake dvije funkcije $f(n)$ i $g(n)$, $f(n) = \Theta(g(n))$ ako i samo ako $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$.*

Prema ovom teoremu, ako je vremenska složenost nekog algoritma $\Theta(f(n))$, ona je ujedno $O(f(n))$ i $\Omega(f(n))$ i obratno. Vremenska složenost algoritma za linearno pretraživanje je $\Theta(n)$ jer mu je vremenska složenost u najgorem slučaju $O(n)$ i $\Omega(n)$.

2.1.3.4 Upotreba asimptotske notacije

Za analizu algoritama upotreba simbola O, Ω i Θ nije nužna jer performanse algoritma možemo opisati i kroz broj koraka u odnosu na veličinu ulaznih podataka, kao što smo učinili s algoritmima za linearno pretraživanje i sortiranje. Međutim, ako se bavimo asimptotskom analizom algoritama, ova nam je notacija izuzetno korisna jer omogućava izražavanje vremenske složenosti algoritama bez navođenja detaljnog opisa broja potrebnih koraka za njihovo izvršavanje. U ovom dijelu objašnjene su tipične upotrebe asimptotske notacije.

Kod mnogih algoritama, poput opisanih u ovom udžbeniku, najčešće nas zanima njihova vremenska složenost u najgorem, a ponekad i u najboljem slučaju. Prvo što treba istaknuti je da najgori slučaj *nije* isto što i asimptotsko ograničenje odozgo kao što najbolji slučaj *nije* isto što i asimptotsko ograničenje odozdo. Ne treba poistovjećivati slučaj koji analiziramo s asimptotskim ograničenjima. Kao prvo, kakav god slučaj analizirali, imat će gornje i donje ograničenje koje može, ali i ne mora biti isto. Kao drugo, ne postoje samo najgori, najbolji i prosječni slučaj. Primjerice, za neki algoritam može nas zanimati njegova vremenska složenost za nekakav specifičan skup ulaznih podataka koji ne moraju predstavljati jedan od ova tri slučaja. U takvom slučaju također će nas zanimati jedno ili oba ograničenja. Kada govorimo o specifičnim asimptotskim ograničenjima najčešće

ćemo istaknuti specifičan slučaj za koji analiziramo algoritam (kao što je, primjerice, najbolji slučaj). Iz tog razloga pisat ćemo, primjerice, *algoritam P ima vremensku složenost* $O(n^2)$ ***u najboljem (ili najgorem) slučaju*** tamo gdje se to ne podrazumijeva.

Kao primjer opet možemo uzeti Algoritam 2.4 za linearno pretraživanje. U najgorem slučaju element koji tražimo ne postoji u nizu. Gornja granica tog slučaja je n jer moramo proći sve elemente niza. Prema tome, sasvim je ispravno reći da je vremenska složenost ovog algoritma u najgorem slučaju $O(n)$. Međutim, što je s donjom granicom najgoreg slučaja? Ona je za ovaj algoritam ista kao i gornja granica, odnosno n, pa je zato također ispravno reći da je vremenska složenost ovog algoritma $\Omega(n)$ u najgorem slučaju. Drugim riječima, ono što smo rekli s $O(n)$ je da u najgorem slučaju ograničenje odozgo neće rasti brže od n, a s $\Omega(n)$ da neće rasti sporije od n. Obje su tvrdnje ispravne, odnosno vremenska složenost ovog algoritma je $O(n)$, $\Omega(n)$ prema teoremu **??** i $\Theta(n)$, ali se odnose na različita ograničenja rasta funckije. Ako se poslužimo jednim primjerom iz stvarnog života, možemo reći da nismo vozili brže od 100 km na sat (gornje ograničenje) ili da smo vozili brže od 30 km na sat (donje ograničenje).

Ako se sada nakratko vratimo na Algoritam 2.6 (SORTIRANJE) možemo lako izvesti njegovu vremensku složenost. Kao što smo vidjeli, taj se algoritam sastoji od dva općenita koraka. Prvi je provjera je li ulazni niz već sortiran i drugi koji sortira taj niz. Što bi ovdje bio najgori slučaj? Da bi se utvrdilo je li niz sortiran potrebno je n koraka. Ako niz nije sortiran algoritam mora ići na drugi korak koji će taj niz sortirati. Vidjeli smo da taj drugi korak ima kvadratnu vremensku složenost, odnosno da mu treba $\frac{1}{2}n^2 - \frac{1}{2}n$ koraka. Prema tome, ukupan broj koraka je $n + (\frac{1}{2}n^2 - \frac{1}{2}n)$. Upotrebom asimptotske notacije za gornju granicu ova funkcija je $O(n^2)$. To je vremenska složenost Algoritma 2.6 u najgorem slučaju. U najboljem slučaju, tj. kada je ulazni niz već sortiran, gornja granica je n, odnosno u asimptotskoj notaciji $O(n)$.

Ovo smo mogli napisati i preciznije. S obzirom da su gornja i donja granica Algoritma 2.7 (provjera je li niz sortiran) jednake, prema Teoremu **??** možemo napisati da je njegova vremenska složenost $\Theta(n)$. Slično, Algoritam 2.6 ima vremensku složenost $\Theta(n)$ u najboljem slučaju i $\Theta(n^2)$ u najgorem.

2.1.4 Grafički prikaz vremenske složenosti

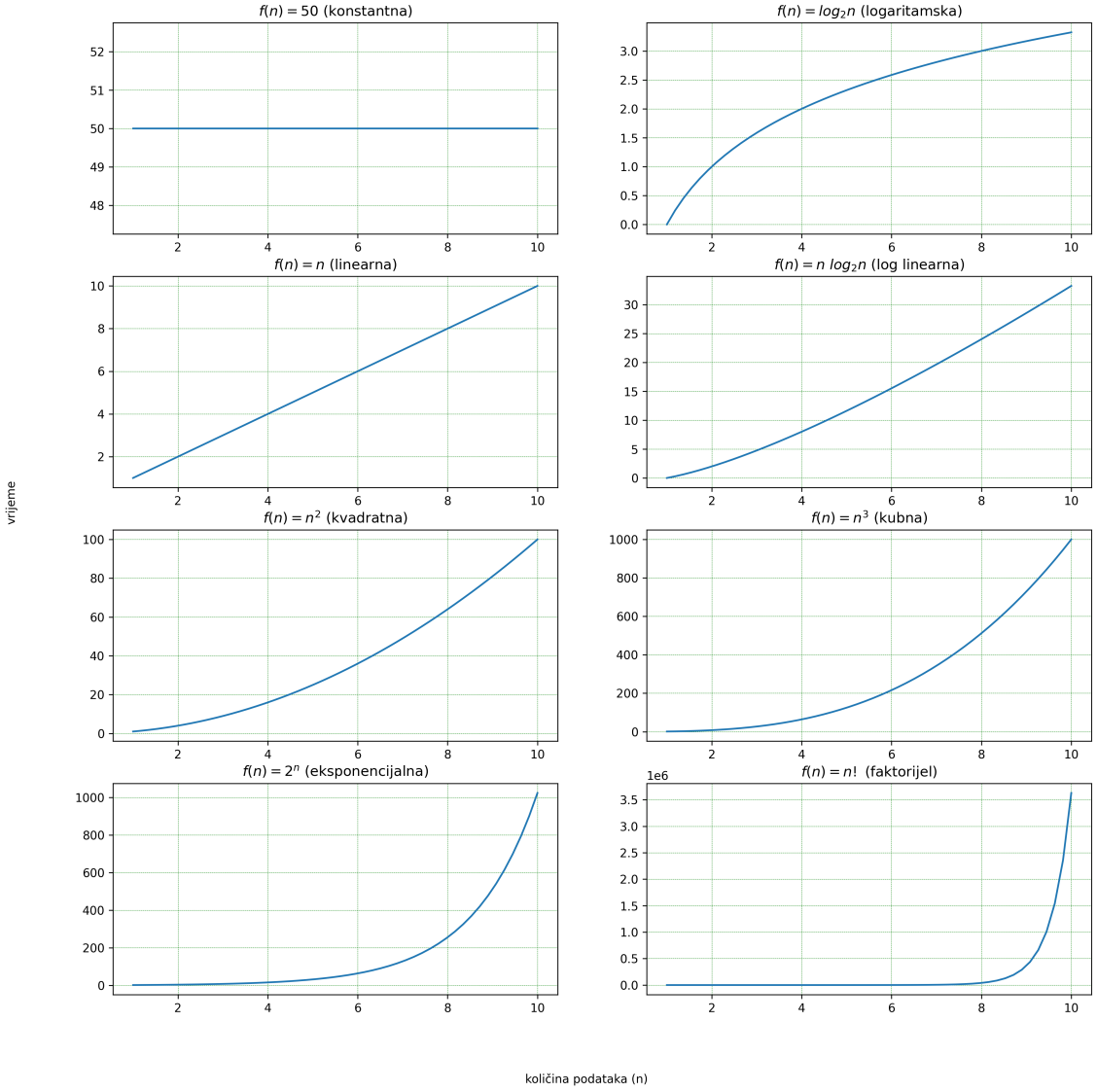
Vremensku složenost korisno je prikazati grafički kako bismo jasnije vidjeli brzinu rasta vremena u odnosu na porast veličine ulaza. Vremensku složenost Algoritma 2.5 možemo prikazati kao na Slici 2.7. Slika 2.8 prikazuje oblik funkcije 2.1 i konkretna vremena izvršavanja jedne implementacije ovog algoritma. Vidimo da se ta vremena prilično dobro poklapaju s oblikom grafa ove funkcije što potvrđuje njegovu kvadratnu vremensku složenost.

2.1.4.1 Neke tipične vremenske složenosti

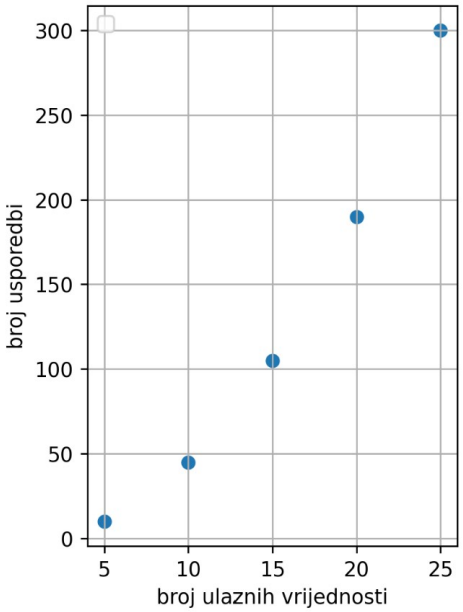
U literaturi se često obrađuju "standardni" algoritmi sortiranja i pretraživanja čija vremenska složenost pripada jednoj od onih prikazanih na Slici 2.6. Na toj su slici vremenske

složenosti raspoređene prema brzini rasta, od najsporije prema najbržoj:

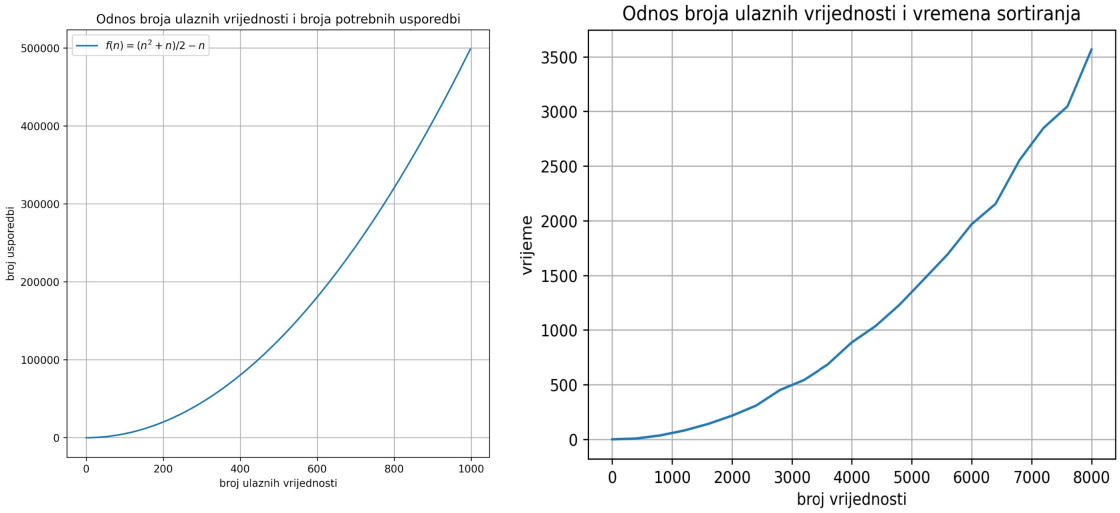
- $\Theta(1)$ ili $O(1)$ (konstantna): Ova vremenska složenost ne ovisi o veličini ulaza. Primjerice, ako se jedan skup naredbi uvijek izvršava isti broj puta, izvršava se u konstantnom vremenu.
- $\log_2 n$ (logaritamska): Kod ove vremenske složenosti algoritmu treba više vremena kako n raste. Ova se vremenska složenost često pojavljuje kod algoritama koji neki veći problem rješavaju rastavljajući ga na manje podprobleme, čime smanjuju veličinu problema za neki konstantni faktor. Iako baza logaritma donekle ima utjecaja, razlike nisu velike. Kada je $n = 1000$, $log_{10} n = 3$ i $\log_2 n \approx 10$. Kada je $n = 1.000,000$, $\log_2 n$ je dvostruko veći. Kada se n udvostruči $\log_2 n$ se poveća za neku konstantu, ali $\log_2 n$ se ne povećava dvostruko sve dok dok se n ne poveća na n^2.
- n (linearna): Kod ove vremenske složenosti obično algoritam utroši podjednako vrijeme za svaki ulazni element. Kada se n udvostruči, udvostruči se i vrijeme.
- $n \log_2 n$ (log-linearna ili linearitamska): Ova se vremenska složenost često javlja kod algoritama koji problem rastavljaju na manje podprobleme koje riješe zasebno i dobivena rješenja kombiniraju u globalno riješenje. Kada se n udvostruči, vrijeme se više nego udvostruči (ali ne znatno više).
- n^2 (kvadratna): Ova vremenska složenost nije praktična za veće količine ulaznih podataka. Javlja se, primjerice, kod algoritama koji obrađuju sve moguće parove ulaznih elemenata. Kada je $n = 1000$, vrijeme je $1.000,000$, a kada se n udvostruči vrijeme se učetverostruči.
- n^3 (kubna): Kao i kvadratna složenost, ova je vremenska složenost praktična samo za male količine ulaznih podataka. Kada se n udvostruči vrijeme se poveća osam puta.
- 2^n (eksponencijalna): Neki algoritmi koji rade "grubom silom" (engl. *brute force*) imaju ovakvu vremensku složenost. Zbog vrlo visoke stope rasta ovakve funkcije, takvi algoritmi obično nisu primjenjivi u praksi. Kada je $n = 20$, vrijeme je $1.000,000$, a kada se n udvostruči vrijeme se poveća na kvadrat.
- $n!$ (faktorijelna): Ova vremenska složenost ima izuzetno visok stupanj rasta i zbog toga nije praktična za upotrebu. Jedan od primjera algoritma s ovom vremenskom složenošću je za generiranje svih permutacija skupa od n elemenata.



Slika 2.6: Oblici grafova funkcija nekoliko tipičnih vremenskih složenosti.



Slika 2.7: Prikaz odnosa broja usporedbi i broja ulaznih vrijednosti Algoritma 2.5.



Slika 2.8: Lijevo: oblik funkcije 2.1. Desno: vrijeme izvršavanja jedne implementacije Algoritma 2.5.

POGLAVLJE 3

Rekurzija

S obzirom da su funkcije osnovni elementi gotovo svakog programa, one kao takve određuju zakonitosti o načinu odvijanja računalnog procesa na razini tih funkcija, odnosno kako se jedna faza tog procesa nadograđuje na prethodnu. Rekurzija je jedna od temeljnih tehnika programiranja koja je korisna za rješavanje mnogih vrsta računalnih problema. Funkcija koja u svojoj definiciji poziva samu sebe naziva se *rekurzivnom funkcijom*. Rekurziju možemo promatrati kao jedan oblik iteracije, odnosno ponavljanja, ali pri tome moramo uzeti u obzir neke karakteristike takvog pristupa poput učinkovitosti i potrošnje memorije [12, 13]. U ovom dijelu proučit ćemo osnovna načela rekurzije i vidjeti neke primjere u kojima je ona primjenjiva.

3.1 Rekurzivne funkcije

Osnovni oblik rekurzivne funkcije prikazan je u Algoritmu 3.1.

Algoritam 3.1 Oblik rekurzivne funkcije.

```
1: function REK(n)
2:     ...
3:     ... REK(n) ...
4:     ...
5: end function
```

Rekurzivna funkcija u osnovnom obliku ona je koja poziva samu sebe s jednog ili više mjesta unutar svoje definicije. Rekurzija, međutim, ne mora biti izravna kao u gornjem primjeru, kako je prikazano u Algoritmu 3.2. U tom primjeru funkcija *f* poziva funkciju *g*, a funkcija *g* poziva opet funkciju *f*. Ovakve se funkcije nazivaju *uzajamno rekurzivnim* (engl. *mutually recursive functions*) [14].

Svaki postupak koji se može definirati iteracijom može se također definirati rekurzijom i obratno. Oba načina imaju prednosti i nedostatke, a koji od njih treba koristiti ovisi, kao

Algoritam 3.2 Oblik neizravne rekurzije.

```
1: function F(x)
2:     ...
3:     ... G(n) ...
4:     ...
5: end function
6:
7: function G(x)
8:     ...
9:     ... F(m) ...
10:    ...
11: end function
```

uvijek, o prirodi problema koji rješavamo. Za usporedbu iteracije i rekurzije proučit ćemo jednostavnu matematičku funkciju koja se zove *faktorijel* i daje umnožak svih (cijelih) brojeva u intervalu $1..n$ gdje je n zadan kao parametar. Ova se operacija u matematici označava s $n!$ i definira na sljedeći način:

$$n! = n * (n-1) * (n-2) * \ldots 3 * 2 * 1.$$

Na primjer, $5! = 4 * 3 * 2 * 1 = 24$. Iterativna inačica funkcije, koju ćemo zvati FAKT-ITER, može se napisati kao u Algoritmu 3.3.

Algoritam 3.3 Iterativno izračunavanje faktorijela.

```
1: function FAKT-ITER(n)
2:     rezultat ← 1
3:     while n > 0 do
4:         rezultat ← rezultat * n
5:         n ← n − 1
6:     end while
7:     return rezultat
8: end function
```

Ako u svakom koraku zamijenimo varijable $rezultat$ i n s njihovom vrijednošću u tom koraku, možemo jasno vidjeti kako smo došli do rezultata:

```
1. rezultat = 1 (početna vrijednost)
2. rezultat = 1 * 4, n = 4
3. rezultat = 4 * 3, n = 3
4. rezultat = 12 * 2, n = 2
5. rezultat = 24 * 1, n = 1
return 24
```

Vidimo da se ovaj postupak odvija tako da je u svakom koraku vrijednost varijabli $rezultat$ i n bliža nekoj konačnoj vrijednosti. Isto tako, uvjet petlje $n > 0$ osigurava njegovo prekidanje kada n dosegne vrijednost 0.

Faktorijel se često definira i na ovaj način:

$$n! = \begin{cases} 1, & \text{za } n = 0 \\ n(n-1)!, & \text{za } n > 0 \end{cases}$$

U drugom retku vidimo da je u ovoj definiciji faktorijel definiran pomoću faktorijela, ali za manju vrijednost, odnosno za $n - 1$. Ovo se naziva *rekurzivnom definicijom* [14] i za nju možemo napisati rekurzivnu inačicu ove funkcije koju ćemo zvati FAKT-REK i prikazana je u Algoritmu 3.4.

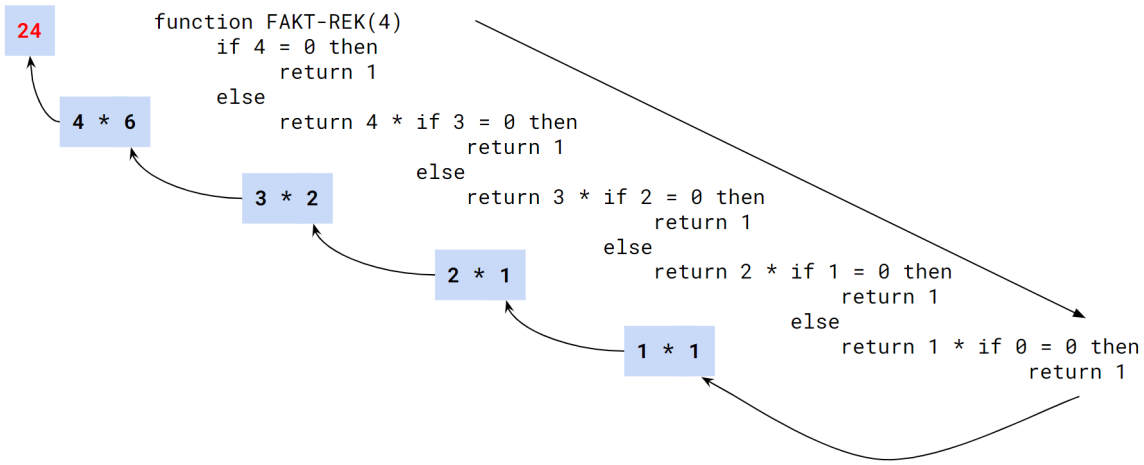
Algoritam 3.4 Rekurzivno izračunavanje faktorijela.

```
1: function FAKT-REK(n)
2:     if n = 0 then
3:         return 1
4:     else
5:         return n * FAKT-REK(n − 1)
6:     end if
7: end function
```

Vidimo da u ovoj funkciji ne koristimo petlju nego pozivamo funkciju FAKT-REK unutar te iste funkcije! Ovdje se odmah možemo pitati čime množimo n ako svaki put idemo na početak ove funkcije baš kada nam treba neki broj. Rekurzivni poziv funkcije odvija se na potpuno isti način kao i svaki drugi poziv, odnosno ne postoji nikakav posebni mehanizam koji je ugrađen u interpreter ili prevoditelj programskog jezika koji bi se koristio upravo za rekurzivne funkcije. Ovaj rekurzivni poziv FAKT-REK$(n-1)$ zapravo daje umnožak brojeva u intervalu $1..n-1$ koji onda pomnožimo s n. Drugim riječima, konačni rezultat dobijemo tako da svaki broj k pomnožimo umnoškom brojeva na manjem intervalu, $1..k-1$. Kao što smo iterativnu inačicu ove funkcije prikazali kao jedan niz naredbi koji se ponavlja više puta, njenu rekurzivnu inačicu prikazujemo kao jedan izraz koji se sastoji od više podizraza. Funkciju FAKT-REK možemo predočiti tako da na svako mjesto gdje se nalazi rekurzivan poziv FAKT-REK$(n-1)$ smjestimo cijelu definiciju ove funkcije, s tim da varijablu n zamijenimo njenom trenutnom vrijednošću kako je pokazano na Slici 3.1.
Ako izdvojimo samo izraz naredbe $return$ dobit ćemo sljedeći niz koraka:

```
1. return 4 * FAKT-REK(4 - 1)
2. return 4 * 3 * FAKT-REK(3 - 1)
3. return 4 * 3 * 2 * FAKT-REK(2 - 1)
4. return 4 * 3 * 2 * 1 * FAKT-REK(1 - 1)
5. return 4 * 3 * 2 * 1 * 1
6. return 4 * 3 * 2 * 1
7. return 4 * 3 * 2
8. return 4 * 6
9. return 24
```

U svakom koraku imamo isti izraz: $n *$ FAKT-REK$(n-1)$. U prvom koraku, prema tome, rješavamo početni izraz FAKT-REK(4) čiji je rezultat $4 *$ FAKT-REK$(4-1)$. U sljedećem koraku rješavamo izraz iz prethodnog koraka, odnosno FAKT-REK(3), čiji je rezultat $3 *$FAKT-REK$(3-$



Slika 3.1: Rekurzivno računanje faktorijela.

1). Ovdje ne možemo zaboraviti na vrijednost 4 iz prethodnog koraka jer tom vrijednošću moramo pomnožiti rezultat ovog idućeg koraka, to jest rezultat poziva FAKT-REK$(4-1)$, što je $3 *$ FAKT-REK$(3-1)$, i tako dalje. Množenje brojem 4 i drugim brojevima koji slijede do koraka 5 je odgođeno. Tek kada se dosegne korak 5 mogu se početi množiti vrijednosti jer su tek tada sve one na raspolaganju. Izraz `if n = 0 then return 1` osigurava prekidanje ovog procesa kada n dosegne vrijednost 0. To se vidi nakon koraka 4, gdje više ne pozivamo funkciju FAKT-REK sa sljedećim manjim argumentom jer smo došli do "kraja", to jest došli smo do FAKT-REK(0) za koji znamo da je 1, pa prema tome nema potrebe dalje pozivati ovu funkciju. Nakon toga, u koraku 5 imamo potpun izraz množenja koji se postupno izračunava do konačnog rezultata. Koracima 5 do 8 vraćamo se iz svih ovih prethodnih poziva. Korakom 5 vraćamo se iz poziva u koraku 4, korakom 6 iz poziva u koraku 3, korakom 7 iz poziva u koraku 2, i na kraju korakom 8 vraćamo se iz poziva u prvom koraku čime dobivamo rezultat ove funkcije.

Sada možemo vidjeti razliku između iterativnog i rekurzivnog procesa. Rad iterativne funkcije temeljio se na postupnim promjenama vrijednosti varijabli $rezultat$ i n. Varijablu $rezultat$ povećavali smo u svakom koraku tako da smo je pomnožili varijablom n, a varijabla n nam je istovremeno služila kao brojač koraka. Kaže se da su $rezultat$ i n varijable stanja ovog procesa. To znači da je u svakom koraku njegov napredak bio određen tim dvjema varijablama. Iterativni je proces onaj čije je stanje u svakom koraku određeno varijablama stanja i pravilom kako se mijenjaju vrijednosti tih varijabli iz jedne iteracije u iduću. Možemo reći da svaki put nakon što smo varijablu $rezultat$ postavili na $rezultat * n$ i varijablu n na $n-1$ u našem primjeru, ovaj je proces prešao na sljedeći korak. Rekurzivna funkcija izvršavala se tako da se svaka vrijednost i operacija (kao što je množenje) morala pamtiti da bismo na kraju izračunali konačni rezultat. Operacije množenja ovdje su bile odgođene dok nismo dobili potpuni izraz. Ovo odgađanje operacija nije ništa drugo nego određivanje vrijednosti argumenata koje mora prethoditi pozivu funkcije. S obzirom da je sâm argument rezultat te iste funkcije, dobivamo ovaj tipični kružni proces koji eventualno završava nekom konkretnom vrijednošću za taj argument (kao što je u našem primjeru broj 1 kada je n došao do 0). To se vidi na gornjem nizu koraka kao izraz koji se širi, nakon čega slijedi skupljanje (kao posljedica vraćanja iz poziva funkcije). Takav je

oblik tipičan za rekurzivne procese. Proces koji se temelji na takvom jednom nizu poziva funkcija zove se *rekurzivan proces* [14].

Rekurzivne funkcije mogu biti kompleksnije, gdje se ista funkcija može koristiti kao argument na više mjesta unutar jednog izraza. Za primjer proučit ćemo još jednu matematičku operaciju koja izračunava vrijednosti Fibonaccijevog niza [15] koji izgleda ovako: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...$ Ovaj niz počinje brojevima 0 i 1, a svaki idući broj zbroj je prethodna dva. Prema tome, funkcija $fib(n)$ kojom se izračunava vrijednost ovog niza na položaju n (gdje je 0 početni položaj) može se matematički definirati ovako:

$$fib(n) = \begin{cases} 0, & \text{za } n = 0 \\ 1, & \text{za } n = 1 \\ fib(n-1) + fib(n-2), & \text{za } n > 1 \end{cases}$$

U Algoritmu 3.5 ova je definicija napisana u pseudokodu.

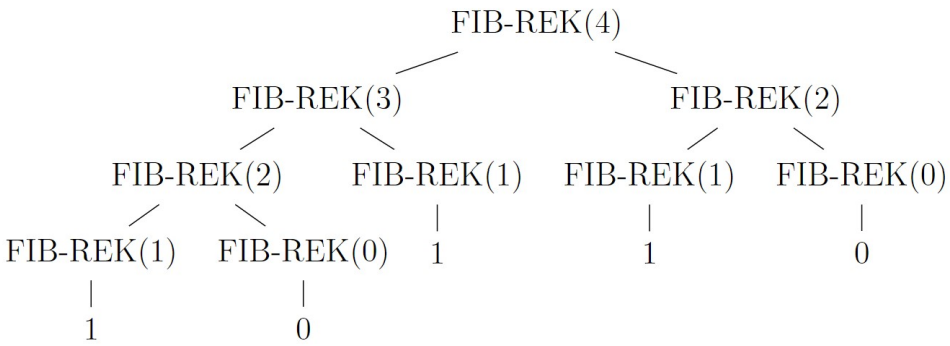
Algoritam 3.5 Rekurzivno izračunavanje *n*-tog elementa Fibonaccijevog niza.

```
1: function FIB-REK(n)
2:     if n = 0 or n = 1 then
3:         return n
4:     else
5:         return FIB-REK(n − 1) + FIB-REK(n − 2)
6:     end if
7: end function
```

Ovaj proces sada možemo prikazati na isti način kao u prethodnom primjeru za izraz FIB-REK(4):

```
1. return FIB-REK(4 - 1) + FIB-REK(4 - 2)
2. return FIB-REK(3 - 1) + FIB-REK(3 - 2) + FIB-REK(4 - 2)
3. return FIB-REK(2 - 1) + FIB-REK(2 - 2) + FIB-REK(3 - 2) + FIB-REK(4 - 2)
4. return 1 + FIB-REK(2 - 2) + FIB-REK(3 - 2) + FIB-REK(4 - 2)
5. return 1 + 0 + FIB-REK(3 - 2) + FIB-REK(4 - 2)
6. return 1 + 0 + 1 + FIB-REK(4 - 2)
7. return 1 + 0 + 1 + FIB-REK(2 - 1) + FIB-REK(2 - 2)
8. return 1 + 0 + 1 + 1 + FIB-REK(2 - 2)
9. return 1 + 0 + 1 + 1 + 0
10. return 1 + 1 + 1
11. return 2 + 1
12. return 3
```

Grananje izraza ovog procesa dobivamo tako da svaki poziv FIB-REK zamijenimo dvama novim pozivima jer se rezultat ove funkcije sastoji od dva rekurzivna poziva (za $n > 1$). Zbog toga je oblik izraza koje generira ovaj proces takav da se oni šire i skupljaju više puta. Takav oblik izraza generiranih nekom rekurzivnom funkcijom zove se *rekurzivno stablo* [14] jer ima oblik stabla. Takvo stablo za izraz FIB-REK(4) prikazano je na Slici 3.2 na



Slika 3.2: Rekurzivno stablo izraza FIB-REK(4).

kojoj vidimo da se svaki poziv ove funkcije grana na dvije strane jer u ovoj funkciji imamo dva rekurzivna poziva povezana operatorom "+". Na krajevima ovog stabla promatranog odozgo prema dolje nalaze se brojevi ćiji zbroj daje rezultat ove funkcije.

U rekurzivnom stablu ovog procesa možemo vidjeti da je dio tog stabla za $n = 2$ ponovljen dva puta. Prvi dio bio je formiran pozivom FIB$(n-2)$ za $n = 4$, a drugi pozivom FIB$(n-1)$ za $n = 3$, što je lako vidljivo na slici. Iz ovoga možemo zaključiti da rekurzivni procesi mogu biti neučinkoviti u nekim slučajevima jer se tada, kao u ovom slučaju, vrijednost za jedan te isti izraz izračunava više puta.

Iterativna inačica za Fibonaccijev niz, prikazana u Algoritmu 3.6, po strukturi je slična funkciji FAKT-ITER.

Algoritam 3.6 Iterativno izračunavanje *n*-tog elementa Fibonaccijevog niza.

```
1: function FIB-ITER(n)
2:     prethodni ← 0
3:     iduci ← 1
4:     while n > 0 do
5:         t ← prethodni + iduci
6:         prethodni ← iduci
7:         iduci ← t
8:         n ← n − 1
9:     end while
10:    return prethodni
11: end function
```

Kao što je bio slučaj kod FAKT-ITER, proces za FIB-ITER(4) izgleda puno jednostavnije od rekurzivnog procesa:

```
1. prethodni = 0, idući = 1 (početne vrijednosti)
2. t = 1, prethodni = 1, idući = 1, n = 4
3. t = 2, prethodni = 1, idući = 2, n = 3
4. t = 3, prethodni = 2, idući = 3, n = 2
```

```
5. t = 5, prethodni = 3, idući = 5, n = 1
return prethodni
```

Ovdje vidimo kako se varijabla $prethodni$ postupno približavala rezultatu, a varijabla $iduci$ je sadržavala vrijednost koja dolazi nakon one u varijabli $prethodni$. Ovaj proces, kao i onaj za FAKT-ITER, nema nikakvog odgođenog računanja i svaki korak je isključivo određen vrijednostima varijabli t, $prethodni$, $idui$ i n.
Grananje koje je tipično za rekurzivne procese ima jednu, u nekim situacijama negativnu posljedicu, a to je potreba za većom količinom memorije u odnosu na iterativne procese. U primjeru funkcija FAKT-REK i FIB-REK vidjeli smo da izraz, to jest broj vrijednosti koje treba pamtiti, sve više raste, pa su i potrebe za memorijom tijekom izvršavanja tih funkcija sve veće. To nije bio slučaj kod njihovih iterativnih inačica jer smo tamo koristili varijable stanja u kojima smo držali sve privremene vrijednosti dok nismo došli do konačnog rezultata. Kod rekurzivne inačice, što je n bio veći trebalo je više memorije, dok je kod iterativne inačice potreba za memorijom bila konstantna, to jest nije ovisila o n. U primjeru Fibonaccijevog niza, funkcija FIB-REK imala je, osim memorije, jedan dodatni problem, a to je da je neke izraze izračunavala više puta. To bi kod većih vrijednosti za n rezultiralo značajnim rastom vremena potrebnog da pronađe rezultat. Iz ovoga možemo zaključiti da su iterativna rješenja u nekim slučajevima učinkovitija od rekurzivnih sa stajališta veličine potrebne memorije i brzine izvršavanja. To, međutim, nije uvijek slučaj, odnosno njihova učinkovitost može biti i podjednaka.

Za kraj je ostalo još jedno pitanje vezano za ove dvije vrste procesa: Kada neki proces, odnosno funkciju treba definirati iterativno, a kada rekurzivno? Odgovor najčešće ovisi o tome koja je najprirodnija definicija problema koji rješavamo; međutim, ni ta odluka nije uvijek jednostavna. U našim primjerima s izračunavanjem faktorijela i Fibonaccijevog niza lako smo mogli definirati oba problema na iterativan ili rekurzivan način. U takvim slučajevima možemo uzeti u obzir učinkovitost izvršavanja tako da bismo se u oba slučaja opredijelili za iterativnu inačicu. S druge strane, kod nekih problema rekurzija je prirodno i prihvatljivo riješenje.

Rekurzivno pretraživanje niza. Pretpostavimo da želimo napisati program koji traži neku zadanu vrijednost u nizu, a da sam taj niz može imati druge nizove kao elemente [16]. Za početak možemo definirati funkciju POSTOJI prikazanu u Algoritmu 3.7 koja u nizu traži zadanu vrijednost, ne uzimajući u obzir činjenicu da elementi tog niza mogu biti drugi nizovi. Ova funkcija radi ispravno ako se element koji tražimo nalazi u glavnom nizu, to jest ne nalazi se u nekom unutrašnjem nizu. Primjerice, za ulazni niz kao što je $[a, [b, c, [d]]]$ ova funkcija ne pronalazi element c jer on se nalazi u jednom od podnizova glavnog niza. Razlog je taj da c nije isto što i $[c]$ – u prvom slučaju imamo tekstualnu vrijednost, a u drugom (pod)niz. Budući da su to dvije različite vrste podataka ne možemo ih uspoređivati, to jest ne mogu biti jednaki.

Da bi ova funkcija radila s unutrašnjim nizovima trebamo joj omogućiti da nastavi tražiti unutar tog unutrašnjeg niza ako dođe do takvog elementa, počevši od prvog njegovog elementa. Takav program možemo napisati kako je prikazano u Algoritmu 3.8. Dio unutar petlje izvršava se gotovo identično prethodnoj inačici, ali umjesto operatora "=" koristimo

Algoritam 3.7 Jednostavna provjera pripadnosti elementa nizu.

Parametri: v - vrijednost koju tražimo; niz - niz koji pretražujemo.

```
1: function POSTOJI(v, niz)
2:     for all e in niz do
3:         if e = v then
4:             return true
5:         end if
6:     end for
7:     return false                          ▷ traženi element nije pronađen
8: end function
```

funkciju JEDNAKO. Ova funkcija prvo provjeri je li element koji uspoređujemo sa zadanom vrijednošću u stvari i sâm niz. Ako nije, usporedimo ga s tom vrijednošću kao što smo radili u prethodnoj inačici. Međutim, ako jest, onda funkcija JEDNAKO poziva funkciju POSTOJI s tim nizom kao ulaznim nizom. Prema tome, pretraživanje unutrašnjih nizova također obavlja funkcija POSTOJI čiji rezultat ukazuje na to postoji li traženi element u unutrašnjem nizu. Taj rezultat upotrebljava ta ista funkcija dok pretražuje vanjski niz. Ovdje se također radi o rekurziji, samo što funkcija ne poziva samu sebe izravno nego preko neke druge funkcije. U ovom primjeru funkcija POSTOJI poziva funkciju JEDNAKO koja opet poziva funkciju POSTOJI pa su stoga te dvije funkcije uzajamno rekurzivne.

Algoritam 3.8 Provjera pripadnosti elementa nizu uzajamnom rekurzijom.

```
 1: function JEDNAKO(e, x)
 2:     if e je niz then
 3:         return POSTOJI(x, e)
 4:     else
 5:         return e = x
 6:     end if
 7: end function

 8: function POSTOJI(v, niz)
 9:     for all x in niz do
10:         if JEDNAKO(v, x) then
11:             return true
12:         end if
13:     end for
14:     return false
15: end function
```

Na ovaj način možemo naći bilo koji element neovisno o tome koliko se duboko nalazi u unutrašnjim nizovima. Funkciju POSTOJI mogli smo napisati i kako je pokazano u Algoritmu 3.9. Ova inačica izvršava se na isti način kao i ona u Algoritmu 3.8, ali koristi izravnu rekurziju. Prethodna je inačica jasnije napisana jer su prolaz kroz niz i uspoređivanje (funkcijom JEDNAKO) odvojeni.

Algoritam 3.9 Provjera pripadnosti elementa nizu izravnom rekurzijom.

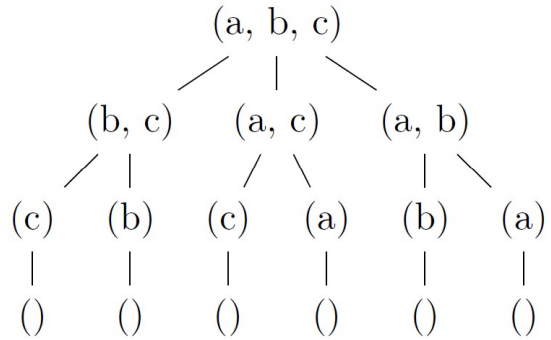
```
1: function POSTOJI(v, niz)
2:     for all e in niz do
3:         if e je niz then
4:             if POSTOJI(v, e) then
5:                 return true
6:             end if
7:         else
8:             if e = v then
9:                 return true
10:            end if
11:        end if
12:    end for
13:    return false
14: end function
```

Ako pogledamo neki niz koji sadrži podnizove, svaki od tih nizova možemo pretraživati na isti način kao i vanjski niz, to jest koristeći istu funkciju. Ulazni niz za tu funkciju upravo je taj podniz. U ovakvim slučajevima upotreba rekurzije izrazito je korisna tehnika programiranja jer nam omogućava rad s podacima koji su organizirani po nekom određenom načelu u jednu strukturu koja se sastoji od više takvih istih, ali manjih podstruktura. Primjerice, niz se sastoji od elemenata koji mogu biti drugi nizovi. Za svaki od tih nizova također važi isto pravilio, odnosno i oni se sastoje od elemenata koji mogu biti drugi nizovi. Ovdje treba uočiti da smo za definiciju niza koristili pojam niza – jedan niz može sadržavati drugi niz. Ova je sama definicija stoga rekurzivna pa smo i problem pretraživanja niza riješili koristeći rekurziju.

Kao što je već rečeno, u nekim situacijama rekurzija nije najučinkovitije rješenje. Čak i ako je problem definiran rekurzivno. potrebno je imati uvid u svojstva takvih procesa (kao što smo učinili kod funkcije FIB-REK) prije nego što se odlučimo na iteraciju ili rekurziju. Isto tako, može postojati i više rekurzivnih rješenja od kojih neka mogu biti učinkovitija od drugih. Rekurzija nije manje ili više učinkovita od iteracije, ali funkciju može učiniti jasnijom, pogotovo ako izravno odgovara definiciji problema kao što je bio slučaj s pretraživanjem nizova. Nadalje, svaki se rekurzivan algoritam može napisati i bez rekurzije, odnosno iterativno, što znači da svaku rekurzivnu funkciju možemo konvertirati u iterativan oblik.

Primjer: kombinacije i permutacije

Kako bismo mogli generirati skup svih kombinacija elemenata nekog niza? Primjerice, za n-torku (a, b, c) želimo dobiti skup $\{(a, c), (c), (a, b, c), (b, c), (a, b), (b), (), (a)\}$ [17]. Broj kombinacija za n elemenata je 2^n. Ova nam činjenica može poslužiti za provjeru ispravnosti rezultata, ali ne i za njihovo generiranje. Kod ovakvih problema često nam može pomoći skica rekurzivnog stabla po kojoj kasnije možemo definirati algoritam te njegovu implementaciju. Za generiranje kombinacija krenut ćemo od stabla prikazanog



Slika 3.3: Stablo generiranja kombinacija. Svaki čvor sadrži jednu kombinaciju, iako se neke ponavljaju.

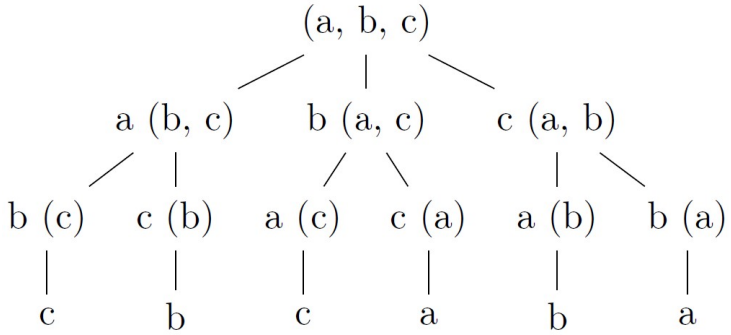
na Slici 3.3. U korijenu se nalazi početni niz. Na nižoj razini taj se niz grana u podnizove bez prvog, drugog i trećeg elementa. Svaki se od tih nizova opet grana na isti način kao i korijen. Očito je da bi za više elemenata stablo bilo dublje. Vidimo da ovo stablo sadrži sve kombinacije, samo nam treba sustavan način izdvajanja njegovih pojedinih dijelova. U načelu, za svaki niz izdvojimo segmente na indeksima $1, 2, ..., n$, a isto učinimo i za svaki od tih segmenata. Ako sada prođemo ovim stablom u dubinu dobit ćemo kombinacije $\{(a, b, c), (b, c), (c), (), (b), (), (a, c), (c), ...\}$. U implementaciji ovog postupka duplikate možemo lako eliminirati njihovim dodavanjem u skup. Ovaj postupak prikazan je u Algoritmu 3.10.

Algoritam 3.10 Sve kombinacije elemenata zadanog niza.

Parametri: niz - niz elemenata; $rezultat$ - skup svih kombinacija elemenata niza
Rezultat: skup svih kombinacija elemenata niza

```
1:  function KOMBINACIJE(niz, rezultat)                ▷ parametar rezultat je skup
2:      for i = 1..niz.veličina-niza do
3:          p ← niz[1..(i − 1)] + niz[(i + 1)..niz.veličina-niza]
4:          rezultat ← rezultat ∪ p
5:          if p.broj-elemenata > 0 then
6:              KOMBINACIJE(p, rezultat)
7:          end if
8:      end for
9:      return rezultat
10: end function
```

Na sličan način možemo generirati i permutacije. Za niz od n elemenata broj permutacija je $n!$. Za razliku od kombinacija, svaka permutacija sadrži isti broj elemenata kao u početnom nizu. I za ovaj se problem možemo poslužiti rekurzivnim stablom na Slici 3.3. Ovdje ćemo to stablo drugačije tumačiti. Permutacije možemo dobiti tako da za svaki čvor na istoj razini izdvojimo drugačiji element kao prvi. Za razliku od kombinacija koje se nalaze u čvorovima stabla, ovdje ćemo uzimati u obzir put od korijena do posljednjeg čvora. Primjerice, jedan takav put lijeva je strana stabla s čvorovima označenim $a(b, c)$, $b(c)$ i c. Takav obilazak ovog stabla dao bi permutaciju (a, b, c) gdje smo uzimali prvi (izdvojeni) element svakog čvora nakon korijena. Po povratku iz čvora c pokušali bismo uzeti



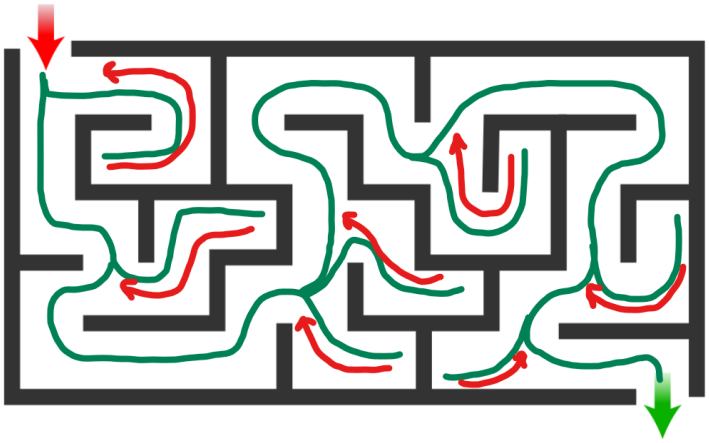
Slika 3.4: Stablo generiranja permutacija. U zagradi se nalazi ostatak elemenata iz prethodnika (roditelja) nakon što je izdvojen element koji se nalazi izvan zagrada. Izdvojeni elementi ulaze u permutaciju koja se dobiva od izdvojenih elemenata na putu od korijena do lista stabla.

desnu granu čvora označenog s $b(c)$. S obzirom da takva grana ne postoji, vraćamo se na prethodni čvor označen s $a(b, c)$. Sada uzimamo desnu granu tog čvora i ponavljamo postupak čime bismo generirali permutaciju (a, c, b). Povratkom u korijen nastavljamo na isti način čime bismo dobili permutacije (b, a, c) i (b, c, a). Ponavljanjem ovog postupka dobili bismo sve permutacije. Ovo je implementirano funkcijom PERMUTACIJE prikazanom u Algoritmu 3.11. Funkcija PERMUTACIJE na svakoj razini rekurzivnog poziva (koji odgovara razini rekurzivng stabla) uzima različiti element kao polazni (iz niza elemenata u kojem nema elementa koji je uzet na prethodnoj višoj razini) kao što je prikazano na Slici 3.4. Ta je slika ista kao i Slika 3.3 samo što su elementi koji ulaze u permutaciju izdvojeni tako da se za svaki obilazak stabla jasno vidi permutacija koju on generira.

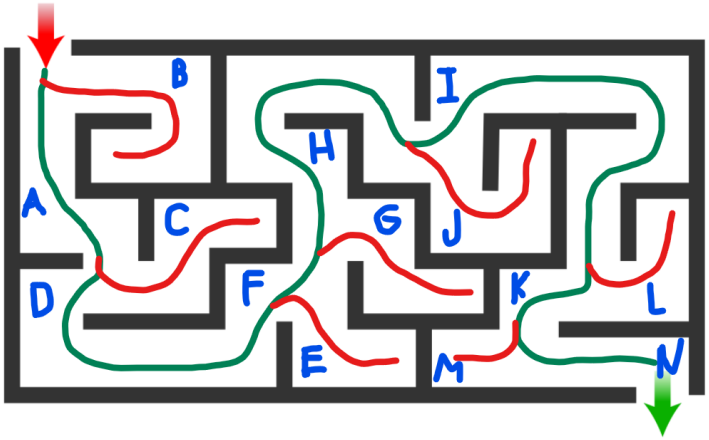
Algoritam 3.11 Sve permutacije elemenata zadanog niza.

Parametri: niz - niz elemenata; p - niz koji sadrži jednu permutaciju; $rezultat$ - skup svih permutacija
Rezultat: skup svih permutacija

```
1:  function PERMUTACIJE(niz, p, rezultat)
2:      if niz.veličina-niza = 0 then
3:          rezultat ← rezultat ∪ p
4:      else
5:          for i = 1..niz.veličina-niza do
6:              prvi ← niz[i]
7:              ostatak ← niz[1..(i − 1)] + niz[(i + 1)..niz.veličina-niza]
8:              PERMUTACIJE(ostatak, p + prvi, rezultat)
9:          end for
10:         return rezultat
11:     end if
12: end function
```



Slika 3.5: Labirint s označenim vraćanjem istim putem.

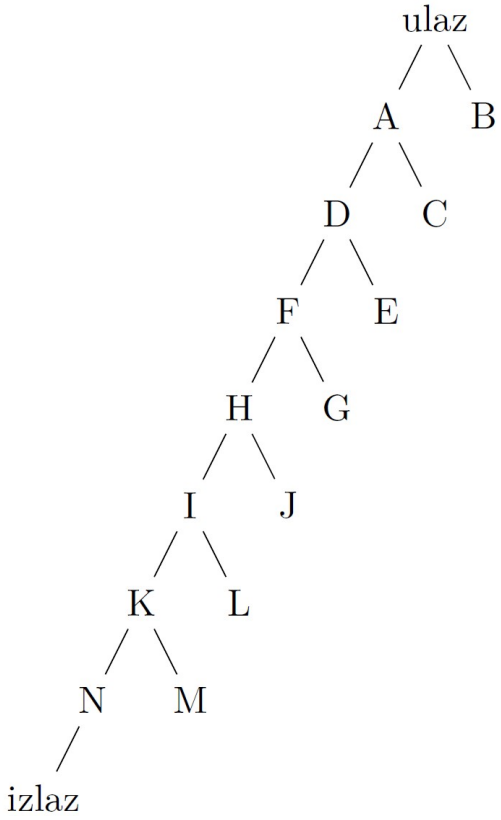


Slika 3.6: Labirint s označenim izborima puteva.

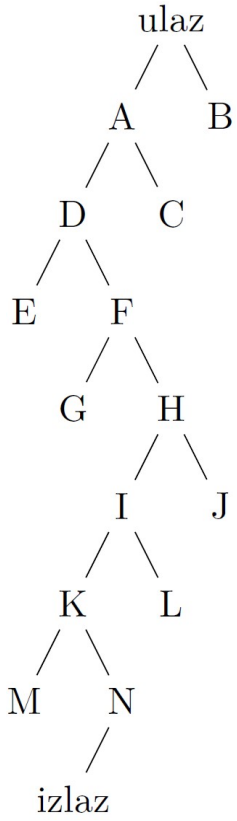
3.2 Rekurzivno traženje i vraćanje istim putem

Slika 3.5 prikazuje jednostavan labirint. Ako pratimo zelenu crtu možemo doći do mjesta u labirintu od kojeg ne možemo dalje. Crvena crta označava da se moramo vratiti na mjesto posljednjeg grananja i pokušati s primjenom neke alternative. To se naziva *vraćanjem istim putem* (engl. *backtracking*). Na Slici 3.6 putevi u labirintu označeni su slovima. Svaki dio puta od jednog grananja do idućeg označen je svojim slovom. Na ovoj se slici može uočiti stablo kod kojeg je svaka grana označena jednim slovom (Slika 3.7). Vidimo da svaki čvor ovog stabla ima dvije alternative za svako grananje jer ovaj labirint kod svakog grananja ima dva moguća smjera. Kod složenijih labirinata i njihovo bi stablo bilo složenije. Ovo smo stablo mogli nacrtati i tako da ne vodi uvijek lijeva grana k izlazu. Primjer takvog stabla prikazan je na Slici 3.8. Ovo stablo identično je prethodnom jer su čvorovi identični kao i sljedbenici svakog čvora. Isto tako, vidimo da je put do izlaza isti kod oba stabla: $ulaz, A, D, F, H, I, K, N, izlaz$.

Labirint, dakle, možemo prikazati kao stablo. Labirint na Slici 3.6 možemo u kodu prikazati kao niz koji sadrži podnizove ili *ugnježđene* nizove (Slika 3.9). Na toj slici vidimo



Slika 3.7: Labirint kao stablo.



Slika 3.8: Labirint kao stablo.

```
['ulaz',
 ['A',
  ['D',
   'E',
   ['F',
    'G',
    ['H',
     ['I',
      ['K',
       'M',
       ['N', 'izlaz']],
      'L'],
     'J']]],
  'C'],
 'B']
```

Slika 3.9: Stablo sa Slike 3.8 prikazano kao lista.

da se takav niz sastoji od podnizova od kojih svaki ima dva člana koji predstavljaju lijevu (prvi član) i desnu (drugi član) granu.

Na Slici 3.10 prikazan je još jedan labirint gdje su slovima označena grananja. Tom labirintu odgovara stablo na Slici 3.11. Traženje izlaza u takvom labirintu se, prema tome, svodi na pretraživanje niza. Funkcija NAĐI-IZLAZ u Algoritmu 3.12 pronalazi izlaz iz labirinta primjenom vraćanja istim putem. Vidimo da smo se nakon čvora L morali vratiti na razinu čvorova A, B i C gdje je pronađen izlaz. To je vraćanje istim putem: kada smo došli do čvora L vidjeli smo da ne možemo dalje pa smo se vraćali unazad sve dok nismo došli do čvora čije sljedbenike još nismo ispitali. Ako govorimo o labirintu, vratili smo se unazad do mjesta kod kojeg je postojao put koji još nismo istražili, što je u ovom primjeru C. Prema tome, ako slijedimo čvorove počevši od čvora $ulaz$ doći ćemo do izlaza putem $ulaz \rightarrow C \rightarrow izlaz$.

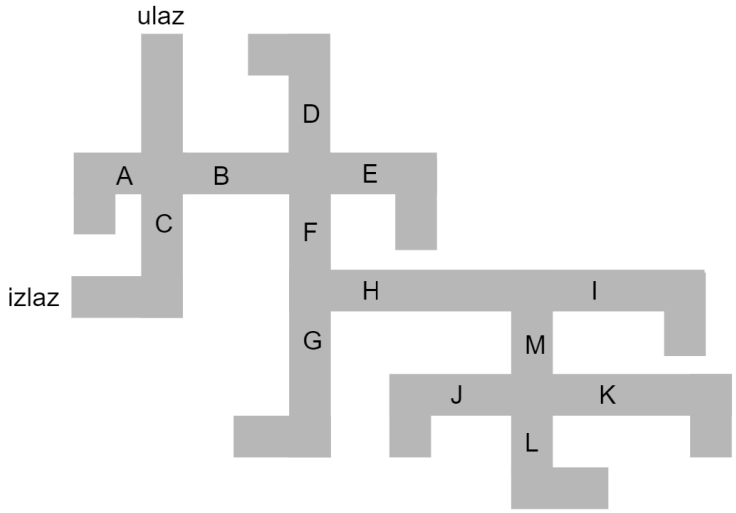
Možemo primjetiti da je Algoritam 3.12 praktički identičan Algoritmu 3.9, s razlikom da kod Algoritma 3.12 tražimo specifičan element nazvan $izlaz$, dok je kod Algoritma 3.9 element koji tražimo zadan kao parametar. Pretraživanje ovakvih ugnježđenih struktura podataka primjenjivo je u programiranju kod mnogih problema, što ćemo vidjeti kasnije u poglavlju o strukturama podataka.

Vraćanje istim putem karakteristično je za neke rekurzivne postupke kao što je pretraživanje ugnježđenih struktura podataka, ali i mnoge druge. Kada se kaže da se u nekom rekurzivnom postupku upotrebljava vraćanje istim putem, znamo da se radi o nečemu što uključuje alternativne pokušaje pronalaska rješenja.

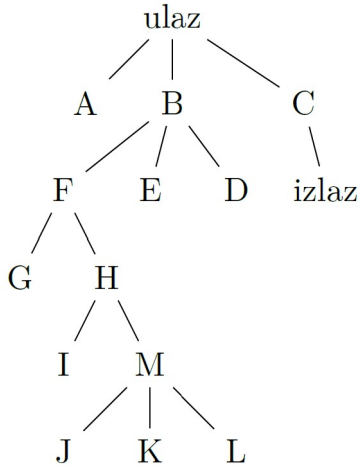
Algoritam 3.12 Pronalaženje izlaza iz labirinta.

Parametri: $stablo$ - stablo kao ugnježđeni nizovi koji predstavljaju labirint
Rezultat: $true$ ako izlaz postoji, inače $false$

```
1:  function NAĐI-IZLAZ(stablo)
2:      for all čvor in stablo do
3:          if točka je niz then
4:              if NAĐI-IZLAZ(cvor) then
5:                  return true
6:              end if
7:          else
8:              if cvor = izlaz then
9:                  return true
10:             end if
11:         end if
12:     end for
13:     return false                                    ▷ vraćanje
14: end function
```



Slika 3.10: Labirint.



Slika 3.11: Stablo labirinta na Slici 3.10.

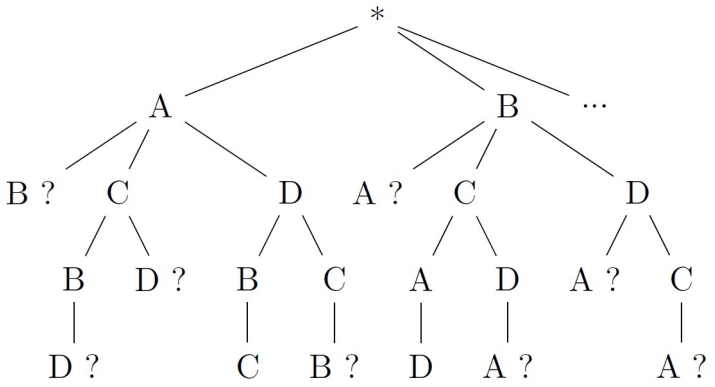
3.2.1 Primjer: Elementi u poretku s ograničenjima

U ovom dijelu proučit ćemo još jedan primjer upotrebe rekurzije vraćanjem istim putem. Zadan je problem: četiri osobe, A, B, C i D treba rasporediti za okruglim stolom po sljedećim pravilima:

- B ne smije sjediti pored A
- D mora sjediti odmah lijevo od A.

Jedan takav poredak je ADBC, gdje možemo zamisliti da elemente slažemo oko okruglog stola u smijeru kazaljki na satu, uzimajući ih iz ovakvog niza s lijeva na desno.

Na koji način možemo dobiti sve kombinacije sjedenja koje zadovoljavaju gornja pravila? Jedan način bio bi generirati sve permutacije s ova četiri elementa, ali za veći broj elemenata to bi bilo nepraktično. Ovdje ćemo primijeniti tehniku programiranja s rekurzijom i vraćanjem istim putem. Prvo od čega možemo krenuti prikaz je rekurzivnog stabla i sljedeće strategije: krenemo od prvog elementa, recimo A, i onda uzmemo idući element, recimo B. Ako kombinacija tih dvaju elemenata ne narušava bilo koje pravilo uzmemo treći element, i tako dalje. Ako je barem jedno pravilo narušeno, ne uzimamo sljedeći element nego zamijenimo zadnji koji smo uzeli nekim drugim i opet ponavljamo isti postupak. Postupak ponavljamo dok ne nađemo sve kombinacije sjedenja koje zadovoljavaju zadana pravila.
Početak takvog rekurzivnog stabla prikazan je na Slici 3.12. Ako prelazak stabla započnemo s elementom A i nakon njega odaberemo B, već imamo poredak koji narušava prvo pravilo. Zbog toga nema potrebe dalje tražiti i na tom mjestu prekidamo traženje te pokušavamo novu kombinaciju koja počinje s A, onu koja nakon A odabire C. Djelomična kombinacija AC ne krši niti jedno pravilo pa nakon C uzimamo i B. ACB i dalje ne krši niti jedno pravilo pa nakon B uzimamo posljednji element, D.Kombinacija ACBD krši drugo pravilo, a to je da D mora sjediti odmah lijevo od A. U ovom slučaju D se nalazi desno od A. Sada se opet moramo vratiti do čvora gdje možemo odabrati neki drugi element, a to je C. Nakon C možemo umjesto B odabrati samo D pa imamo parcijalnu kombinaciju ACD



Slika 3.12: Dio rekurzivnog stabla s odgovarajućim poretkom sjedenja.

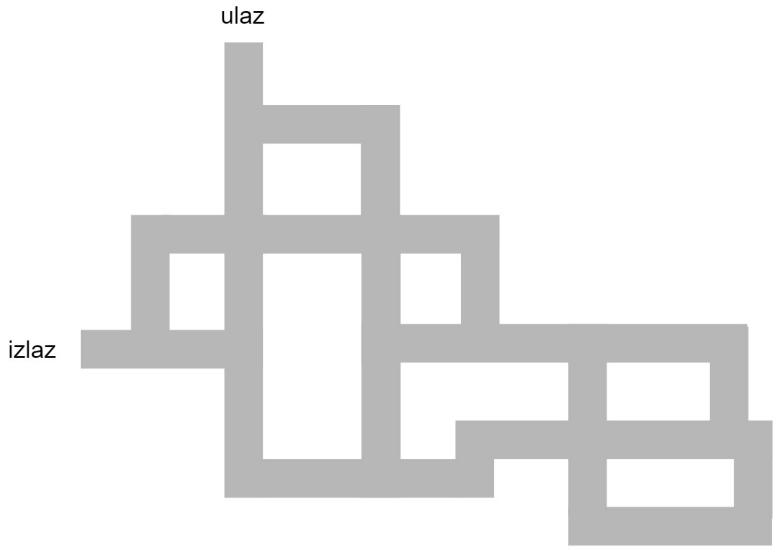
koja ne zadovoljava drugo pravilo. Ponovo se vraćamo sve do A gdje za idući element pokušamo s jedinom preostalom alternativom, D. Vidimo da polazeći od ovog elementa možemo dobiti jednu ispravnu kombinaciju, ADBC. Na isti način, polazeći od prvog elementa B možemo dobiti ispravnu kombinaciju BCAD i preostale dvije kombinacije. Ovaj postupak primijenjen je u Algoritmu 3.13 u funkciji POREDAK. Ovaj bismo postupak mogli proširiti i na više elemenata i/ili pravila jer funkcija POREDAK ne pretpostavlja ništa o broju elemenata i pravila. Ona samo treba podatak o tome zadovoljava li neki poredak (djelomičan ili potpun) uvjete.

Algoritam 3.13 Pronalaženje poredaka koji zadovoljavaju zadana ograničenja.

Parametri: $elementi$ - niz elemenata; $poredak$ - pronađeni poredak; $rezultat$ - skup zadovoljavajućih poredaka
Rezultat: skup zadovoljavajućih poredaka

```
1: function POREDAK(elementi, poredak, rezultat)
2:     if uvjeti nisu ispunjeni then
3:         return                          ▷ vraćanje i pokušaj iduće alternative
4:     end if
5:     if elementi.veličina-niza = 0 then
6:         rezultat ← rezultat ∪ poredak
7:     else
8:         for i = 1..elementi.veličina-niza do
9:             prvi ← elementi[i]
10:            ostatak ← elementi[0..(i − 1)] + elementi[(i + 1)..elementi.veličina-niza]
11:            POREDAK(ostatak, poredak + prvi, rezultat)
12:        end for
13:        return rezultat
14:    end if
15: end function
```

Problemi ovog tipa, kod kojih treba naći rješenje koje zadovoljava zadane uvjete nazivaju se *problemi zadovoljavanja ograničenja* (engl. *constraint satisfaction problems*) [15].



Slika 3.13: Labirint u kojem se može ići ukrug.

Iz ovog primjera vidimo da je vraćanje istim putem zapravo prekidanje rekurzivnih poziva Ako vidimo da iz bilo kojeg razloga nema potrebe ići dalje (dublje) prekidamo postupak, odnosno vraćamo se iz rekurzivnog poziva i pokušamo s drugom alternativom. Nadalje, ovdje smo uštedjeli i na vremenu jer nismo analizirali sve permutacije zadanih elemenata nego smo kod mnogih prekinuli traženje dok još nisu bile potpune. Možemo primjetiti da je funkcija POREDAK jako slična funkciji PERMUTACIJE. Razlog je što funkcija POREDAK također generira permutacije, ali je razlika u tome što ona traži samo određene permutacije, to jest one koje zadovoljavaju zadane uvjete. Dakle, funkcija POREDAK traži i prekida traženje kada prepozna da nema potrebe tražiti dublje u rekurzivno stablo, a funkcija PERMUTACIJE samo generira sve permutacije. Rekurzija s vraćanjem istim putem korisna je upravo za ovakve probleme gdje tražimo nešto što zadovoljava zadana ograničenja.

3.3 Zadaci

1. Napišite rekurzivnu funkciju *okreni(p)* koja okreće listu p. Primjer:

```
>>> okreni([1, 2, 3])
[3, 2, 1]
```

2. Proširite funkciju GEN_KOMBINACIJE tako da generira kombinacije koje se sastoje samo od zadanog broja elemenata. Primjerice, za zadani broj elemenata 2 i niz (a, b, c) ova funkcija trebala bi vratiti samo $\{(a, c), (b, c), (a, b)\}$.

3. U sustavu ISVU predmet može imati druge predmete koji su mu ekvivalenti. Napišite rekurzivnu funkciju koja će izdvojiti sve predmete koji spadaju u istu skupinu ekvivalenata za neki zadani predmet. Ulazni podaci će biti oblika <ISVU šifra 1>:<ISVU šifra 2>, što znači da su predmeti s ISVU šiframa 1 i 2 ekvivalenti. Primjerice,

```
190:200
199:400
118:105
190:140
120:100
105:132
144:145
105:199
```

   Sada bi prema gornjem primjeru za predmet 118 ovaj program izdvojio sljedeće ekvivalente: $105, 132, 199, 400$.

4. Napišite rekurzivnu funkciju koja za niz brojeva ispisuje sve kombinacije tih brojeva čiji je zbroj neka zadana vrijednost. Primjerice, za niz $[5, 2, 1, 2, 1, 8, 4]$ i vrijednost 4 program treba ispisati $[[2, 2], [1, 1, 2], [1, 1, 2], [4]]$.

5. Napišite rekurzivnu funkciju koja generira sve liste zadane duljine tako da se ispišu sve kombinacije dvaju elemenata (slova, znamenke i sl.). Primjerice, za elemente 0 i 1 te duljinu 3 ta funkcija treba vratiti/ispisati sljedeće:

```
[0, 0, 0]
[1, 0, 0]
[0, 1, 0]
[1, 1, 0]
[0, 0, 1]
[1, 0, 1]
[0, 1, 1]
[1, 1, 1]
```

   Na jednostavnom primjeru prikažite rekurzivno stablo.

6. Napišite rekurzivnu funkciju koja će ispisati sve načine na koje se može proći N stepenica ako se možemo penjati po jednu ili dvije stepenice odjednom. Primjerice, tri stepenice mogu se proći na tri načina: $1-1-1$, $1-2$ i $2-1$.

7. Neka je zadan broj N i dvije operacije:

   (a) `dodaj 1` ($x+1$)

   (b) `udvostruči` ($x*2$)

   gdje je *x* rezultat prethodne operacije. Napišite rekurzivnu funkciju koja će pronaći minimalni broj operacija da bi se došlo do broja N počevši od 0. Neka je $N=7$. Tada je minimalni niz operacija $0+1=1, 1+1=2, 1+2=3, 3*2=6, 6+1=7$, što je ukupno pet operacija. Međutim, sljedeći niz operacija nije minimalan: $0+1=1, 1+1=2, 2*2=4, 4+1=5, 5+1=6, 6+1=7$, što daje ukupno šest operacija. Ovdje je problem u tome što nakon množenja $2*2=4$ moramo imati još tri zbrajanja. **Program treba ispisati pronađeni minimalni niz operacija.**

8. Neka je zadan niz od N brojeva gdje svaki broj predstavlja najveći pomak udesno koji možemo napraviti od indeksa tog broja (s tim da pomak može biti i manji od tog broja). Napišite rekurzivnu funkciju koja će dati najmanji broj skokova potrebnih da se dođe do kraja niza počevši od prvog broja, gdje broj 0 označava kraj. Primjerice, za niz $[2,4,1,1,1]$ najmanji broj skokova bio bi 2 ako prolazimo kroz brojeve $2, 4, 1$, gdje od broja 2 napravimo skok za jedno mjesto udesno, a od broja 4 napravimo skok za tri mjesta udesno.

9. Zadana je matrica veličine NxM i niz slova $A, B, C, ...$ Napišite rekurzivnu funkciju koja će posložiti zadani niz slova tako da se dva ista slova ne nalaze jedno pored drugoga po vertikali, horizontali ili dijagonali. Primjerice, za matricu $4x2$ i niz slova A, B, CiD, kombinacija

   ```
   ABCD
   DCBA
   ```

   nije dobro rješenje jer su dva slova B jedno pored drugoga (po dijagonali), isto kao i dva slova C. Međutim, sljedeće rješenje je dobro:

   ```
   ABCD
   CDAB
   ```

10. Zadana je matrica veličine NxN u kojoj je u svakom polju različit broj. Napišite rekurzivnu funkciju koja će u takvoj matrici pronaći najdulji put od jedne vrijednosti do druge tako da se taj put sastoji samo od vrijednosti koje rastu za 1 (počevši od neke polazne vrijednosti) i nalaze se na poljima koja su jedno do drugoga samo po vertikali ili horizontali. Drugim riječima, za polazno polje (i,j) sljedeće polje može bit jedno od $(i+1,j), (i-1,j), (i,j+1), (i,j-1)$. U sljedećem primjeru zadana je matrica $3x3$:

$$\begin{bmatrix} 7 & 8 & 9 \\ 2 & 5 & 6 \\ 3 & 4 & 1 \end{bmatrix}$$

Najdulji put koji možemo dobiti je duljine 5 za vrijednosti $2 \to 3 \to 4 \to 5 \to 6$, počevši od polja $(1, 2)$ s vrijednošću 2.

11. Napišite program koji traži izlaz iz labirinta u kojem se može ići u krug (vidi Sliku 3.13).

POGLAVLJE 4

Algoritmi za sortiranje

Ovisno o vremenskoj složenosti, algoritme za sortiranje temeljene na uspoređivanju elemenata možemo podijeliti u dvije skupine, s kvadratnom i s log-linearnom vremenskom složenošću. Algoritme s kvadratnom vremenskom složenošću smatramo one s prosječnom vremenskom složenošću $\Theta(n^2)$. Primjerice, to su algoritmi *Bubble sort*, *Selection sort* i *Insertion sort* (Tablica 4.1). S druge strane, algoritmima s log-linearnom vremenskom složenošću smatramo one koji korištenjem rekurzije, tj. strategije "podijeli pa vladaj" prosječno imaju vremensku složenost $\Theta(n \log_2 n)$. Toj skupini pripadaju algoritmi poput *Merge sort*, *Heap sort* i *Quick sort* (Tablica 4.1).

Tablica 4.1: Algoritmi za sortiranje (vremenska i prostorna složenost).

Algoritam	Najgori slučaj	Prosječno vrijeme	Najbolji slučaj	Prostorna složenost
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\mathrm{O}(1)$
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\mathrm{O}(1)$
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\mathrm{O}(1)$
Shell sort	$\Theta(n^2)$	-	$\Theta(n \log n)$	$\mathrm{O}(1)$
Merge sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\mathrm{O}(n)$
Tim sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	$\mathrm{O}(n)$
Heap sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\mathrm{O}(1)$
Quick sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\mathrm{O}(\log n)$

U ovom poglavlju, radi njihove raznolikosti, analizirat ćemo rad algoritama za sortiranje navedenih u Tablici 4.1. Dok prva tri algoritma demonstriraju jednostavne koncepte usporedbi i zamjena, oni su osnova za razumijevanje složenijih tehnika sortiranja. Tako se u nastavku opisuju i njihove optimizacije te skupina algoritama koja radi korištenjem rekurzije i strategije "podijeli pa vladaj". Sortiranje nizova demonstrirano je rekurzivnim dijeljenjem i spajanja niza, korištenjem strukture gomila, particioniranjem te hibridnim pristupom, odnosno kombiniranjem više različitih algoritama za sortiranje.

U nastavku su prikazani gore navedeni algoritmi sortiranja. Za svaki od njih pretpostavljamo da kao ulazni parametar primaju niz elemenata za koje su definirane operacije uspoređivanja (kao < i >).

4.1 Bubble sort

Jedan od najjednostavnijih algoritama za sortiranje je *Bubble sort*. Radi na način da se zamjenom susjednih elemenata najveća vrijednost u nizu postavlja na kraj niza (Algoritam 4.1). Iako jednostavan, prosječno vrijeme izvođenja ovog algoritma je $\Theta(n^2)$, što ga čini neprikladnim pri radu s velikim količinama podataka.

Algoritam 4.1 Bubble sort.

```
1: procedure BUBBLE-SORT(niz)
2:     for i ← 0 .. niz.veličina-niza − 1 do
3:         for j ← 0 .. niz.veličina-niza − 1 − i do
4:             if niz[j + 1] < niz[j] then
5:                 zamijeni niz[j] i niz[j + 1]
6:             end if
7:         end for
8:     end for
9: end procedure
```

Za demonstraciju pretpostavimo da je potrebno uzlazno sortirati sljedeći niz brojeva: 5, 2, 9, 1, 5.

Prolazak 1:
5, **2**, 9, 1, 5 // zamjena 5 i 2
2, **5**, **9**, 1, 5 // nema zamjene (5 i 9 su na već dobrim pozicijama)
2, 5, **9**, **1**, 5 // zamjena 9 i 1
2, 5, 1, **9**, **5** // zamjena 9 i 5
2, 5, 1, 5, $[9]$

Nakon svake iteracije vanjske petlje u algoritmu *Bubble sort*, zadnji element niza postaje sortiran pa se na isti način sortira ostatak niza.

Prolazak 2:
2, **5**, 1, 5, $[9]$ // nema zamjene
2, **5**, **1**, 5, $[9]$ // zamjena 5 i 1
2, 1, 5, $[5]$, $[9]$

U svakoj sljedećoj iteraciji vanjske petlje pregledava se samo ostatak niza (niz.*veličina-niza*$-1-i$), gdje i predstavlja broj već obavljenih iteracija vanjske petlje, odnosno broj već sortiranih elemenata niza. Kako su ti elementi već sortirani, vanjska petlja preskače njihovu provjeru, pa svaka sljedeća iteracija traje za 1 element kraće.

Prolazak 3:
2, **1**, 5, [5], [9] // zamjena 2 i 1
1, 2, [5], [5], [9] // nema zamjene

Već nakon tri prolaska cijeli niz je uspješno sortiran. U ovom slučaju niz je sortiran i prije zadnje predviđene iteracije vanjske petlje algoritma, pa je zbog ovakvih slučajeva algoritam *Bubble sort* moguće dodatno optimizirati korištenjem zastavice (Algoritam 4.2).

Algoritam 4.2 Bubble sort sa zastavicom.

```
1:  procedure BUBBLE-SORT-SA-ZASTAVICOM(niz)
2:      for i ← 0 .. niz.veličina-niza − 1 do
3:          zastavica ← false                    ▷ zamjena elemenata nije se dogodila
4:          for j ← 0 .. niz.veličina-niza − 1 − i do
5:              if niz[j] > niz[j + 1] then
6:                  zamijeni niz[j] i niz[j + 1]
7:                  zastavica ← true             ▷ dogodila se zamjena elemenata
8:              end if
9:          end for
10:         if zastavica = false then            ▷ ako nije bilo zamjene elemenata ...
11:             break                            ▷ ... niz je sortiran pa možemo prekinuti
12:         end if
13:     end for
14: end procedure
```

Pri početku svake iteracije vanjske petlje $zastavica$ ima vrijednost $false$ te poprima vrijednost $true$ samo ukoliko se u unutrašnjoj petlji dogodila zamjena elemenata niza. Ukoliko nije došlo do niti jedne zamjene, cijeli niz je u međuvremenu već sortiran te nema potrebe za daljnjim iteracijama vanjske petlje.

4.2 Selection sort

Za razliku od algoritma *Bubble sort* koji postavlja najveće elemente na kraj niza, *Selection sort* radi obrnuto, odnosno postavlja najmanje elemente na početak niza. U svakom prolazu trenutni element niza deklarira se kao najmanji, a zatim se u ostatku niza s desne strane traži element s manjom vrijednošću. Ukoliko je najmanji element sa desne strane niza manji od trenutnog elementa niza, provodi se njihova zamjena (Algoritam 4.3). Prosječno vrijeme izvođenja ovog algoritma je također $\Theta(n^2)$.

Za demonstraciju algoritma koristit ćemo isti niz brojeva: 5, 2, 9, 1, 5.

Prolazak 1:
5, 2, 9, **1**, 5 // $min = 0$ (5), a u ostatku niza najmanji broj je 1
[1], 2, 9, 5, 5 // nakon zamjene 5 i 1

Algoritam 4.3 Selection sort.

```
1: procedure SELECTION-SORT(niz)
2:     for i ← 0 .. niz.veličina-niza − 1 do
3:         min ← i                                  ▷ najmanji element je na početku niza
4:         for j ← i + 1 .. niz.veličina-niza do    ▷ provjeri ostatak niza
5:             if niz[j] < niz[min] then
6:                 min ← j
7:             end if
8:         end for
9:         if min ≠ i then
10:            zamijeni niz[min] i niz[i]
11:        end if
12:    end for
13: end procedure
```

Nakon prvog prolaska vanjske petlje algoritma sortiran je nulti element niza (broj 1). Po istom načelu dalje se sortira ostatak niza, gdje se u svakoj sljedećoj iteraciji vanjske petlje nastavlja od sljedećeg elementa.

Prolazak 2:
[1], **2**, 9, **5**, 5 // $min = 1$ (2), a u ostatku niza najmanji broj je 5
[1], [2], 9, 5, 5 // zamjena 2 i 5 nije potrebna

Prolazak 3:
[1], [2], **9**, **5**, 5 // $min = 2$ (9), a u ostatku niza najmanji broj je 5
[1], [2], [5], 9, 5 // nakon zamjene 9 i 5

Prolazak 4:
[1], [2], [5], **9**, **5** // $min = 3$ (9), a u ostatku niza najmanji broj je 5
[1], [2], [5], [5], [9] // nakon zamjene 9 i 5

Iako algoritmi *Bubble sort* i *Selection sort* imaju isto prosječno vrijeme izvođenja, *Bubble sort* zahtijeva više izmjena nego *Selection sort*. U svakom prolazu, *Bubble sort* može zamijeniti susjedne elemente više puta dok se najveći element ne postavi na svoju ispravnu mjesto. S druge strane, *Selection sort* čini samo jednu zamjenu po prolazu, pomičući najmanji (ili najveći, ovisno o implementaciji) element izravno na njegovo ispravno mjesto.

4.3 Insertion sort

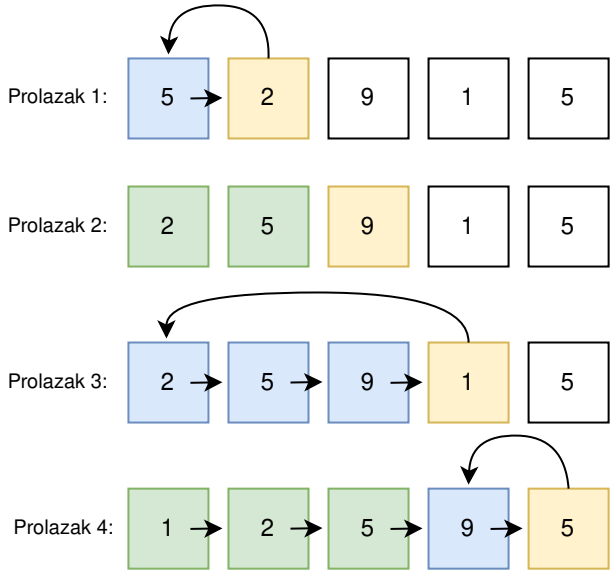
Insertion sort još je jedan u nizu algoritama za sortiranje s prosječnom vremenoskom složenošću $\Theta(n^2)$. Kao takav, pogodan je tek za sortiranje manje veličine podataka. Dizajniran je za učinkovito sortiranje elemenata postupnom izgradnjom sortiranog podniza unutar nesortiranog niza. Temeljna ideja iza ovog algoritma je ispitati svaki nesortirani

element i umetnuti ga na njegovu ispravnu poziciju unutar sortiranog podniza, čime se sortirani podniz postupno povećava (Algoritam 4.4).

Algoritam 4.4 Insertion sort.

```
1:  procedure INSERTION-SORT(niz)
2:      for i ← 1 .. niz.veličina-niza − 1 do
3:          trenutni ← niz[i]
4:          j ← i − 1
5:          while j ≥ 0 and niz[j] > trenutni do
6:              niz[j + 1] ← niz[j]              ▷ premjesti elemente za jedno mjesto udesno
7:              j ← j − 1
8:          end while
9:          niz[j + 1] ← trenutni                ▷ umetni trenutni element
10:     end for
11: end procedure
```

Insertion sort algoritam počinje od elementa s indeksom 1. Ovaj algoritam provjerava sve elemente koji su lijevo od trenutnog elementa. Kako nulti element nema nikoga sa svoje lijeve strane, može ga se preskočiti. Zatim pomiče sve elemente koji su veći od trenutnog elementa za jedno mjesto udesno sve dok ne dođe do elementa koji nije veći od njega ili do početka niza. Trenutni element tada se umeće na mjesto zadnjeg pomaknutog elementa. Kao i u prethodnim slučajevima, algoritam ćemo demonstrirati na nizu brojeva $5, 2, 9, 1, 5$ (Slika 4.1).



Slika 4.1: Demonstracija Insertion sort algoritma.

U prvom prolasku trenutni element je onaj s indeksom 1 (broj 2). Sve elemente s lijeve strane koji su veći od njega (samo je jedan element, broj 5) pomičemo za jedno mjesto udesno. Kad smo došli do samog početka niza na to mjesto umećemo trenutni element (broj 2). Konačno, na kraju prvog prolaza imamo niz 2, 5, 9, 1, 5, gdje sortirani podniz počinje brojem 2. U drugom prolazu trenutni element postaje onaj s indeksom 2 (broj 9).

Budući da s njegove lijeve strane ne postoji niti jedan broj već od njega, ne događaju se nikakve izmjene pa sortirani podniz sada sadrži brojeve 2, 5 i 9. U trećem prolazu trenutni element postaje onaj s indeksom 3 (broj 1), a kako su svi elementi s njegove lijeve strane veći od njega, pomiču se za jedno mjesto udesno. Zatim se trenutni element (broj 1) umeće na početak niza. I konačno, u četvrtom prolazu zadnji element niza (broj 5) umeće se na mjesto broja 9 koji se pomiče na kraj sortiranog niza.

Kada je riječ o radu s velikim nizovima postoji i brža inačica algoritma *Insertion sort - Binary insertion sort*. Dok osnovni algoritam pomiče elemente pojedinačno kako bi napravio mjesta za novi element, poboljšana inačica iskorištava učinkovitost binarnog pretraživanja za brže pronalaženje točke za umetanje elementa u sortirani podniz. Time se smanjuje broj usporedbi za umetanje jednog elementa s $O(n)$ na $O(log_2 n)$.

4.4 Shell sort

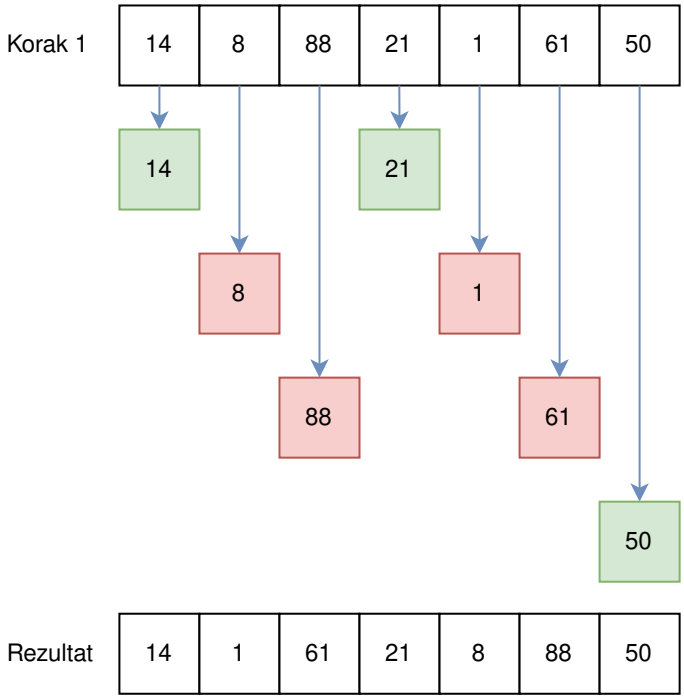
Pri usporedbi i zamjeni elemenata u algoritmu *Insertion sort* koristi *razmak* (eng. *gap*) od samo jednog elementa, što najčešće uzrokuje puno operacija ukoliko se vrijednost elementa treba prebaciti s jednog na drugi kraj niza. Da bismo brže sortirali udaljene elemente može se koristiti algoritam *Shell sort* (Algoritam 4.5) koji inicijalno koristi veći razmak među elementima.

Algoritam 4.5 Shell sort.

```
1:  procedure SHELL-SORT(niz)
2:      n ← niz.veličina-niza
3:      razmak ← n/2                          ▷ razmak je polovica veličine niza
4:      while razmak > 0 do
5:          for i ← razmak .. n − 1 do
6:              temp ← niz[i]
7:              j ← i
8:              while j ≥ razmak and niz[j − razmak] > temp do
9:                  niz[j] ← niz[j − razmak]
10:                 j ← j − razmak
11:             end while
12:             niz[j] ← temp
13:         end for
14:         razmak ← razmak/2          ▷ nakon svakog koraka razmak se dijeli na pola
15:     end while
16: end procedure
```

Shell sort je unaprijeđenje algoritma *Insertion sort* čiji je cilj poboljšati njegovu učinkovitost početnim sortiranjem međusobno udaljenih elemenata (elemenata s većim razmakom) i postupnim smanjenjem tog razmaka u narednim koracima. Ova strategija smanjuje broj zamjena i usporedbi potrebnih za sortiranje, čineći je učinkovitijom za veće nizove. Zbog veličine korištenih razmaka koji uvelike utječu na broj obavljenih operacija nije moguće

precizno utvrditi vrijeme izvršavanja ovog algoritma, ali u najgorem slučaju izvršavat će se kao i *Insertion sort*, odnosno vremenskom složenošću O(n^2).



Slika 4.2: Demonstracija Shell sort algoritma – korak 1.

Pretpostavimo da imamo sljedeći niz brojeva: $14, 8, 88, 21, 1, 61, 50$. Slika 4.2 prikazuje prvi korak sortiranja tog niza korištenjem algoritma *Shell sort* gdje je početni razmak $7/2$, odnosno 3 elementa. Vrše se sljedeće usporedbe:

- 14 i 21. Zamjena nije potrebna.
- 8 i 1. Vrši se zamjena.
- 88 i 61. Vrši se zamjena.
- 51. Nema se s kime uspoređivati.

Nakon prvog koraka niz sadrži sljedeće brojeve: 14, 1, 61, 21, 8, 88, 50. Rubne vrijednosti poput brojeva 1 i 88 sada su puno bliže onim mjestima u nizu gdje zaista trebaju i biti, pa će i količina budućih usporedbi i zamjena biti manja. Veličina razmaka u prvom koraku iznosila je polovicu duljine niza, tj. 3 elementa. U svakom sljedećem koraku razmak se dijeli na pola te se opet ponavljaju prethodno opisane usporedbe i zamjene. Na kraju, kada veličina razmaka bude iznosila 1 (kao u ovom slučaju za naš sljedeći korak) izvršava se klasični algoritam *Insertion sort*.

Bubble sort, *Selection sort*, pa čak i *Insertion sort*, iako su vrijedni za obrazovne svrhe i male aplikacije, suočavaju se s izazovima pri radu s velikim količinama podataka. *Shell sort* predstavlja dostojnu sredinu, ali unatoč tome za visokoučinkovito sortiranje najbolja su rješenja algoritmi iz skupine "podijeli pa vladaj" poput opisanih u nastavku *Merge sort*, *Tim sort* i *Quick sort*.

4.5 Merge sort

Prosječna vremenska složenost od $\Theta(n^2)$ ne čini takve algoritme prikladnim za sortiranje velikih nizova, pa tada možemo koristiti skupinu bržih algoritama koji koriste strategiju "podijeli pa vladaj". Njihova prosječna vremenska složenost je $\Theta(n \log_2 n)$, a prvi od njih je *Merge sort*. Radi na način da ulazni niz rekurzivno dijeli na dva dijela sve dok ne dođe do jednog elementa. Zatim, generirani podnizovi spajaju se nazad na sortirani način korištenjem procedura MERGE (Algoritam 4.6) i MERGE-SORT (Algoritam 4.7).

Algoritam 4.6 Merge.

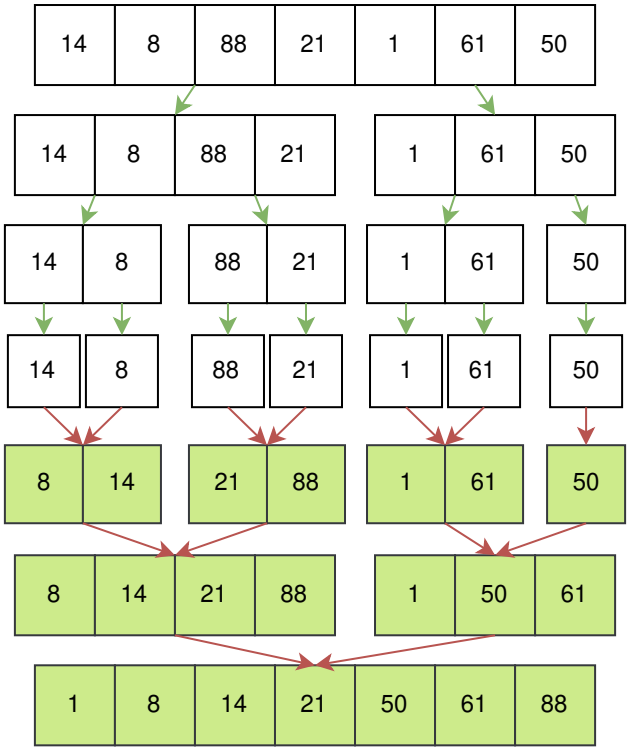
```
1: function MERGE(lijevo, desno)
2:     rezultat ← []
3:     while lijevo.veličina-niza > 0 and desno.veličina-niza > 0 do
4:         if lijevo[1] ≤ desno[1] then
5:             dodaj lijevo[1] u rezultat
6:             ukloni lijevo[1] iz lijevo
7:         else
8:             dodaj desno[1] u rezultat
9:             ukloni desno[1] iz desno
10:        end if
11:    end while
12:    dodaj lijevo u rezultat
13:    dodaj desno u rezultat
14:    return rezultat
15: end function
```

Algoritam 4.7 Merge sort.

```
1: function MERGE-SORT(niz)
2:     if niz.veličina-niza ≤ 1 then
3:         return niz                ▷ osnovni slučaj: niz je prazan ili već sortiran
4:     end if
5:     sredina ← niz.veličina-niza/2
6:     lijevo ← MERGE-SORT(niz[1..sredina])          ▷ rekurzivno sortiraj lijevi dio
7:     desno ← MERGE-SORT(niz[(sredina + 1)..(niz.veličina-niza)])   ▷ a zatim desni
8:     return MERGE(lijevo, desno)                   ▷ spoji sortirane polovice
9: end function
```

Za potrebe demonstracije pretpostavimo da imamo sljedeći niz brojeva koje je potrebno sortirati algoritmom *Merge sort*: $[14, 8, 88, 21, 1, 61, 50]$. Postupak sortiranja demonstriran je na Slici 4.3, a započinje dijeljenjem postojećeg niza u dva dijela. Nakon dijeljenja izvornog niza dobili smo dva podniza: $[14, 8, 88, 21$ i $1, 61, 50]$. Na isti način dijele se lijevi i desni podniz. Tada dobivamo četiri podniza: $[14, 8]$, $[88, 21]$, $[1, 61]$ i $[50]$, te u zadnjem dijeljenju sedam podnizova: $[14]$, $[8]$, $[88]$, $[21]$, $[1]$, $[61]$ i $[50]$. Podnizove više nije moguće dalje dijeliti i sada počinje njihovo spajanje. U tom procesu podnizovi se spajaju na način da su njihovi elementi sortirani. Tako se generiraju sljedeći podnizovi: $[8, 14]$,

[21, 88], [1, 61] i [50]. Njihovim spajanjem dobivamo dva veća sortirana podniza: [8, 14, 21, 88] i [1, 50, 61] te u zadnjem spajanju konačni sortirani niz.



Slika 4.3: Demonstracija Merge sort algoritma.

Njegova stabilnost, predvidljivost i dosljedna izvedba čine algoritam *Merge sort* jako dobrim izborom za sortiranje velike količine podataka, odnosno svugdje gdje su performanse sortiranja bitne. Međutim, ovaj algoritam zahtijeva dodatnu memoriju zbog dijeljenja i spajanja podnizova što treba razmotriti u okruženjima s ograničenom memorijom.

4.6 Tim sort

Tim sort hibridni je algoritam za sortiranje koji kombinira koncepte algoritama *Insertion sort* i *Merge sort*. U programskim jezicima Python i Java koristi se kao podrazmijevani algoritam za sortiranje, a njegova primjena postoji i u operacijskom sustavu Android pri sortiranju nizova i listi. Radi na jednostavnom načelu koji se sastoji od tri koraka:

- Odredi veličinu bloka (broj elemenata) kojim će se podijeliti niz.
- Svaki pojedini blok, često zvan i *pokušaj* (engl. *run*) sortiraj algoritmom *Insertion sort*.
- Sortirane blokove spoji korištenjem algoritma *Insertion sort* ili funkcijom kao što je MERGE.

Prilikom operacije spajanja blokova, ako je veličina pojedinog bloka manja od predefiniranog praga, *Tim sort* se prebacuje na korištenje algoritma *Insertion sort* za spajanje ovih manjih blokova. *Insertion sort* je učinkovitiji pri radu s malim nizovima te ima manje zahtjeva za resursima u odnosu na funkciju MERGE. S druge strane, veliki blokovi se spajaju korištenjem funkcije MERGE. Kombinacija ovih dviju tehnika sortiranja omogućuje *Tim sortu* da bude vrlo prilagodljiv i učinkovit za širok raspon ulaznih podataka, uključujući slučajeve u kojima bi ulazni podaci mogli biti i djelomično sortirani. Najgore i prosječno vrijeme izvođenja ovog algoritma je $\Theta(n \log_2 n)$, a u najboljem slučaju (kada je riječ o manjim nizovima gdje dominira komponenta algoritma insertion sort) prosječno vrijeme izvođenja je $\Theta(n)$.

4.7 Quick sort

Quick sort jedan je od najbržih algoritama za sortiranje. Također koristi strategiju "podijeli pa vladaj" te mu je prosječna vremenska složenost $\Theta(n \log n)$. Osnovna ideja je da se u odnosu na odabrani *stožerni element* (engl. *pivot*) niz sortira tako da svi brojevi manji od njega budu s lijeve strane niza, a svi veći sa desne strane. Zatim, ovisno o položaju stožernog elementa, niz se rekurzivno dijeli na dva dijela, tj. dio prije i dio poslije stožernog elementa, nakon čega se prethodno opisani postupak ponavlja za oba nova podniza.

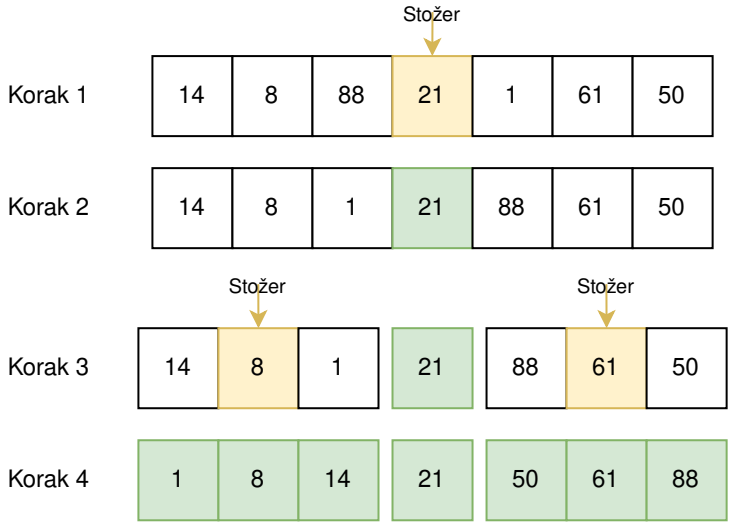
Algoritam 4.8 Quick sort sa srednjim elementom kao stožernim elementom.

```
 1: function QUICK-SORT(niz, donji, gornji)
 2:     if donji < gornji then
 3:         stozerni ← (donji + gornji)/2                ▷ lokacija stožernog elementa
 4:         stozerni ← PARTITION(niz, donji, gornji, stozerni)
 5:         QUICK-SORT(niz, donji, stozerni − 1)
 6:         QUICK-SORT(niz, stozerni + 1, gornji)
 7:     end if
 8: end function

 9: function PARTITION(niz, donji, gornji, stozerni)
10:     zamijeni niz[stozerni] i niz[gornji]          ▷ pomakni stožerni element na kraj
11:     stozerni ← niz[gornji]
12:     i ← donji − 1
13:     for j ← donji .. gornji − 1 do
14:         if niz[j] ≤ stozerni then
15:             i ← i + 1
16:             zamijeni niz[i] i niz[j]
17:         end if
18:     end for
19:     zamijeni niz[i + 1] i niz[gornji]      ▷ postavi stožerni element na točnu poziciju
20:     return i + 1
21: end function
```

Quick sort postoji u mnogo inačica, a najčešće se razlikuju u načinu odabira stožernog

elementa. Tako, primjerice, stožerni element može biti odabran kao prvi, zadnji, srednji ili slučajni element niza. Algoritam 4.8 prikazuje pseudokod implementacije algoritma *Quick sort* kada je stožerni element srednji element niza. Za njegovu demonstraciju opet ćemo koristiti sljedeći niz brojeva: $14, 8, 88, 21, 1, 61, 50$ (Slika 4.4).



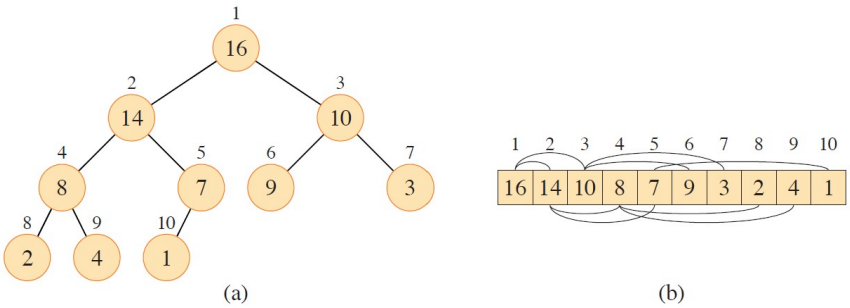
Slika 4.4: Demonstracija Quicksort algoritma.

U početnom nizu od sedam elemenata odabiremo srednji element (broj 21) kao stožerni element niza. Zatim, u drugom koraku sve brojeve manje od 21 prebacujemo lijevo, tj. prije stožernog elementa, a sve veće na desnu stranu. Kao rezultat dobivamo niz gdje je broj 21 na ispravnom (sortiranom) mjestu te možemo pristupiti podijeli inicijalnog niza u dva dijela. U trećem koraku sada postoje dva podniza – jedan s elemenatima lijevo od stožernog elementa i jedan s elementima desno od stožernog elementa. Na svakom od tih podnizova primijenjujemo isti postupak, odnosno odabiremo srednji element niza kao stožerni element te manje vrijednosti smještamo lijevo, a veće desno od stožernog elementa. Konačno, već u četvrtom koraku cijeli niz je sortiran.

Iako brz, algoritam *Quick sort* u najgorem slučaju može rezultirati $\Theta(n^2)$ vremenskom složenošću. Najčešće je riječ o situacijama gdje se sortira već sortirani ili djelomično sortirani niz u kombinaciji s korištenjem uvijek iste lokacije stožernog elementa. Zato je preporuka izbjegavati stalno korištenje prvog ili zadnjeg elementa, već umjesto njih kao stožerni element koristiti srednji ili slučajno odabrani element niza.

4.8 Heap sort

Heap sort algoritam je sortiranja temeljen na strukturi podataka koja se zove *binarna gomila* (engl. *binary heap*). Iako su strukture podataka tema idućeg poglavlja, ovdje opisujemo binarnu gomilu jer nam je neophodna za razumijevanje ovog algoritma sortiranja kao i strukture podataka koja se zove *prioritetni red* (engl. *priority queue*).



Slika 4.5: Binarna gomila kao (a) stablo i (b) niz [1].

4.8.1 Binarna gomila

Binarna gomila skoro je potpuno binarno stablo (Slika 5.25) koje ima jedno od sljedećih svojstava:

- svaki čvor sadrži vrijednost veću ili jednaku od svojih sljedbenika ili
- svaki čvor sadrži vrijednost manju ili jednaku od svojih sljedbenika.

U prvom slučaju radi se o *maks-gomili*, a u drugom o *min-gomili*. Ova se struktura podataka može prikazati kao niz (Slika 4.5) koji sadrži *skoro potpuno binarno stablo*. Ovakvo je stablo na svim razinama potpuno popunjeno, osim možda na posljednjoj, gdje jedan dio desne strane može ostati neispunjen (u kojem slučaju su, promatrajući stablo slijeva nadesno, sva mjesta do prvog praznog popunjena). Svaki element niza odgovara jednom čvoru u stablu.

Ako se radi o nizu fiksne veličine koji predstavlja gomilu, možemo razlikovati dva atributa takvog niza: *veličina niza* i *veličina gomile*. Veličina niza ukupan je broj elemenata u nizu, a veličina gomile ukupan broj elemenata u gomili (koji može biti manji od veličine niza ako su neki elementi iz gomile uklonjeni), tako da vrijedi

$$0 \leq \textit{veličina-gomile} \leq \textit{kapacitet-niza}$$

Korijen stabla nalazi se na indeksu 0 dok za indeks čvora i možemo dobiti njegovog roditelja te lijevog i desnog sljedbenika operacijama RODITELJ (Algoritam 4.9), LIJEVO (Algoritam 4.10 [1]) i DESNO (Algoritam 4.11 [1]).

Algoritam 4.9 Roditelj čvora gomile.

Parametri: i - indeks čvora u nizu
Rezultat: roditelj čvora i

```
1: function RODITELJ(i)
2:     return ⌊i/2⌋
3: end function
```

Ove se operacije mogu učinkovito implementirati. Množenje brojem 2 može se implementirati kao pomak bitova za jedno mjesto ulijevo (engl. *left shift*), dok se kod dijeljenja

Algoritam 4.10 Lijevo dijete čvora gomile.

Parametri: i - indeks čvora u nizu
Rezultat: lijevo dijete čvora i

```
1: function LIJEVO(i)
2:     return 2i
3: end function
```

Algoritam 4.11 Desno dijete čvora gomile.

Parametri: i - indeks čvora u nizu
Rezultat: desno dijete čvora i

```
1: function DESNO(i)
2:     return 2i + 1
3: end function
```

brojem 2 pomiču za jedno mjesto udesno (engl. *right shift*). Ove operacije pomaka bitova vrlo su učinkovite na modernim računalima i prevoditelji za programske jezike automatski ih generiraju za operacije množenja i dijeljenja s 2.

Ako gomilu promatramo kao stablo, visina čvora u tom stablu odgovara broju lukova najdužeg puta od tog čvora do nekog lista stabla. Visina gomile je, prema tome, visina njenog korijena. S obzirom da se gomila temelji na cijelom stablu, njena visina je $\Theta(\lg n)$.

4.8.2 Održavanje svojstva gomile

U ovom dijelu prikazani su algoritmi kojima se gomila može upotrebljavati za sortiranje i kao *prioritetni red* (engl. *priority queue*). Ti su algoritmi neophodni za sortiranje, izgradnju i održavanje svojstva gomile.

Algoritam 4.12 [1] održava svojstva gomile (primjerice, nakon dodavanja ili brisanja elementa) i njegov je rad ilustriran Slikom 4.6. Na dijelu (a) vidimo da čvor 2 narušava svojstvo maks-gomile jer nije veći od oba djeteta (izravna sljedbenika). Na dijelu (b) procedura je čvoru 2 vratila svojstvo maks-gomile, ali čvor 4 više ne zadovoljava to svojstvo. Sada će rekurzivan poziv MAKS-GOMILANJE$(A, 4)$ vratiti svojstvo maks-gomile tom čvoru čime dobivamo rezultat prikazan na dijelu (c). Promjene više nisu potrebne jer svi čvorovi sada zadovoljavaju svojstvo maks-gomile.
Vremenska složenost algoritma MAKS-GOMILANJE ovisi o visini stabla jer se element u najgorem slučaju mora popeti ili spustiti na odgovarajuće mjesto u stablu pa je njegova vremenska složenost $O(h)$.

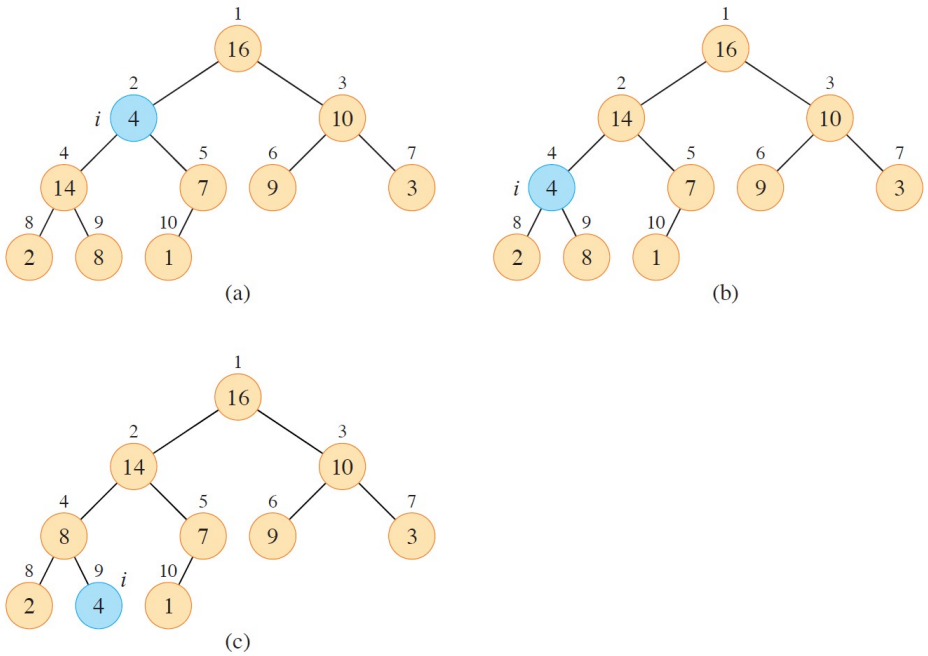
4.8.3 Izgradnja gomile

Algoritam 4.12 možemo iskoristiti za izgradnju gomile na osnovu ulaznog niza. Algoritam za izgradnju gomile uzima u obzir činjenicu da ako na osnovu ulaznog niza izgradimo

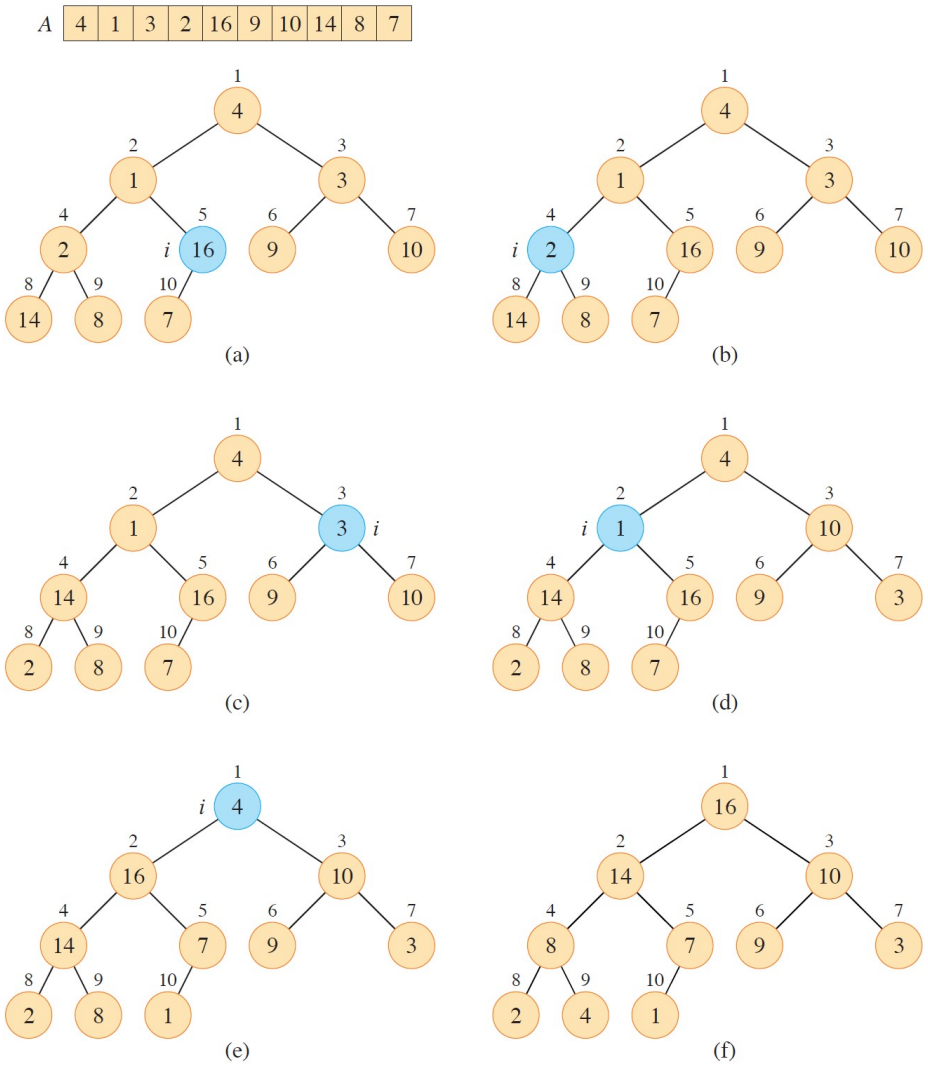
Algoritam 4.12 Održavanje svojstva gomile.

Parametri: A - ulazni niz s gomilom; i - indeks polaznog čvora

```
1:  procedure MAKS-GOMILANJE(A, i)
2:      l ← LIJEVO(i)
3:      d ← DESNO(i)
4:      if l ≤ A.veličina-gomile and A[l] > A[i] then
5:          najveci ← l
6:      else
7:          najveci ← i
8:      end if
9:      if d ≤ A.veličina-gomile and A[d] > A[najveci] then
10:         najveci ← d
11:     end if
12:     if najveci ≠ i then
13:         zamijeni A[i] i A[najveci]
14:         MAKS-GOMILANJE(A, najveci)
15:     end if
16: end procedure
```



Slika 4.6: Rad procedure MAKS-GOMILANJE$(a, 2)$ [1].



Slika 4.7: Izgradnja gomile [1].

stablo kako je prikazano na Slici 4.7 a), elementi na segmentu niza $A[(\lfloor n/2 \rfloor + 1)..n]$ listovi su stabla (čvorovi bez sljedbenika). Algoritam 4.13 [1] radi po ovom načelu.

Algoritam 4.13 Izgradnja gomile.

Parametri: A - ulazni niz koji ovaj algoritam konvertira u gomilu; n - broj elemenata u gomili (veličina gomile)

```
1: procedure IZGRADI-MAKS-GOMILU(A, n)
2:     A.veličina-gomile ← n
3:     for i ← ⌊n/2⌋ downto 1 do
4:         MAKS-GOMILANJE(A, i)
5:     end for
6: end procedure
```

Slika 4.7 prikazuje korake izgradnje gomile prema ovom algoritmu. Početni je čvor (a) onaj na indeksu 5, odnosno čvor 16. S obzirom da taj čvor zadovoljava svojstvo maks-gomile, nisu potrebne promjene i prelazi se na čvor 2 (indeks 4). U ovom čvoru potrebno je uspostaviti maks-gomilu čime dobivamo stablo na slici (c). Sljedeći je čvor 3 (indeks 3) na kojem se također mora uspostaviti maks-gomila pa dolazimo do stabla na slici (d). Slijedi čvor 1 gdje uspostavom maks-gomile dobivamo stablo na slici (e). Posljedni čvor je onaj s vrijednošću 4 gdje uspostavom maks-gomile dobivamo konačnu gomilu prikazanu na slici (f).

Vremenska složenost algoritma MAKS-GOMILANJE je $O(log_2 n)$ u najgorem slučaju ili $O(h)$ za čvor visine h, a algoritam IZGRADI-MAKS-GOMILU poziva MAKS-GOMILANJE $O(n)$ puta (jednom za svaki element niza) pa je gornja granica vremenske složenosti algoritma za izgradnju maks-gomile $O(n\, log_2 n)$. Međutim, ova procjena zanemaruje činjenicu da mnogi čvorovi imaju malu visinu. Ako to uzmemo u obzir (iako se ovdje nećemo baviti složenijim matematičkim izvodima) doći ćemo do vremenske složenosti $O(n)$ za algoritam IZGRADI-MAKS-GOMILU.

4.8.4 Algoritam *Heapsort*

Gomila je struktura podataka na kojoj se temelji algoritam za sortiranje *Heapsort*. Prvi korak ovog algoritma je IZGRADI-MAKS-GOMILU nad ulaznim nizom $A[1..n]$, gdje je n duljina niza. Nakon tog koraka najveći element bit će na mjestu $A[1]$. Sada se taj element smješta na položaj $A[n]$ (gdje i treba biti) nakon čega nastavljamo s nizom $A[1..i-1]$ za koji pozivamo MAKS-GOMILANJE. Ovaj se postupak ponavlja za sve preostale elemente niza. Algoritam 4.14 [1] prikazuje algoritam *Heapsort*.

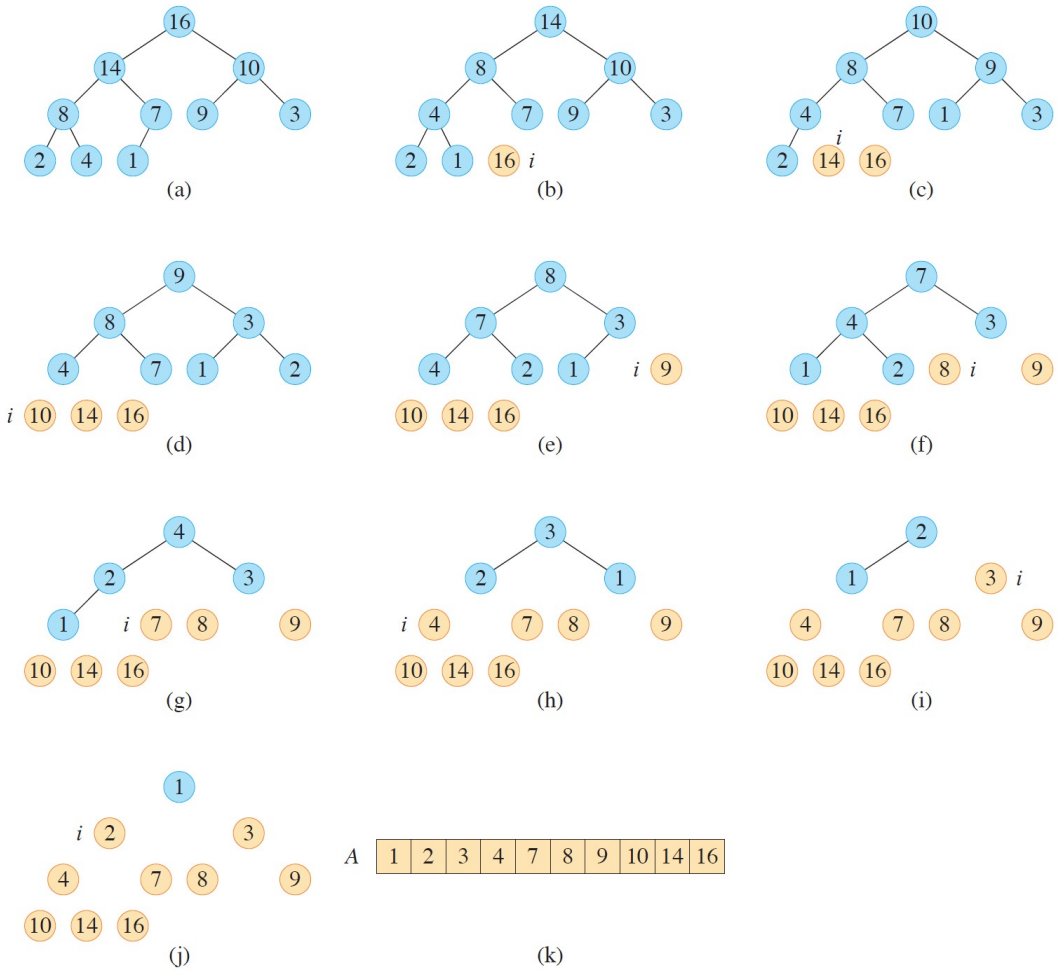
Vremenska složenost algoritma *Heapsort* je $O(n\, log_2 n)$ jer poziv IZGRADI-MAKS-GOMILU treba $O(n)$ vremena, a svaki od $n-1$ poziva MAKS-GOMILANJE treba $O(log_2 n)$ vremena.

Na Slici 4.8 prikazan je rad algoritma HEAP-SORT. U dijelu (a) prikazana je struktura gomile nakon poziva IZGRADI-MAKS-GOMILU. U dijelovima (b) do (j) prikazana je struktura gomile nakon svakog poziva MAKS-GOMILANJE s čvorom na kojem je u tom trenutku

Algoritam 4.14 Heap sort.

Parametri: A - ulazni niz; n - veličina gomile

```
1: procedure HEAP-SORT(A, n)
2:     IZGRADI-MAKS-GOMILU(A, n)
3:     for i ← n downto 2 do
4:         zamijeni A[1] i A[i]
5:         A.veličina-gomile ← A.veličina-gomile − 1
6:         MAKS-GOMILANJE(A, 1)
7:     end for
8: end procedure
```



Slika 4.8: Rad algoritma *Heapsort* [1].

varijabla i. Čvorovi koji su obojani žuto predstavljaju preostale čvorove gomile. U dijelu (k) prikazan je konačni rezultat.

4.8.5 Red prioriteta

Iako smo gomilu kao strukturu podataka u prethodnom dijelu koristili za algoritam *Heapsort*, ona je korisna i kao osnova za neke druge strukture podataka. U ovom dijelu opisan je *prioritetni red*. Prioritetni red kolekcija je podataka kod koje svaki element sadrži *ključ* (pored ostalih podataka) koji određuje prioritet ili "važnost" elementa. Na osnovi prioriteta elemenata utvrđuje se njihov poredak, odnosno koji je od njih prvi u redu. Prioritetni red može biti *min-prioritetni red* kod kojeg je prvi na redu element s najmanjim ključem ili *maks-prioritetni red* kod kojeg je prvi na redu element s najvećim ključem. *Maks-prioritetni red* podržava sljedeće operacije:

- MAKSIMUM(A) – vraća element s najvećim ključem.
- IZDVOJI-MAKSIMUM(A) – uklanja i vraća element s najvećim ključem.
- POVEĆAJ-KLJUČ(A, x, k) – postavlja ključ elementa *x* na *k* koji je veći ili jednak postojećem ključu elementa *x*.
- DODAJ(A, x) – dodaj element *x* u gomilu *A* ($A = A \cup \{x\}$).

S obzirom da je najveći element gomile uvijek na početnom indeksu niza, operacija MAKSIMUM trivijalna je i može se implementirati u vremenu $\Theta(1)$ kako je pokazano Algoritmom 4.15 [1]. U 2. liniji provjeravamo sadrži li gomila barem jedan element. Ako ne sadrži, signaliziramo grešku.

Algoritam 4.15 Najveći element gomile.

Parametri: A - gomila kao niz
Rezultat: element na prvom indeksu koji je ujedno i najveći element gomile

```
1: function MAKSIMUM(A)
2:     if A.veličina-gomile < 1 then
3:         greška "gomila je prazna"
4:     end if
5:     return A[1]
6: end function
```

Operacija IZDVOJI-MAKSIMUM (Algoritam 4.16 [1]) izdvaja element s najvećim ključem tako da prebaci posljednji element niza na njegovo mjesto i smanjuje podatak o veličini gomile za 1. S obzirom da će prebacivanje posljednjeg elementa niza na početni indeks poremetiti svojstvo gomile, ovaj algoritam u zadnjem koraku poziva algoritam MAKS-GOMILANJE koji će ponovo uspostaviti svojstvo gomile. Vremenska složenost ovog algoritma ovisi isključivo o vremenskoj složenosti operacije MAKS-GOMILANJE jer sve su operacije prije nje (redovi ..7) konstantne vremenske složenosti. Prema tome, vremenska složenost operacije IZDVOJI-MAKSIMUM je $O(log_2 n)$.

Algoritam 4.16 Izdvajanje i vraćanje maksimuma gomile.

Parametri: A - gomila kao niz
Rezultat: Najveći element gomile

```
1: function IZDVOJI-MAKSIMUM(A)
2:     if A.veličina-gomile < 1 then
3:         error "gomila je prazna"
4:     end if
5:     maks ← A[1]
6:     A[1] ← A[A.veličina-gomile]
7:     A.veličina-gomile ← A.veličina-gomile − 1
8:     MAKS-GOMILANJE(A, 1)
9:     return maks
10: end function
```

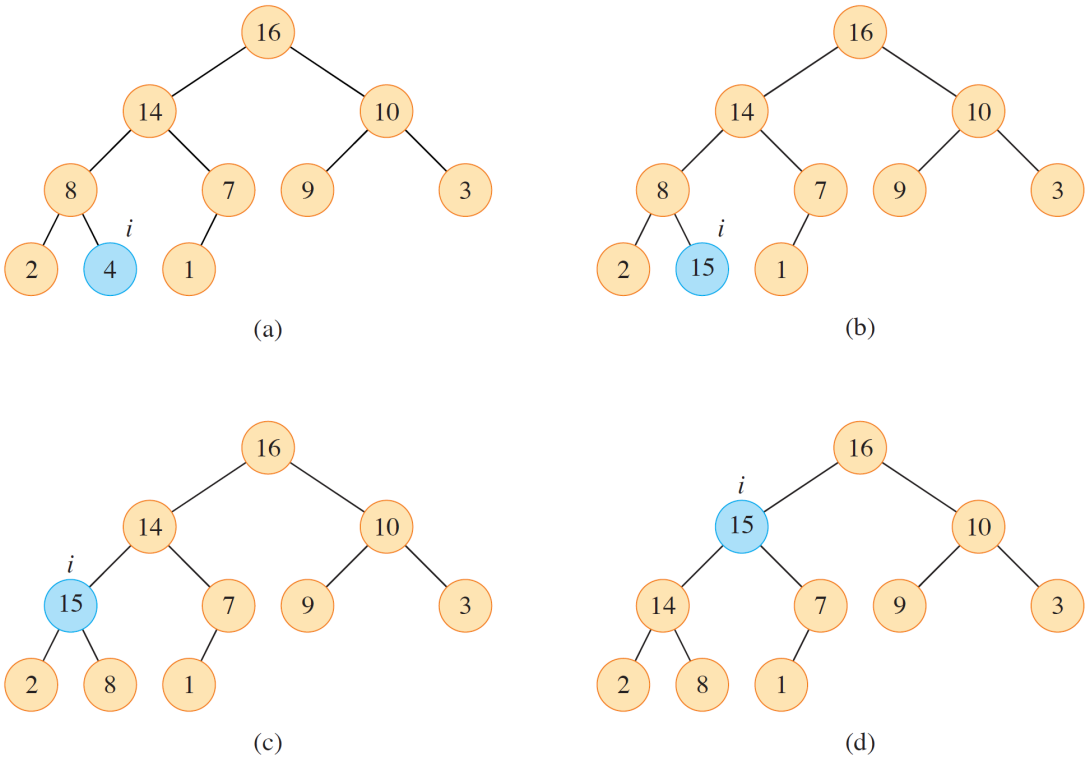
Operacija POVEĆAJ-KLJUČ povećava ključ postojećeg elementa x na vrijednost k (Algoritam 4.17 [1]). Ovaj algoritam ne koristi algoritam MAKS-GOMILANJE nego nakon povećanja ključa uspoređuje novi ključ elementa x koji se nalazi na indeksu i s ključem roditelja, odnosno elementom RODITELJ(i) i, ako x ima veći ključ, zamijeni mjesta tih dvaju elemenata. Ovaj postupak ponavlja s roditeljem na idućoj razini sve dok je potrebna zamjena ili dok ne dođe do korijena, odnosno indeksa 1. Na ovaj način element s novim ključem "podiže" se na odgovarajuće mjesto u stablu pa je zbog toga vremenska složenost ovog algoritma $O(log_2 n)$ u najgorem slučaju.

Algoritam 4.17 Povećavanje ključa gomile.

Parametri: A - gomila kao niz; x - vrijednost čiji ključ povećavamo; k - novi ključ

```
1: procedure POVEĆAJ-KLJUČ(A, x, k)
2:     if k < x.ključ then
3:         greška "novi ključ ne smije biti manji od postojećeg"
4:     end if
5:     x.ključ ← k
6:     i ← index na kojem se nalazi x
7:     while i > 1 and A[RODITELJ(i)].ključ < A[i].ključ do
8:         zamijeni A[i] sa A[RODITELJ(i)]
9:         i ← RODITELJ(i)
10:    end while
11: end procedure
```

Rad ovog algoritma ilustriran je na Slici 4.9. Plavom bojom označen je element kojem se ključ povećava s 4 na 15 (slika (a)). S obzirom da roditelj tog elementa sadrži ključ koji je manji od 15 (slika (b)), odnosno 8, algoritam zamjenjuje položaje ovih dvaju elemenata čiji je rezultat prikazan na slici (c). Sada je roditelj novog ključa opet manji, 14, pa se zamjenjuju položaji ovih dvaju elementa (slika (d)). Nakon toga svojstvo je gomile



Slika 4.9: Rad Algoritma 4.17 [1].

ispunjeno čime algoritam završava s radom.

Operacija DODAJ (Algoritam 4.18 [1]) dodaje novi element u gomilu. Taj se element dodaje na kraj gomile (promatrajući je kao niz) s početnom vrijednošću ključa postavljenoj na $-\infty$. Time se novi element inicijalno dodaje na kraj stabla (kao list), a zatim se poziva algoritam POVEĆAJ-KLJUČ koji će mu povećati ključ, odnosno postaviti ga na zadanu vrijednost i postaviti taj element na odgovarajuće mjesto u gomili. Upravo zbog korištenja algoritma POVEĆAJ-KLJUČ početna vrijednost ključa novog elementa postavlja se na $-\infty$.

Algoritam 4.18 Dodavanje elementa u gomilu.

Parametri: A - gomila kao niz; x - novi element

```
1:  procedure DODAJ(A, x)
2:      if A.veličina-gomile = A.veličina-niza then
3:          greška "novi element ne stane u gomilu"
4:      end if
5:      A.veličina-gomile ← A.veličina-gomile + 1
6:      k ← x.ključ
7:      x.ključ = −∞
8:      A[A.veličina-gomile] = x
9:      POVEĆAJ-KLJUČ(A, x, k)
10: end procedure
```

S obzirom da se i ovdje dodani element podiže prema vrhu stabla, vremenska složenost operacije DODAJ je $O(log_2 n)$.

Iako bismo prioritetni red mogli implementirati i kao sortirani niz, time bismo izgubili na performansama nekih operacija. Primjerice, dodavanje novog elementa zahtijevalo bi pomicanje elemenata koji slijede čime bi $O(log_2 n)$ složenost porasla na $O(n)$. Isto tako, operacija uklanjanja najvećeg ključa zahtijevala bi pomicanje svih elemenata desno od njega za jedno mjesto ulijevo što bi također pogoršalo vremensku složenost ove operacije na $O(n)$.

POGLAVLJE 5

Strukture podataka

5.1 Tip podatka

Podaci mogu biti primitivni, cijeli ili decimalni brojevi, znakovi kao slova ili znamenke te logičke vrijednosti kao *istinito* (engl. *true*) i *lažno* i *neistinito* (engl. *false*). Pored toga, podatke karakteriziraju i operacije koje su nad njima definirane. Primjerice, operacije kao što su zbrajanje, množenje i dijeljenje definirane su za cijelobrojni tip podatka. Isto tako, znakovni niz (engl. *string*) često podržava operaciju spajanja, ali ne i, primjerice, operaciju dijeljenja. Specifikacija vrste podatka kao što su cijeli broj ili znak, zajedno s pripadajućim operacijama naziva se *apstraktnim tipom podatka* (engl. *abstract data type* ili skraćeno *ADT*) [13]. Apstraktan tip podatka može se sastojati i od više jednostavnih tipova podataka. Primjerice, datum promatramo kao jedan podatak iako se on sastoji od tri cijela broja za dan, mjesec i godinu. Često ćemo umjesto naziva *apstraktan tip podatka* pisati samo *tip podatka*.

Definicija novog tipa podatka. Uzevši datum kao primjer, novi tip podatka za datum možemo u pseudokodu definirati kako je prikazano u definiciji tipa 5.1.

Tip 5.1 Definicija apstraktnog tipa podatka za datum koja se sastoji samo od podatkovnih članova.

```
1: type DATUM
2:     DAN ← 1
3:     MJESEC ← 1
4:     GODINA ← 1900
5: end type
```

Ovim smo definirali novi tip podatka, DATUM. Ovaj se tip podatka sastoji od tri podatkovna *člana*: DAN, MJESEC i GODINA. Ovakve članove nekog tipa podatka nazivat ćemo *atributima*. Imena članova tipa podatka pisat ćemo smanjenim velikim slovima da su lakše vidljivi u pseudokodu. Nadalje, atributima može biti pridružena početna vrijednost kao što je pokazano za atribute DAN, MJESEC i GODINA.

Tip podatka DATUM sastoji se samo od podatkovnih članova odnosno atributa. Tip 5.2 proširuje ovaj tip podatka s dvije operacije: ZADNJI-DAN-MJESECA i DODAJ-DANE. Operacije će uvijek nakon naziva imati popis parametara u okruglim zagradama (koji može biti i prazan).

Tip 5.2 Definicija apstraktnog tipa podatka za datum koja sadrži i podatke i operacije.

```
1: type DATUM
2:     ZADNJI-DAN-MJESECA()    ▷ vraća TRUE ako je DAN posljednji dan u mjesecu
3:     DODAJ-DANE(N)                                  ▷ dodaje datumu n dana
4:     DAN ← 1
5:     MJESEC ← 1
6:     GODINA ← 1900
7: end type
```

I operacije se računaju kao članovi tipa tako da tip DATUM sada ima pet članova.

Instanca tipa. Instanciranjem tipa podatka dobijamo *objekt* tog tipa. Tip podatka ne predstavlja nekakvu vrijednost. On je samo "nacrt" koji specificira što, odnosno koji atributi i operacije svaka *vrijednost* tog tipa sadrži. Kreiranje objekta nekog tipa podatka naziva se *instanciranjem* tog tipa. Kada kažemo da je neki objekt instanca određenog tipa to znači da se on sastoji od operacija i atributa definiranih za taj tip, s tim da svaki objekt ima svoju kopiju tih atributa.[1] Drugim riječima, pretpostavit ćemo da dvije instance istog tipa imaju svaka svoju kopiju podatkovnih članova tog tipa i njihovih vrijednosti. Ovdje možemo upotrijebiti analogiju s formularima ili obrascima: svaki obrazac ima broj ili ime i sadrži određeni skup polja (atributa) koja treba ispuniti. Broj ili ime tog obrasca predstavljaju tip obrasca, a svaki prazni ili ispunjeni obrazac s tim brojem ili imenom predstavlja jednu instancu tog obrasca. Nadalje, dva obrasca s istim brojem ili imenom, odnosno dvije instance istog obrasca mogu sadržavati različite podatke iako se radi o istom tipu obrasca.

Pristup članovima tipa preko instance. Atributima instance nekog tipa pristupat ćemo sintaksom `<instanca>.<atribut>`. Primjerice, ako preko instance tipa DATUM koja je pridružena varijabli *x* želimo pročitati vrijednosti atributa DAN pisat ćemo x.DAN. Isto tako, da bismo pridružili novu vrijednost 25 atributu DAN instance x pisali bi x.DAN $\leftarrow 25$. Na isti ćemo način pristupati operacijama definiranim za neki tip podatka. Primjerice, funkciju DODAJ-DANE možemo pozvati sa x.DODAJ-DANE(1).

Definiranje operacija nad tipom. Možemo primijetiti da operacijama ZADNJI-DAN-MJESECA i DODAJ-DANE nije pridružen nikakav opis postupka po kojem one rade. Takav ćemo opis pisati odvojeno od same definicije tipa podatka. Primjerice, za definiciju postupka operacije DODAJ-DANE pisat ćemo

procedure DATUM.DODAJ-DANE(N) ...

prefiks DATUM. Kod definicije nove funkcije ili procedure označavat funkciju ili proceduru koja se odnosi na operaciju koja pripada tipu DATUM. Za svaku operaciju nekog tipa

[1] U nekim slučajevima, atributi mogu dijeljeni među objektima istog tipa (npr. statički članovi klase u programskom jeziku C++), no to svojstvo neće biti razmatrano i korišteno u ovom udžbeniku.

podatka pretpostavit ćemo da ima izravan pristup svim ostalim članovima tog tipa. Primjerice, operacija ZADNJI-DAN-MJESECA može biti definirana kao u Algoritmu 5.1. Vidimo da se atributu DAN unutar operacije ZADNJI-DAN-MJESECA pristupa izravno, bez pisanja prefiksa DATUM.

Algoritam 5.1 Operacija ZADNJI-DAN-MJESECA.

```
1: function DATUM.ZADNJI-DAN-MJESECA()
2:     if DAN je zadnji dan u mjesecu then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end function
```

5.2 Struktura podataka

U ovom poglavlju govorimo o načinu rada sa skupinama podataka u kojima su pojedinačni podaci organizirani i spremljeni na neki određeni način. Takve skupine podataka nazivat ćemo *strukturama podataka* (engl. *data structures*) i tretirat ćemo ih kao zasebne tipove podataka od kojih su neki izravno podržani u samom računalu (kao nizovi), a drugi se moraju eksplicitno definirati.

Struktura podataka sastojat će se od skupine podataka i operacija koje ta struktura podataka podržava. Iako će te operacije biti realizirane drugačijim algoritmima ovisno o specifičnoj strukturi podataka, sve strukture podataka podržavat će barem sljedeće tri operacije:

- *Pronalaženje* elementa x u skupini S. Rezultat ovakve operacije obično će biti indeks ili adresa elementa.
- *Dodavanje* novog elementa x u skupinu S.
- *Brisanje* postojećeg elementa x iz skupine S.

Ove operacije razlikovat će se od jedne strukture podataka do druge po vremenskoj složenosti. S obzirom da način pronalaženja, dodavanja i brisanja podataka ovisi o načinu na koji su ti podaci organizirani u memoriji računala, vremenska složenost ovih operacija jedna je od karakteristika ovakvih tipova podataka po kojoj se oni razlikuju. Primjerice, jedna struktura podataka može podržavati učinkovito dodavanje novih elemenata, ali neučinkovit pristup elementima putem numeričkog indeksa. Isto tako, neke će strukture podataka podržavati neke operacije koje druge neće.
U ovom poglavlju obrađene su sljedeće strukture podataka:

- niz
- povezana lista

- stog
- red
- stablo
- tablicu raspršivanja.

Za sve ključne operacije ovih struktura podataka bit će opisani algoritmi po kojima one rade i njihova vremenska složenost.



Slika 5.1: Primjer niza znakova koji sadrži tekst "ABC". Okomite crte označavaju segmente memorije od 16 bitova iznad kojih se nalazi njihova adresa. Adresa niza je 102 (prvo slovo) i svako slovo je veličine 2 bajta ili 16 bita. Cijeli niz zauzima 6 bajtova i proteže se od adrese 102 do 106.

5.3 Niz

Niz (engl. *array*) temeljna je struktura podataka koja se sastoji od elemenata smještenih na susjednim memorijskim lokacijama. Jedna važna karakteristika nizova je da za njih postoji izravna podrška u samom računalu. Naime, memorija računala može se promatrati kao niz memorijskih lokacija gdje je svaka od njih određena jedinstvenom adresom. Procesor računala pomoću posebnih instrukcija može preko tih adresa vrlo učinkovito pristupati tim memorijskim lokacijama, a time i elementima niza. Nizovi su zbog toga idealna struktura podataka za slučajeve kada nam treba brz pristup podacima pomoću numeričkog indeksa.

Kao struktura podataka, za nizove se obično definiraju sljedeće operacije:

- **Indeksiranje**: pristup elementima putem cjelobrojnog indeksa.
- **Dodavanje**: dodavanje novog elementa na određeno mjesto u nizu.
- **Brisanje**: brisanje indeksom određenog elementa niza.
- **Pretraživanje**: pronalaženje tražene vrijednosti unutar niza.

Za svaki se niz mora znati adresa njegovog prvog elementa koja predstavlja početnu adresu niza. Evidenciju o adresi vodi sam prevoditelj ili interpreter programskog jezika. Na Slici 5.1 prikazan je primjer jednog niza. U nastavku ovaj ćemo niz i njegovu početnu adresu označavati s A.

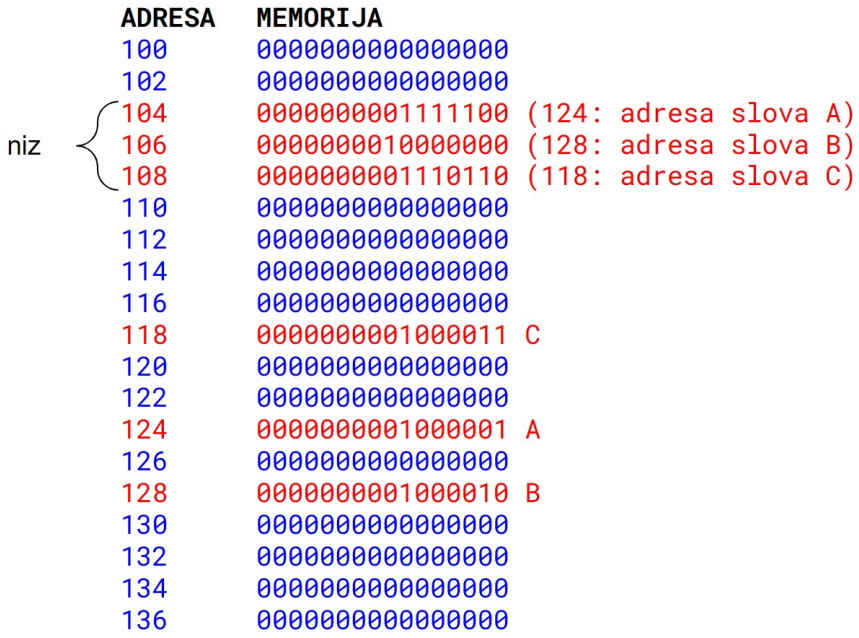
5.3.1 Indeksiranje niza

S obzirom da prevoditelj ili interpreter vodi računa o tome na kojoj adresi počinje svaki niz, indeksiranje niza vrlo je jednostavna i učinkovita operacija. Da bi se odredila adresa indeksiranog elementa, početnoj se adresi jednostavno doda indeks. Ako A sadrži adresu prvog člana tog niza, element niza na indeksu i računa se po formuli

$$A[i] = A + ib$$

gdje je b veličina elementa u bajtovima[2]. Ovdje pretpostavljamo da je niz *homogen*, odnosno da su svi elementi niza istog tipa te da se prvi element nalazi na indeksu 0. Primjerice, adresa niza je 102, a tip $char$ veličine 1 bajt (kao što je u jezicima C i C++). Za niz na Slici 5.1 adresa znaka na ideksu 2 bila bi $A[2] = 102 + 2 * 2 = 106$ jer je adresa niza 102, indeks je 2, a veličina elementa 2 bajta. Isto tako nam $A[0] = 102 + 0 * 2 = 102$ daje prvi element tog niza.

[2] 1 bajt = 8 bitova.



Slika 5.2: Primjer niza koji sadrži adrese elemenata veličine 16 bitova. Radi jednostavnosti adrese su također 16-bitne. Segment adresa niza je od adrese 104 do 108, ali dodatni memorijski prostor na adresama 118, 124 i 128 upotrebljen je za sama slova.

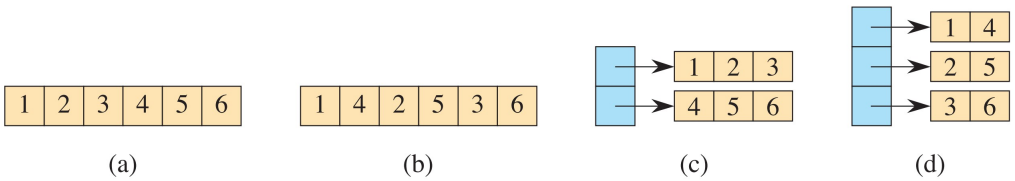
Kod *heterogenih* nizova [18] , odnosno onih koji mogu sadržavati elemente različitog tipa, niz sadrži pokazivače (adrese) na elemenate, a s obzirom da je pokazivač uvijek iste veličine gornja formula primjenjiva je i u takvim slučajevima. I homogeni nizovi mogu sadržavati pokazivače na elemente, ovisno o programskom jeziku. Slika 5.2 prikazuje niz koji sadrži adrese elemenata. Ovaj niz zauzima adrese od 104 do 108. Međutim, umjesto koda znaka na njima se nalaze adrese znaka. Primjerice, na adresi 108 (indeks 2) nalazi se adresa slova C, odnosno 118, umjesto koda tog slova. Poredak adresa (124, 128, 118) odgovara poretku slova teksta ABC. Dakle, indeksiranjem ovog niza i dalje ćemo dobiti ispravno slovo na svakom indeksu, ali neizravno, preko adrese tog slova. Možemo primijetiti da sama slova ne moraju nužno biti smještena u memoriji u odgovarajućem poretku. Važno je da su adrese koje predstavljaju sadržaj niza (na segmentu od adrese 104 do 108) smještene u odgovarajućem poretku.

Operacija indeksiranja niza prikazana je Algoritmom 5.2. Funkcija INDEKS vraća adresu elementa na indeksu i. Veličina elementa dobiva se na osnovu veličine elementa i adrese niza.

Algoritam 5.2 Adresa elementa na indeksu i.

Parametri: A - niz s elementima tipa T; i - indeks niza
Rezultat: adresa elementa na indeksu i

```
1: function INDEKS(A : T[], i : INT)
2:     return A + i * VELIČINA(T)
3: end function
```



Slika 5.3: Konceptualni prikaz matrice 5.1. Na slici (a) elementi su poslagani po redcima, a na slici (b) po stupcima. Na slici (c) elementi su poslagani po redcima, ali tako da niz sadrži pokazivač na svaki redak. Na slici (d) elementi su poslagani po stupcima, a niz sadrži pokazivače na svaki stupac [1].

U programskim jezicima niz obično je ugrađena struktura podataka za koju nije eksplicitno definiran tip podatka.

Višedimenzionalni nizovi

Mnogi programski jezici koji podržavaju rad s nizovima omogućuju više dimenzija. Takve nizove nazivamo *višedimenzionalnim nizovima* (engl. *multidimensional arrays*). Mogu se smjestiti u memoriju na isti način kao i jednodimenzionalni nizovi. Kod računanja adrese elementa na osnovu indeksa moramo uzeti u obzir raspored elemenata. Primjerice, dvodimenzionalni niz možemo smjestiti u memoriju na dva načina:

- po redcima (engl. *row-major order*)
- po stupcima (engl. *column-major order*).

Primjer višedimenzionalnog niza je *matrica* (engl. *matrix*) prikazana u 5.1. Ova se matrica sastoji od dva redka s elementima $[1, 2, 3]$ i $[4, 5, 6]$ i tri stupca s elementima $[1, 4]$, $[2, 5]$ i $[3, 6]$.

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{5.1}$$

Za matricu veličine $r \times s$ (r redaka i s stupaca) i indeks $M[i, j]$ (redak i i stupac j) adresa elementa za dvodimenzionalni niz s početnim indeksom 0, strukturiran po redcima je

$$M[i, j] = M + (is + j)b \tag{5.2}$$

a za niz strukturiran po stupcima

$$M[i, j] = M + (jr + i)b \tag{5.3}$$

gdje je b veličina elementa u bajtovima (matrica je homogena struktura podataka). Na Slici 5.4 prikazana je matrica 5.1 smještena u memoriji na adresi 108. Primjerice, adresa elementa u redku 1 i stupcu 1 (element 5) je 116, odnosno $M[1, 1] = 108 + (1 \times 3 + 1) \times 2 = 116$. Za matricu smještenu po stupcima za isti element 5 imamo $M[1, 1] = 108 + (1 \times 2 + 1) \times 2 = 114$.

ADRESA	MEMORIJA	
104	0000000000000000	
106	0000000000000000	
108	0000000000000001	(1)
110	0000000000000010	(2)
112	0000000000000011	(3)
114	0000000000000100	(4)
116	0000000000000101	(5)
118	0000000000000110	(6)
120	0000000000000000	
122	0000000000000000	

(a)

ADRESA	MEMORIJA	
104	0000000000000000	
106	0000000000000000	
108	0000000000000001	(1)
110	0000000000000100	(4)
112	0000000000000010	(2)
114	0000000000000101	(5)
116	0000000000000011	(3)
118	0000000000000110	(6)
120	0000000000000000	
122	0000000000000000	

(b)

Slika 5.4: Primjer matrice 5.1 u 16-bitnoj memoriji. Segment memorije koji zauzima matrica označen je crveno. Na slici (a) matrica je smještena po redcima, a na slici (b) po stupcima.

Ove formule možemo generalizirati na k-dimenzionalne nizove [19]. Za niz od k dimenzija strukturiran po redcima, adresu elementa na indeksu $i_1..i_k$ računamo po formuli

$$A[i_1..i_k] = A + ((...(i_1 n_2 + i_2)n_3 + i_3)...)n_k + i_k)b \tag{5.4}$$

gdje je n_j broj elemenata dimenzije j. Primjerice, za matricu 5.1 imamo $M[1,1] = M + (i_1 \times 3 + i_2) \times b = 108 + (1 * 3 + 1) * 2 = 116$.

Slika 5.5 prikazuje trodimenzionalni niz s dimenzijama $2 \times 3 \times 2$ i s adresama elemenata izračunatih pomoću formule 5.4. Primjerice, adresa elementa $A[0,2,1]$ je 118 jer je $108 + ((0 \times 3 + 2) \times 2 + 1) \times 2 = 118$.

Vidimo da je operacija indeksiranja niza vrlo učinkovita, odnosno odvija se u konstantnom vremenu jer ne ovisi o veličini niza. To znači da indeksiranje niza od 5 milijuna elemenata u načelu ne zahtijeva više vremena od indeksiranja niza od 5 elemenata. Prema tome, ako nam treba struktura podataka s brzim indeksiranjem, niz je dobar kandidat. Međutim, šta ako u neki niz želimo dodati novi ili iz njega ukloniti neki postojeći element? Performanse tih operacija nisu konstantne jer ovise o veličini niza, što ćemo vidjeti u sljedećem dijelu.

5.3.2 Dodavanje i brisanje elemenata iz niza

U slučaju dodavanja novog elementa u niz moramo uzeti u obzir mjesto dodavanja novog elementa, odnosno dodajemo li ga na kraj niza ili negdje drugdje. Ako novi element dodajemo negdje prije kraja niza, svi se elementi koji slijede moraju pomaknuti za jedan indeks dalje. Kod nizova s velikim brojem elemenata to može negativno utjecati na performanse dodavanja novog elementa. S istim se problemom suočavamo kod brisanja elementa koji se također nalazi negdje prije kraja niza, ali tada se svi elementi koji slijede moraju pomaknuti za jedan indeks prema početku niza. Za ovo pomicanje elemenata mora postojati nekakva operacija, neovisno o tome je li ona sastavni dio programskog jezika ili je moramo sami napisati. Učinkovitost te operacije ovisi o broju elemenata koje ona mora pomaknuti, što znači da se ne izvršava u konstantnom vremenu. Budući da se jedna takva operacija

```
ADRESA   MEMORIJA
104      0000000000000000
106      0000000000000000
108      0101000000000001 A[0, 0, 0]
110      0000000010000010 A[0, 0, 1]
112      0000100100000011 A[0, 1, 0]
114      0010000000000100 A[0, 1, 1]
116      0000000000000101 A[0, 2, 0]
118      0000100100100110 A[0, 2, 1]
120      0000000000000001 A[1, 0, 0]
122      0000011000000010 A[1, 0, 1]
124      0000001000111011 A[1, 1, 0]
126      0001011100110100 A[1, 1, 1]
128      0100010010000101 A[1, 2, 0]
130      1111111000000110 A[1, 2, 1]
132      0000000000000000
134      0000000000000000
```

Slika 5.5: Prikaz memorije koja sadrži trodimenzionalni niz s dimenzijama $2 \times 3 \times 2$ na adresi 108. Na svakoj adresi nalazi se 16-bitna vrijednost. Zelenom bojom označeni su indeksi dimenzija niza kojima se pristupa svakom tom elementu.

mora izvršavati kod svakog dodavanja ili brisanja elementa, lako je vidjeti da dodavanje i brisanje elemanata iz niza nije niti približno učinkovito kao indeksiranje.

Algoritmom 5.3 prikazano je dodavanje novog elementa u niz. Vidimo da Algoritam 5.3 ovisi o veličini niza jer mora proći sve elemente počevši od kraja pa do indeksa i pa mu je stoga vremenska složenost $O(n)$.

Algoritam 5.3 Dodavanje novog elementa u niz.

Parametri: A - niz elemenata tipa T; i - indeks na koji se postavlja novi element; x - novi element tipa T

```
1:  procedure DODAJ(A : T[], i : INT, x : T)
2:      if A.broj-elemenata = A.kapacitet-niza then
3:          greška "niz je pun"
4:      end if
5:      n ← A.broj-elemenata
6:      for p ← n − 1 downto i do
7:          A[p + 1] ← A[p]
8:      end for
9:      A[i] ← x
10:     A.broj-elemenata ← A.broj-elemenata + 1
11: end procedure
```

Operacija brisanja elementa iz niza je slična, samo što u ovom slučaju elemente koji se nalaze desno od onog koji brišemo pomičemo ulijevo, što je pokazano Algoritmom 5.4. Prema tome, vremenska složenost ove operacije je također $O(n)$.

Algoritam 5.4 Brisanje elementa iz niza.

Parametri: A - niz elemenata tipa T; i - indeks na kojem se nalazi element koji brišemo

```
1: procedure UKLONI(A : T[], i : INT)
2:     n ← A.broj-elemenata
3:     for p ← i..(n − 1) do
4:         A[p] ← A[p + 1]
5:     end for
6:     A.broj-elemenata ← A.broj-elemenata − 1
7: end procedure
```

Iz ovih primjera jasno je da niz kao struktura podataka nije idealan za česta dodavanja ili brisanja elemenata, osim ako se to ne događa na kraju niza. U tom slučaju vremenska složenost ovih operacija je $O(1)$.

5.3.3 Dinamički niz

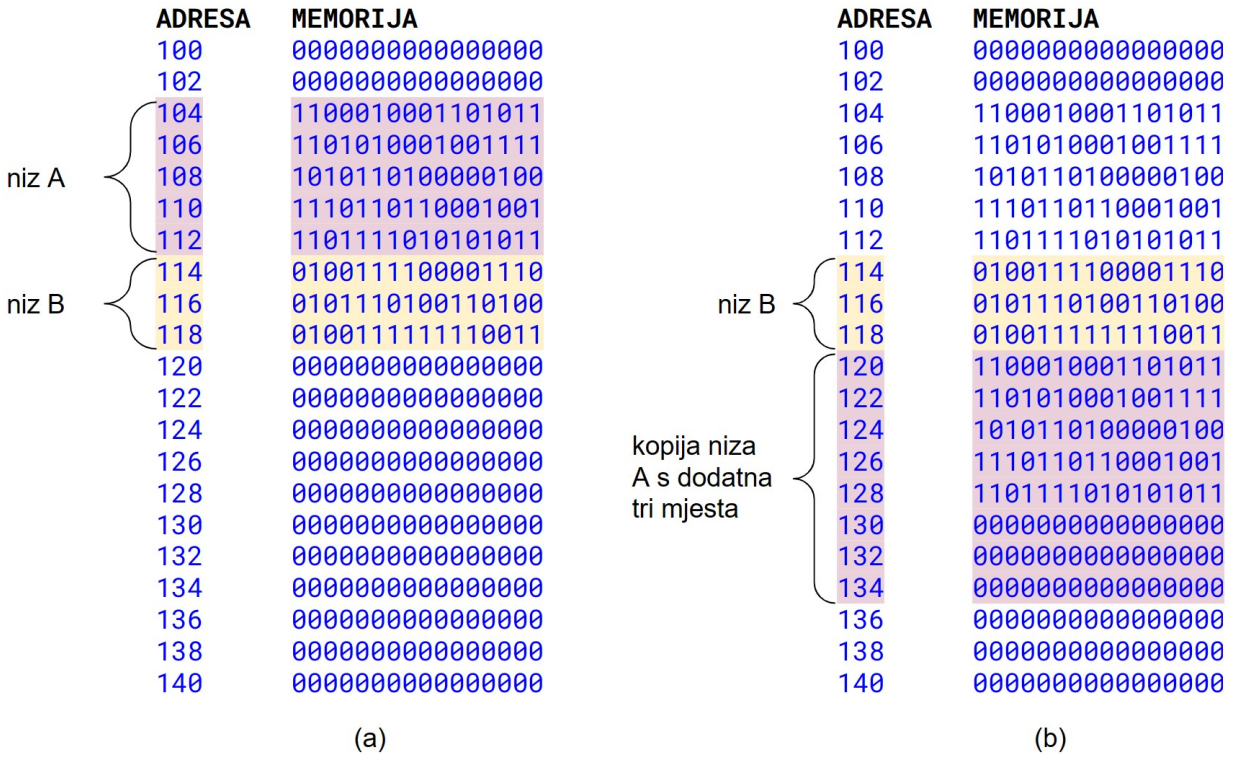
Niz je kompaktna struktura podataka i sastoji se od unaprijed zadanog broja memorijskih lokacija. To znači da ako su sve te lokacije popunjene elementima niza, nema prostora za dodavanje novih elemenata. Neki programski jezici podržavaju nizove koji su napravljeni tako da unaprijed rezerviraju jedan veći dio memorije, više nego što treba za početni broj elemenata. Time se ostavlja mogućnost proširenja niza novim elementima. Ako se taj niz popuni do kraja, novi se elementi ne mogu jednostavno dodati na kraj tog niza. Primjerice, na Slici 5.6 a) vidimo da elementi niza A zauzimaju adrese od 104 do 112. Niz B nalazi se odmah iza niza A, na adresama 114 do 118. Prema tome, novi element na kraj niza A ne možemo dodati iz dva razloga:

1. adresa 114 nije dio niza A
2. ako bismo niz A htjeli proširiti na kraju, ne bismo mogli jer je iduća adresa zauzeta, u ovom primjeru nizom B.

Zbog ovakvih slučajeva neki programski jezici podržavaju *dinamičke nizove* (engl. *dynamic array*).[3] Takvi nizovi, nakon što se popune, postojeće elemente kopiraju u novi, veći dio memorije. Ovaj proces prikazan je na Slici 5.6, gdje se u gornjem dijelu nalazi početni niz A od pet elemenata koji zauzimaju adrese od 104 do 110. U taj niz više ne možemo dodavati nove elemente jer nema mjesta. Stoga se elementi niza A kopiraju na adrese 120 do 134 gdje ima mjesta za dodatna tri elementa (od adrese 130 do 134). Nakon toga, adresa niza A više nije 104 nego 120. Novi se element sada može dodati iza posljednjeg, na adresu 130.

Problem s dinamičkim nizovima nastaje kada neka varijabla sadrži adresu elementa dinamičkog niza. U slučaju proširenja tog niza adresa svih elemenata će se promijeniti pa

[3] Jedan primjer je *vector* u jeziku C++, a drugi *lista* u jeziku Python.



Slika 5.6: Primjer dinamičkog niza. Adresa niza A je 104, a niza B 114. Slika (a): Niz A je popunjen i nema prostora za proširenje na istom segmentu memorije (odmah iza adrese 112) jer je taj dio memorije već zauzet (nizom B u ovom primjeru). Slika (b): Da bismo dodali novi element u niz A, elementi se kopiraju u veći prostor memorije (adrese 120 do 134) u kojem ima mjesta za tri dodatna elementa, počevši od adrese 130. Nakon kopiranja nova adresa niza A je 120.

će time i pokazivači koji sadrže stare adrese biti neispravni. Ovo se vidi na Slici 5.6. Ako pokazivač p sadrži adresu drugog elementa na slici (a), adresa 106, nakon proširenja niza taj će se element nalaziti na adresi 122, ali će p i dalje sadržavati staru adresu, 106, koja više nije dio tog niza i na kojoj se možda nalazi neki drugi podatak.

Upravljanje dinamičkim nizom

Postoji nekoliko strategija kojima prevoditelji i interpreteri upravljanjaju dinamičkim nizovima, iako one u načelu nisu vidljive programeru. Cilj tih strategija je da se niz premješta s jednog dijela memorije u drugi u što manjoj mjeri. Primjerice, ako svaki put popunnjeni taj premjestimo i povećamo za samo jedno mjesto, imat ćemo puno kopiranja elemenata s jednog segmenta memorije u drugi što bi negativno utjecalo na performanse dinamičkog niza. Iz tog razloga postoji nekoliko strategija proširenja (i smanjenja) dinamičkog niza:

- *Udvostručavanje*: Prilikom rezerviranja novog memorijskog prostora za niz, rezervira se dvostruko veći prostor.
- *Postepeno povećavanje*: Kod ovog pristupa rezervira se memorija povećana za neku konstantnu veličinu. Količina potrebne memorije može biti manja nego kod udvostručavanja.
- *Geometrijsko povećavanje*: Nova se memorija uveća za neki omjer u odnosu na veličinu niza, primjerice, za 1.25 puta.
- *Smanjivanje*: Cilj smanjivanja memorije rezervirane za dinamički niz je izbjegavanje nepotrebne potrošnje memorije. Primjerice, ako je niz popunjen ispod 30% smanjimo memoriju za 50% ili za veličinu za koju smo je prethodno povećali.
- *Hibridna strategija*: Koristi se više strategija, primjerice jedna za povećanje niza, a druga za smanjenje.

Neki programski jezici koji podržavaju dinamičke nizove su Python (gdje se takav niz zove "lista") i C++ (pomoću klase *vector* koja je dio standardne biblioteke).

5.3.4 Pretraživanje niza

U Algoritmu 2.4 vidjeli smo najjednostavniji način pretraživanja niza - linearno pretraživanje. Takvo pretraživanje pretpostavlja da elementi niza nisu uređeni ni po kakvom određenom redoslijedu. Stoga ne možemo pretpostaviti gdje se u nizu može nalaziti traženi element, čime bismo izbjegli ispitivanje svakog indeksa tog niza i poboljšali performanse ovakvog algoritma. Jedan od načina uređivanja elemenata niza za učinkovitije pretraživanje je njihovo sortiranje po veličini. Sortiranje u ovom kontekstu ne uključuje samo brojeve, nego bilo koji tip vrijednosti koje možemo postaviti u nekakav njima prirodni poredak. primjer, tekstualne vrijednosti možemo sortirati po abecedi. Ako pretražujemo niz u kojem su vrijednosti sortirane, očito poboljšanje je prekinuti pretraživanje čim dođemo do prvog elementa u nizu koji je veći od onog koji tražimo (za slučaj gdje traženi element ne postoji). U Algoritmu 5.5 pretpostavljamo da su elementi u zadanom nizu sortirani u rastućem poretku.

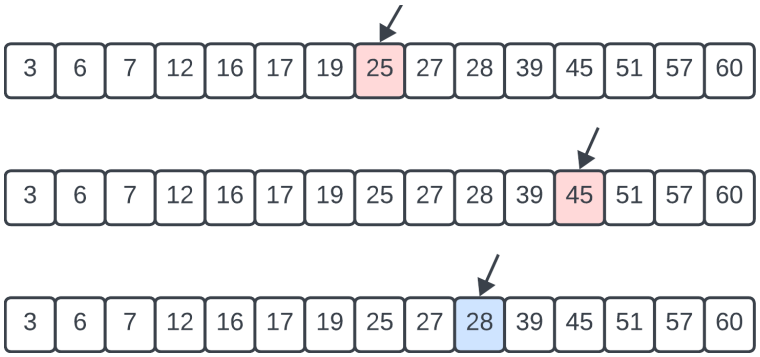
Algoritam 5.5 Pretraživanje sortiranog niza.

Parametri: A - sortirani niz koji pretražujemo; x - vrijednost koju tražimo
Rezultat: indeks na kojem se nalazi traženi element ili -1 ako on ne postoji u nizu

```
1: function PRETRAŽIVANJE-SORTIRANOG-NIZA(A, x)
2:     i ← 1
3:     while i < A.broj-elemenata and A[i] ≤ x do
4:         if A[i] = x then
5:             return i
6:         end if
7:         i ← i + 1
8:     end while
9:     return -1
10: end function
```

Iako je ova operacija poboljšanje u odnosu na Algoritam 2.4, činjenica da su elementi sortirani omogućava nam značajno bolju tehniku pretraživanja koja se zove *binarno pretraživanje* [4]. Ova tehnika pretraživanja temelji se na jednostavnom zapažanju da ako počnemo tražiti od bilo kojeg indeksa u sortiranom nizu, uvijek znamo nalazi li se traženi element negdje prije ili poslije tog indeksa. Ovo je slično postupku po kojem tražimo određenu stranicu u nekoj knjizi. Ako tražimo stranicu 720, a knjigu smo otvorili na stranici 521, moramo nastaviti s traženjem samo među ostatkom stranica, a ne prije toga, jer znamo da se stranica 720 ne može nalaziti prije stranice 521. Na ovaj način u svakom koraku preskačemo velik broj stranica dok ne dođemo do one koju tražimo. Na sličan način možemo definirati operaciju pretraživanja niza, pod pretpostavkom da su njegovi elementi sortirani. Ako zamislimo da su elementi niza sortirani od manjeg prema većem, ova operacija počinje od sredine niza gdje ispita element na tom indeksu. Ako to nije element koji tražimo, promatra se nalazi li se traženi element lijevo ili desno od tog elementa. Ovaj se postupak ponavlja počevši od indeksa u sredini lijevog ili desnog segmenta niza u odnosu na taj element. Na ovaj način svaki put preskačemo pola preostalih elemenata. Postupak je prikazan na Slici 5.7. Na slici (a) počinjemo na polovici niza, gdje se nalazi broj 25. S obzirom da je 28 veći od 25, nastavljamo s desnom polovicom na čijoj se sredini nalazi broj 45, što je prikazano na slici (b). Broj 28 manji je od 45, pa nastavljamo na lijevoj polovici prethodne polovice, gdje dolazimo do traženog broja 28 (slika (c)). Da smo tražili broj 40 ovim bismo postupkom došli do broja 39 (jer je 40 manji od 45, a veći od 28) i tu bismo stali. Došli bismo do segmenta koji više ne možemo rastaviti na lijevu i desnu stranu jer sesastoji od samo jednog elementa. Ovo je pokazano na slici (d).

U ovom postupku vidimo da smo za traženi element 28 morali ispitati svega tri elementa od njih petnaest: 25, 45 i 28 (slike (a), (b) i (c)). Ako bismo, primjerice, tražili broj 40 koji ne postoji, morali bismo ispitati samo četiri elementa. Posljednji ispitani element bio bi broj 39 (slika (d)), za razliku od svih petnaest elemenata koje bismo morali ispitati linearnim pretraživanjem.
Detalji ovog postupka prikazani su Algoritmom 5.6. Varijable *l* i *d* određuju segment niza na čijoj sredini ispitujemo element. Način na koji ona radi prikazan je na Slici 5.8 gdje



Slika 5.7: Binarno pretraživanje niza od 15 elemenata u kojem tražimo broj 28.

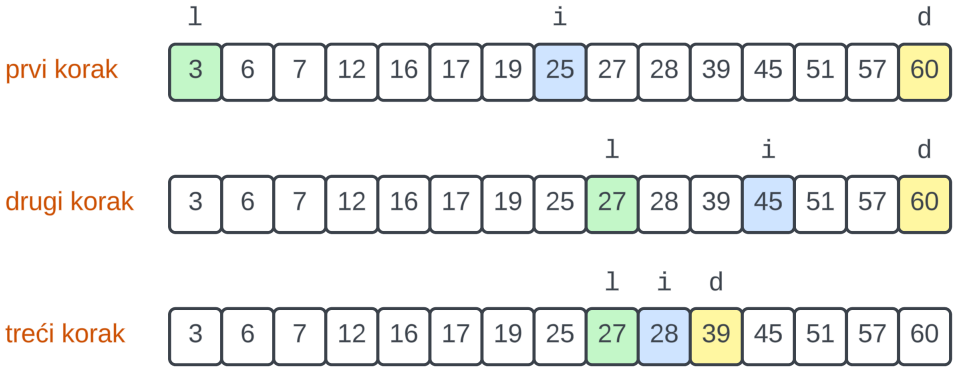
tražimo broj 28. U prvom koraku počinjemo na sredini niza. S obzirom da je 28 veći od 25 moramo nastaviti s desnom stranom niza, pa se prema tome izvršava naredba $l \leftarrow i + 1$ te je sada novi segment onaj pokazan u drugom koraku. Indeks i je sada na vrijednosti 45 koja je na sredini tog segmenta i veća je od 28. To znači da moramo prijeći na lijevi segment koji se nalazi između l i $i - 1$, odnosno izvršava se naredba $d \leftarrow i - 1$ i dobivamo segment pokazan u trećem koraku. Na sredini ovog segmenta nalazi se tražena vrijednost 28.

Algoritam 5.6 Binarno pretraživanje niza.

Parametri: A - sortirani niz koji pretražujemo; x - vrijednost koju tražimo
Rezultat: indeks na kojem se nalazi traženi element ili -1 ako on ne postoji u nizu

```
1:  function BINARNO-PRETRAŽIVANJE(A, x)
2:      l ← 0
3:      d ← A.broj-elemenata
4:      i ← ⌊A.broj-elemenata/2⌋
5:      while l ≤ d do
6:          if A[i] = x then
7:              return i
8:          else
9:              if x < A[i] then
10:                 d ← i − 1
11:             else
12:                 l ← i + 1
13:             end if
14:             i ← ⌊(l + d)/2⌋
15:         end if
16:     end while
17:     return -1
18: end function
```

Jedno od svojstava binarnog pretraživanja je da za niz od *n* vrijednosti treba ispitati najviše $log_2 n + 1$ vrijednosti. To znači da, primjerice, za niz koji sadrži milijardu vrijednosti (pod uvjetom da je taj niz sortiran) nikada ne moramo ispitati više od 30 vrijednosti, što



Slika 5.8: Rad postupka BINARNO-PRETRAŽIVANJE.

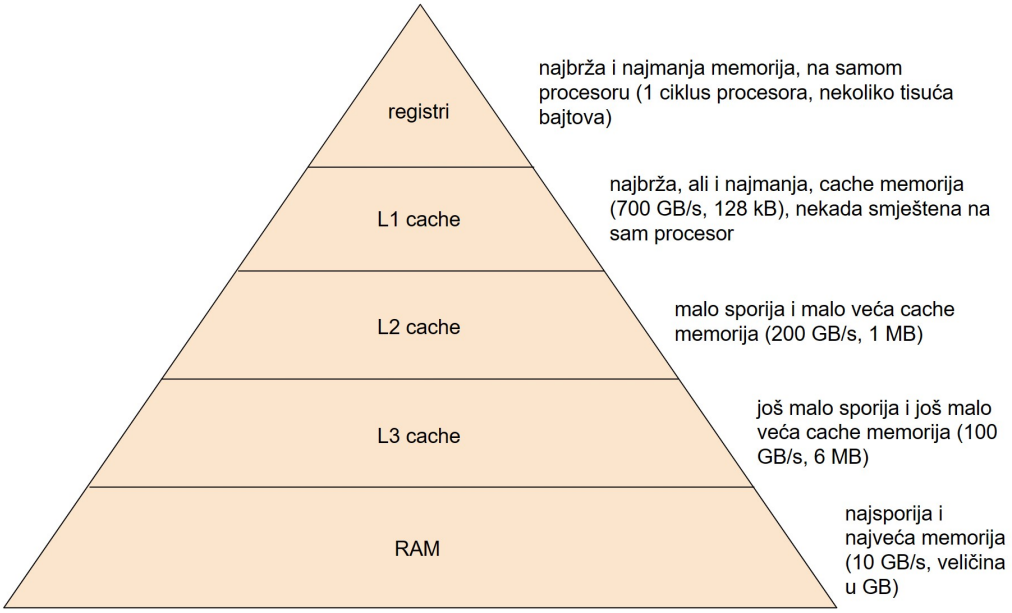
je drastična razlika u odnosu na jednostavno linearno pretraživanje. Ovo se može činiti kao mali broj za toliko vrijednosti, ali ako uzmemo u obzir da kod svakog neuspješnog uspoređivanja vrijednosti njihov broj praktički prepolovimo, lako se vidi da u svakom koraku odbacujemo velik broj vrijednosti, odnosno pola od broja vrijednosti u prethodnom koraku. U slučaju milijarde vrijednosti, nakon prvog koraka taj bi broj pao na $500.000.000$, nakon drugog na $125.000.000$, trećeg na $62.500.000$, i tako dalje. Ovim postupkom bismo u najgorem slučaju imali 30 usporedbi vrijednosti. Vremenska složenost binarnog pretraživanja je $O(log_2 n)$ u najgorem slučaju. Ako usporedimo rast funkcija $f(n) = n$ (linearno pretraživanje) i $f(n) = log_2 n$ (binarno pretraživanje) možemo vidjeti da prva funkcija, koja karakterizira linearno pretraživanje, raste znatno brže od druge. Vrijeme potrebno za pronalaženje nekog elementa linearnim pretraživanjem raste brže s povećanjem broja elemenata u nizu. Iz ovoga možemo zaključiti da binarno pretraživanje ima znatno bolje performanse od linearnog. Ovdje je važno ponovo istaknuti da kada se radi o pretraživanju niza, binarno se pretraživanje može koristiti samo ako su vrijednosti u tom nizu sortirane po veličini.

5.3.5 Načelo lokaliteta

Niz kao kompaktna struktura podataka pored učinkovitog indeksiranja ima još jednu prednost u odnosu na neke druge strukture podataka: dobro iskorištava *načelo lokaliteta* (engl. *locality principle*) [20, 21]. Ono kaže da programi pristupaju relativno malom broju memorijskih lokacija u svakom trenutku. Postoje dvije vrste lokaliteta [22]:

- *Temporalni lokalitet*: ako je program pristupio nekoj adresi, uskoro će opet pristupiti toj adresi.
- *Prostorni lokalitet*: ako je program pristupio nekoj adresi, uskoro će pristupiti i obližnjim adresama.

Prostorni lokalitet posebno se odnosi na nizove. Da bi se iskoristilo načelo lokaliteta memorija računala organizirana je hijerarhijski [22]. *Hijerarhija memorije* (engl. *memory hierarchy*) je struktura memorije koja koristi više razina memorije različitih brzina i veličina. Što je određena razina memorije dalje od procesora, veća je, ali i sporija. Primjer takve hijerarhije prikazan je na Slici 5.9. Najbrža razina memorije registri su procesora, ali njih je i najmanje. Cache-memorija brža je od radne, RAM memorije i podijeljena je u

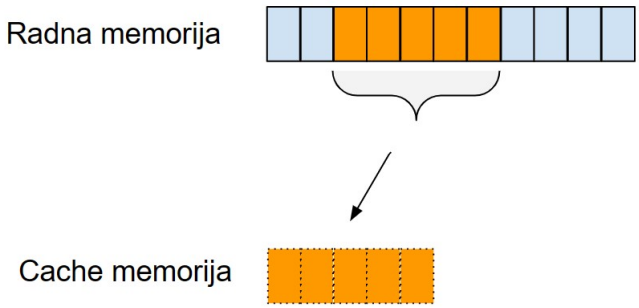


Slika 5.9: Primjer hijerarhije memorije.

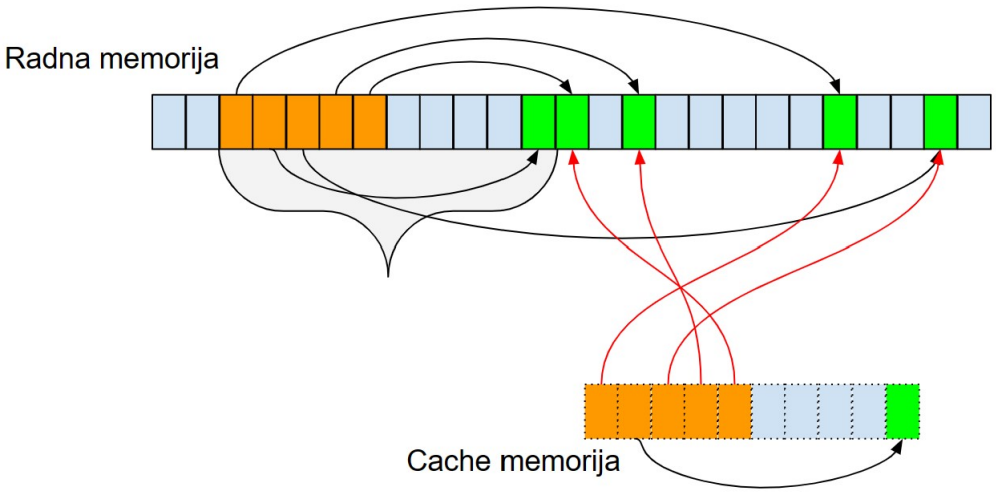
tri razine: L1 kao i najbržu i najmanju takvu memoriju, L2 u sredini te L3 kao najsporiju i najveću cache-memoriju. Na toj slici prikazana je brzina i veličina cache-memorije jednog konkretnog procesora. Radna memorija daleko je najveća, danas često oko 32 GB, ali s obzirom da se obično nalazi na matičnoj ploči, daleko od procesora, pristup njoj je i najsporiji. Radna memorija je otprilike 10 do 100 puta sporija od cache-memorije [22].

Na koji način nizovi iskorištavaju načelo lokaliteta? Najvažnija karakteristika niza s tog gledišta je činjenica da su elementi u njemu smješteni na uzastopnim memorijskim lokacijama. Prema načelu prostornog lokaliteta, ako pristupimo jednom indeksu niza, pristupit ćemo u kratkom vremenu i obližnjim indeksima. Primjerice, ako pristupimo indeksu 50, pristupit ćemo i indeksima 40 i 60. Procesor računala izrađen je tako da u ovom slučaju u cache-memoriju kopira ne samo indeks 50 nego i jedan dio niza prije i poslije tog indeksa, kao što je je prikazano na Slici 5.10. Procesor može cijeli blok susjednih memorijskih lokacija, odnosno jedan segment niza ili cijeli niz, prenijeti u cache-memoriju kada procesor pristupa elementu niza na nekoj memorijskoj lokaciji u radnoj memoriji. Sljedeći pristupi elementima tog niza odvijaju se u cache-memoriji, što je znatno brže od pristupa radnoj memoriji.

Važi li isto i za nizove koji sadrže adrese elemenata umjesto samih elemenata? Slika 5.11 prikazuje niz (narančastom bojom) koji sadrži adrese elemenata (obojenih zeleno). S obzirom da niz sadrži adrese, ako ga procesor prenese u cache-memoriju tamo će se naći samo adrese elemenata, ali sami elementi (iako možda ne svi) ostat će u glavnoj memoriji. Ovdje se djelomično može iskoristiti načelo prostornog lokaliteta jer će pristup adresama elemenata (indeksima niza) biti brz (jer ide kroz cache-memoriju), ali će pristup samim elementima ići kroz glavnu memoriju i stoga biti znatno sporiji.



Slika 5.10: Primjer primjene načela prostornog lokaliteta nad nizom.



Slika 5.11: Primjer primjene načela prostornog lokaliteta nad nizom koji sadrži adrese elemenata. Niz je označen narančastom bojom, a elementi zelenom.

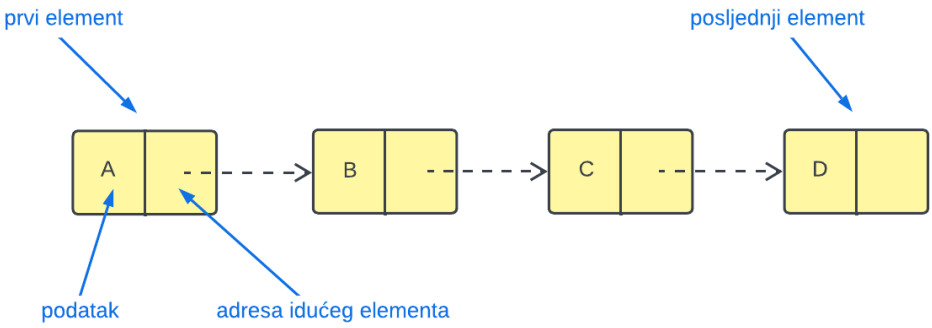
Operacija	***Vremenska složenost***
Indeksiranje	$\Theta(1)$
Dodavanje	$\Theta(n)$ najgori slučaj, $\Theta(1)$ najbolji slučaj
Brisanje	$\Theta(n)$ najgori slučaj, $\Theta(1)$ najbolji slučaj

Tablica 5.1: Vremenske složenosti operacija indeksiranja, dodavanja i brisanja elemenata nesortiranog niza.

U zaključku možemo reći da je niz struktura podataka koja ima dvije prednosti u odnosu na druge strukture podataka:

- brz pristup elementima putem numeričkog indeksa
- dobro iskorištavanje načela lokaliteta.

Isto tako, nedostaci niza kao strukture podataka nalaze se u dodavanju i brisanju elemenata na ili blizu početka niza. U tim slučajevima, elementi se moraju pomicati, odnosno kopirati s jednog mjesta na drugo, a to je skupa operacija. Ovdje opet može pomoći načelo lokaliteta jer ako se dodavanje ili brisanje odvija u cache-memoriji. To znatno ubrzava ove operacije iako njihova asimptotska vremenska složenost ostaje ista. Tablica 5.1 prikazuje vremensku složenost triju operacija nad nizom: indeksiranje, dodavanje i brisanje elemenata. Za dodavanje i brisanje elemenata najgori slučaj je kada se odvija na početku niza. Za indeksiranje vremenska složenost je uvijek konstantna.



Slika 5.12: Konceptualni prikaz jednostruko povezane liste s četiri elementa.

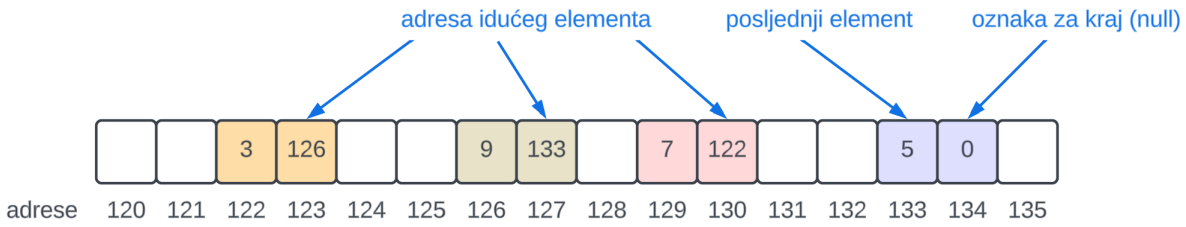
5.4 Povezana lista

Ako nam je učinkovito dodavanje i uklanjanje elemenata važnije od učinkovitog indeksiranja, treba uzeti u obzir takozvane *povezane strukture podataka* (engl. *linked data structures*). Kod takvih struktura podataka elementi nisu smješteni u memoriji jedan iza drugoga, niti je jedan dio memorije unaprijed rezerviran za određeni broj elemenata, nego se memorija rezervira za svaki element posebno.[4] Nadalje, svaki od tih elemenata obuhvaća jednu ili više adresa na kojima se nalaze elementi te strukture koji su s njim povezani. Jedna takva struktura podataka je *povezana lista* (engl. *linked list*). Ova se struktura podataka zove tako jer predstavlja jedan niz elemenata koji su organizirani na drugačiji način od klasičnog niza da bi se omogućilo učinkovito umetanje i brisanje elemenata. Konceptualni prikaz povezane liste koja sadrži četiri elementa prikazan je na Slici 5.12. Ta slika prikazuje *jednostruko povezanu listu* (engl. *singly-linked list*).

5.4.1 Jednostruko povezana lista

Kod jednostruko povezanih lista svaki je element jedan par koji se sastoji od podatka koji želimo smjestiti u listu i adrese idućeg elementa. Ako se spustimo na razinu memorije, jednostruko povezana lista sa Slike 5.12 mogla bi izgledati kako je prikazano na Slici 5.13. Na toj se slici vidi da elementi u memoriji nisu obavezno poslagani istim redoslijedom kao na listi. Jedini redoslijed koji je fiksan redoslijed je podatka i adrese idućeg elementa i te su dvije vrijednosti postavljene na susjedne memorijske adrese. U načelu, za svaki element povezane liste mora se znati gdje se nalazi idući element. Kako ćemo to osigurati ovisi o detaljima implementacije same liste. Nadalje, jedna adresa mora predstavljati početak liste, odnosno njezin prvi element, slično kao s nizom. Ako sada pratimo adrese počevši od 129 na kojoj se nalazi prvi element, dobit ćemo ispravni redoslijed elemenata ove liste. Primjerice, kod elementa na paru adresa $129/130$ vidimo da se idući element nalazi na adresi 122, a na paru adresa $122/123$ idući se element nalazi na adresi 126, itd. Prema tome, niz adresa koje ove podatke povezuju u listu su $129 \rightarrow 122 \rightarrow 126 \rightarrow 133 \rightarrow 0$.

[4] Tehnički, to ne mora biti tako jer elementi liste mogu putem adrese dolaziti i, recimo, iz niza ili statičkog dijela memorije u kojem se oni stvarno nalaze smješteni jedan do drugoga. To nije tipičan način korištenja povezane liste.



Slika 5.13: Primjer jednostruko povezane liste na razini memorije: svaki se element sastoji od vrijednosti i adrese na kojoj se nalazi idući element liste. Ako je adresa prvog elementa 129, ova lista sadrži elemente s ključevima $7 \to 3 \to 9 \to 5$.

Adresa 0 za idući element označava da taj element ne postoji. Kao posebnu oznaku u pseudokodu da varijabla ne sadrži adresu nekog podatka koristit ćemo NIL.

Operacije. Za jednostruko povezanu listu možemo definirati sljedeće operacije:

- NAĐI(k) – pronalazi element s ključem k
- DODAJ(e) – dodaje element e na početak liste
- OBRIŠI(k) – briše prvi element s ključem k

Element jednostruko povezane liste objekt je s atributima za adresu idućeg elementa liste, ključ i podatke kojih je taj element nositelj, kako je prikazano definicijom tipa 5.3. Ključ je podatak po kojem se elementi liste pretražuju.

Tip 5.3 Definicija tipa elementa za jednostruko povezanu listu.

```
1: type ELEMENT-JPL
2:     KLJUČ                          ▷ vrijednost po kojoj se elementi liste pretražuju
3:     IDUĆI: ELEMENT-JPL             ▷ adresa idućeg elementa
4: end type
```

Listu možemo promatrati kao instancu tipa koji se sastoji od jednog atributa, $prvi$, koji sadrži adresu prvog elementa, te operacija za dodavanje, pretraživanje i brisanje elemenata (definicija tipa 5.4).

Tip 5.4 Definicija tipa jednostruko povezane liste.

```
1: type JPL
2:     NAĐI(k)                        ▷ traženje elementa s ključem k
3:     DODAJ(e: ELEMENT-JPL)          ▷ dodavanje novog elementa e na početak liste
4:     OBRIŠI(k)                      ▷ brisanje prvog elementa s ključem k
5:     PRVI: ELEMENT-JPL              ▷ adresa prvog elementa
6: end type
```

Pretraživanje. Postupak pretraživanja povezane liste polazi od početnog elementa i slijedi vezu s idućim dok ne pronađe element sa zadanim ključem ili dok ne dođe do kraja liste. Ovaj je postupak prikazan Algoritmom 5.7. Za listu koja se sastoji od n elemenata, pretraživanje povezane liste u najgorem slučaju (kada element sa zadanim ključem ne postoji u listi) iziskuje $\Theta(n)$ vremena.

Algoritam 5.7 Pretraživanje jednostruko povezane liste.

Parametri: k - ključ koji tražimo
Rezultat: adresa elementa s traženim ključem ili NIL ako on ne postoji u listi

```
1: function JPL.NAĐI(k)
2:     e ← PRVI
3:     while e ≠ NIL and e.KLJUČ ≠ k do
4:         e ← e.IDUĆI
5:     end while
6:     return e                                   ▷ vrati adresu elementa
7: end function
```

Dodavanje elemenata. Da bismo u jednostruko povezanu listu dodali novi element moramo u obzir uzeti dvije stvari:

- ako je lista prazna, atribut *prvi* postavimo na adresu novog elementa
- ako lista sadrži jedan ili više elemenata, novi element dodajemo kao novi prvi element i povežemo ga sa elementom koji je prethodno bio prvi.

Oba slučaja možemo obuhvatiti bez eksplicitne provjere sadrži li lista barem jedan element jer ako je lista prazna, atribut *prvi* sadržavat će NIL. To možemo iskoristiti tako da atributu $idui$ novog elementa odmah pridružimo vrijednost atributa $prvi$. Ovo je pokazano Algoritmom 5.8.

Algoritam 5.8 Dodavanje elementa u jednostruko povezanu listu.

Parametri: e - adresa elementa koji dodajemo

```
1: procedure JPL.DODAJ(e : ELEMENT-JPL)
2:     e.IDUĆI ← PRVI
3:     PRVI ← e
4: end procedure
```

Iz ovoga se jasno vidi da je vremenska složenost dodavanja novog elementa u jednostruko povezanu listu $\Theta(1)$.

Brisanje elemenata. S obzirom da svaki element jednostruko povezane liste sadrži samo adresu idućeg, ova vrsta povezane liste ne može obrisati bilo koji element u konstantnom vremenu. Iz tog razloga brisanje elementa iz jednostruko povezane liste ne može se provesti na osnovi njegove adrese nego ga se prvo mora pronaći na osnovi ključa. Ovaj postupak treba voditi računa o dva moguća slučaja:

- Ako se briše prvi element, treba samo postaviti atribut $prvi$ na element koji slijedi.
- Ako se briše element koji nije prvi, atribut $idui$ prethodnog elementa treba postaviti na element koji slijedi onog koji se briše.

Ovaj postupak prikazan je Algoritmom 5.9. Njegova je vremenska složenost $O(n)$ u najgorem slučaju (kada traženi element ne postoji), ali može biti i bolja ako se taj element nalazi na početku liste.

Algoritam 5.9 Brisanje prvog elementa iz jednostruko povezane liste prema ključu.

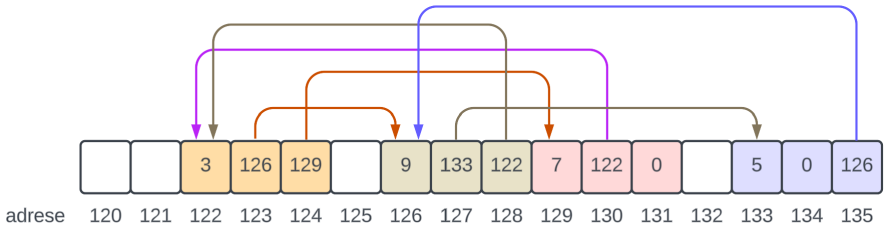
Parametri: k - ključ po kojem se pronalazi element za brisanje
Rezultat: $TRUE$ ako je element za brisanje pronađen, inače $FALSE$

```
1:  function JPL.BRIŠI(k)
2:      prethodni ← NIL
3:      e ← PRVI
4:      while e ≠ NIL do
5:          if e.KLJUČ = k then
6:              if prethodni = NIL then
7:                  PRVI ← e.IDUCI
8:              else
9:                  prethodni.IDUCI ← e.IDUCI
10:             end if
11:             return TRUE
12:         end if
13:         prethodni ← e
14:         e ← e.IDUCI
15:     end while
16:     return FALSE
17: end function
```

U sljedećem dijelu proučit ćemo jednu fleksibilniju inačicu povezane liste koja se zove *dvostruko povezana lista*.

5.4.2 Dvostruko povezana lista

Kod jednostruko povezane liste elementima možemo pristupati samo u jednom smjeru. Da bismo omogućili dvosmjerno kretanje kroz listu, svaki element mora imati poveznicu s prethodnim i idućim elementom. Ovo je ilustrirano na Slici 5.14 gdje svaki element zauzima tri adrese: jednu za ključ, drugu za adresu prethodnog elementa i treću za adresu idućeg elementa (tim redoslijedom). Primjerice, element s ključem 9 ima prethodni element na adresi 133 (ključ 5), a idući na adresi 122 (ključ 3). Ključ 5 je prvi element liste jer mu je adresa prethodnog elementa 0, dok je ključ 7 posljednji element jer mu je adresa idućeg elementa 0. Ovakva se lista zove *dvostruko povezana lista* (engl. *doubly-linked list)*. Kod dvostruko povezane liste svaki je element jedna četvorka koja se sastoji od podatka, ključa, adrese elementa koji mu prethodi te adrese elementa koji ga slijedi. Ovdje je svaki element povezan i s idućim i s prethodnim elementom. To ovakvoj listi daje dodatnu fleksibilnost u odnosu na jednostruko povezanu listu.



Slika 5.14: Primjer dvostruko povezane liste. Svaki se element sastoji od podatka i dvije adrese: jedne za prethodni i druge za idući element liste. Poveznice za idući i prethodni element prikazane su istom bojom. Ova se lista sastoji od elemenata s ključevima u poretku $5 \to 9 \to 3 \to 7$. Element s ključem 5 zauzima adrese $133 - 135$, onaj s ključem 9 adrese $126 - 128$, itd.

U definiciji 5.5 definiran je tip elemenata dvostruko povezane liste. Jedina razlika u odnosu na tip ELEMENT-JPL je što ovdje imamo jedan dodatni atribut $prethodni$ koji služi kao poveznica s prethodnim elementom.

Tip 5.5 Definicija tipa elementa za dvostruko povezanu listu.

```
1: type ELEMENT-DPL
2:     KLJUČ                          ▷ vrijednost po kojoj se elementi liste pretražuju
3:     PRETHODNI: ELEMENT-DPL         ▷ adresa prethodnog elementa
4:     IDUĆI: ELEMENT-DPL             ▷ adresa idućeg elementa
5:     PODACI                         ▷ podaci koje "nosi" element
6: end type
```

U definiciji 5.6 definiran je tip dvostruko povezane liste. U odnosu na jednostruko povezanu listu, dodavanje i brisanje elemenata u dvostruko povezanu listu malo je složenije jer moramo voditi računa o dvije veze: onoj s idućim i onoj s prethodnim elementom. Isto tako, u definiciji 5.6 vidimo još jednu operaciju brisanja, onu koja prima adresu elementa koji se briše. Kod dvostruko povezane liste takva je operacija moguća jer svaki element ima vezu s prethodnim i idućim elementom pa je time moguće prespojiti ta dva elementa nakon brisanja onog između njih.

Tip 5.6 Definicija tipa dvostruko povezane liste.

```
1: type DPL
2:     NAĐI(k)                        ▷ traženje elementa s ključem k
3:     DODAJ(e)                       ▷ dodavanje novog elementa e na početak liste
4:     OBRIŠI(k)                      ▷ brisanje prvog elementa s ključem k
5:     OBRIŠI(e)                      ▷ brisanje elementa sa zadanom adresom e
6:     PRVI                           ▷ adresa prvog elementa
7: end type
```

Dodavanje elemenata Dodavanje novog elementa u dvostruko povezanu listu prikazano je Algoritmom 5.10. Kao i kod jednostruko povezane liste, vremenska složenost dodavanja novog elementa je $\Theta(1)$.

Algoritam 5.10 Dodavanje elementa u dvostruko povezanu listu.

Parametri: e - adresa elementa kojeg dodajemo

```
1: procedure DPL.DODAJ(e : ELEMENT-DPL)
2:     e.IDUĆI ← PRVI
3:     if PRVI ≠ NIL then
4:         PRVI.PRETHODNI ← e
5:     end if
6:     PRVI ← e
7:     e.PRETHODNI ← NIL
8: end procedure
```

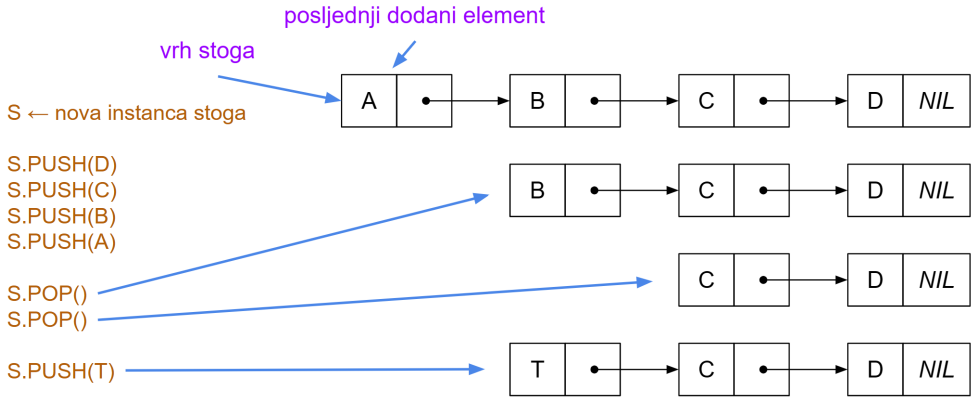
Brisanje elemenata. Brisanje elemenata dvostruko povezane liste možemo realizirati kroz zadanu adresu elementa koji brišemo, što je pokazano Algoritmom 5.11. Sada ne vraćamo nikakvu vrijednost kao rezultat pa je stoga ova operacija prikazana kao procedura. Brisanje putem ključa slično je kao i kod jednostruko povezane liste.

Algoritam 5.11 Brisanje elementa iz dvostruko povezane liste

Parametri: e - adresa elementa kojeg brišemo

```
1: procedure DPL.BRIŠI(e : ELEMENT-DPL)
2:     if e.PRETHODNI ≠ NIL then
3:         e.PRETHODNI.IDUĆI ← e.IDUĆI
4:     else
5:         PRVI ← e.IDUĆI
6:     end if
7:     if e.IDUĆI ≠ NIL then
8:         e.IDUĆI.PRETHODNI ← e.PRETHODNI
9:     end if
10: end procedure
```

Za razliku od brisanja elementa iz jednostruko povezane liste gdje smo morali pronaći element sa zadanim ključem, kod dvostruko povezane liste možemo prihvatiti samo adresu elementa koji brišemo jer preko njega možemo doći i do prethodnog i do idućeg elementa. Stoga, vremenska je složenost brisanja elementa iz dvostruko povezane liste $\Theta(1)$.



Slika 5.15: Primjer stoga implementiranog pomoću jednostruko povezane liste.

5.5 Stog i red

Niz i povezana lista dvije su općenite strukture podataka koje mogu poslužiti kao osnova za druge strukture podataka, od kojih ćemo ovdje proučiti dvije: *stog* i *red*.

5.5.1 Stog

Stog je struktura podataka koja se, kao i povezana lista ili niz, sastoji od jednog niza podataka, ali kod koje elemente možemo dodavati i uklanjati na samo jednom kraju. Ovu struktura podataka karakteriziraju dvije specifične operacije – *push* za dodavanje elemenata i *pop* za njihovo uklanjanje. Kod stoga posljednji dodani element prvi je koji se može iz njega uzeti, odnosno posljednji element koji je ušao, prvi je koji ide van. Stog radi po načelu LIFO (engl. *Last-In, First-Out)*, kao što je prikazano na Slici 5.15. Ova struktura podataka, iako vrlo jednostavna, ima veliku važnost u programiranju, što ćemo vidjeti kasnije na nekim primjerima pretraživanja.

Stog možemo implementirati na više načina, primjerice, kao niz fiksne veličine, dinamički niz, jednostruko ili dvostruko povezanu listu. Takvu listu već samu po sebi možemo koristiti kao stog koristeći operacije *dodaj* i *ukloni*. Stog se razlikuje od povezane liste po operacijama koje su za njega definirane, a ne po načinu na koji povezujemo elemente. Mogućnosti stoga kao strukture podataka više su ograničene u odnosu na povezanu listu jer stog, barem u teoriji, ne podržava operacije arbitrarnog pretraživanja ili umetanja novih elemenata bilo gdje u listi. Tip podatka za stog obično podržava jedan mali broj temeljnih operacija kako je prikazano u definiciji tipa 5.7.

Tip 5.7 Definicija tipa stog.

```
1: type STOG
2:     PUSH(e)          ▷ dodaje novi element e na vrh stoga
3:     POP()            ▷ vraća i uklanja element s vrha stoga
4:     VRH()            ▷ vraća element s vrha stoga, ali ga ne uklanja
5:     PUN()            ▷ vraća TRUE ako je stog pun
6:     PRAZAN()         ▷ vraća TRUE ako je stog prazan
```

```
 7:     VRH ← 1                                  ▷ element na vrhu stoga
 8:     VELIČINA                  ▷ veličina niza u koji se spremaju elementi
 9:     NIZ[1..VELIČINA]                  ▷ niz u koji se spremaju elementi
10: end type
```

Standardna biblioteka mnogih programskih jezika sadrži neku inačicu ovakvog stoga. Jednostavan način implementacije stoga je pomoću dinamičkog niza. Algoritam 5.12 prikazuje rad operacije PUSH.

Algoritam 5.12 Dodavanje elementa na stog.

Parametri: e - element koji dodajemo

```
1: procedure STOG.PUSH(e)
2:     if PUN() then
3:         greška "stog je pun"
4:     end if
5:     NIZ[VRH] ← e
6:     VRH ← VRH + 1
7: end procedure
```

S obzirom da za implementaciju stoga koristimo niz fiksne veličine, potrebno je prvo provjeriti (redci $2..4$) ima li prostora u nizu za novi element i, ako nema, dojaviti grešku.

Algoritmom 5.13 prikazana je operacija POP. S obzirom da ova operacija uzima element sa stoga, moramo prvo provjeriti je li stog prazan (redci $2..4$) i, ako je, dojaviti grešku. Ova operacija ilustrirana je na Slici 5.16 [23].

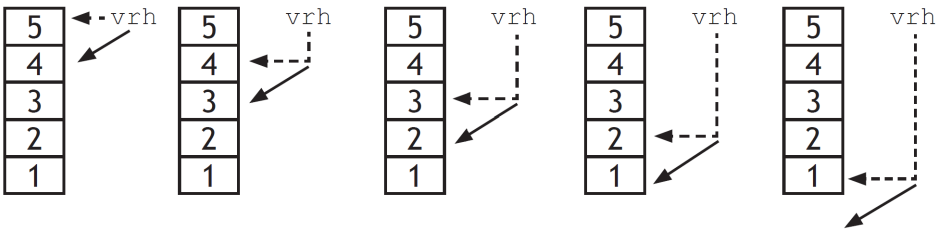
Algoritam 5.13 Uzimanje elementa sa stoga.

Rezultat: element sa vrha stoga

```
1: function STOG.POP()
2:     if PRAZAN() then
3:         greška "stog je prazan"
4:     end if
5:     VRH ← VRH − 1
6:     return NIZ[VRH]
7: end function
```

Operacije PUN i PRAZAN trivijalne su (Algoritmi 5.14 i 5.15) kao i operacija VRH (Algoritam 5.16).

Kao što je prethodno rečeno, stog se može implementirati upotrebom povezane liste ili niza (fiksne veličine ili dinamičkog). Stog se razlikuje od niza ili povezane liste po operacijama. Dok niz i povezana lista u načelu omogućavaju slobodan pristup elementima, elementima stoga pristupa se isključivo pomoću operacija PUSH i POP. Stog kao koncept



Slika 5.16: Primjer operacije STOG.POP sa stogom implementiranim kao niz [23].

Algoritam 5.14 Provjera je li stog pun.

Rezultat: $TRUE$ ako je stog pun, inače $FALSE$

```
1: function STOG.PUN()
2:     return VRH > VELIČINA
3: end function
```

Algoritam 5.15 Provjera je li stog prazan.

Rezultat: $TRUE$ ako je stog prazan, inače $FALSE$

```
1: function STOG.PRAZAN()
2:     return VRH − 1 = 0
3: end function
```

Algoritam 5.16 Element na vrhu stoga.

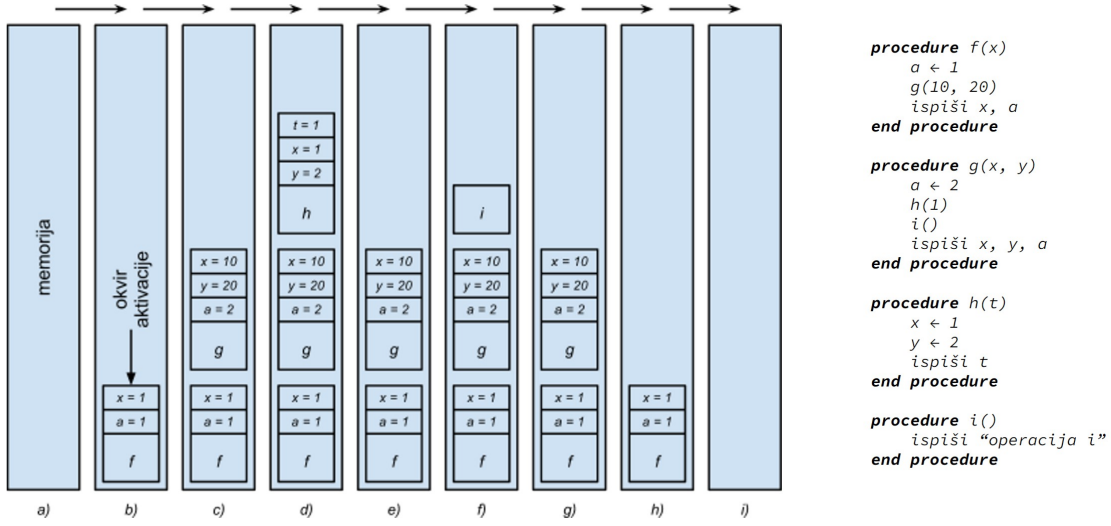
Rezultat: element na vrhu stoga

```
1: function STOG.VRH()
2:     return NIZ[VRH - 1]
3: end function
```

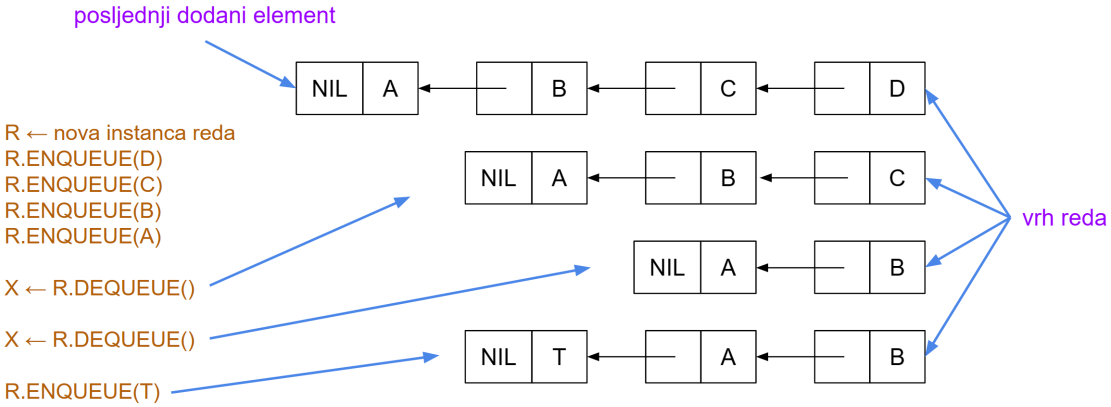
ima temeljnu važnost u računarstvu. Jedna od primjena stoga je u izvršavanju programa, konkretno u izvršavanju poziva podprograma (funkcija i/ili procedura). Na Slici 5.17 prikazano je odvijanje poziva procedura na razini memorije računala, gdje vanjski okviri predstavljaju memoriju računala. Kod svakog poziva funkcije ili procedure prevoditelj programskog jezika ili kôd koji je prevoditelj generirao, formira tzv. *okvir aktivacije* ili *aktivacijski okvir* (engl. *activation frame*) za tu funkciju ili proceduru. Detalji oko toga koje sve informacije sadrži jedan okvir aktivacije zavise od programskog jezika i nekih drugih tehničkih faktora. Ovdje ćemo se usredotočiti na ono što je zajedničko mnogim programskim jezicima kao što su Python ili C++, bez detalja koji nisu važni za razumijevanje načela o kojima je ovdje riječ. Jedan aktivacijski okvir obično sadrži podatke kao što su vrijednosti parametara i lokalnih varijabli. Tijekom izvršavanja neke funkcije ili procedure vrijednosti njenih lokalnih varijabli (uključujući i parametre) uzimaju se samo iz njenog okvira aktivacije. Ti okviri nisu vidljivi programeru – oni služe kao mehanizam u pozadini koji upravlja ovakvim pozivima i dio je implementacije programskog jezika.

Načelo poziva operacije proučit ćemo na gornjem primjeru. Prije poziva `f(1)` imamo situaciju kao na slici a) gdje je dio memorije rezerviran za okvire aktivacije prazan. Nakon poziva `f(1)` u memoriji će biti formiran okvir aktivacije za operaciju *f*, što je pokazano na slici b). Vidimo da taj okvir sadrži vrijednost parametra *x* i lokalne varijable *a* ove procedure. Nakon poziva operacije *f* slijedi poziv operacije *g*, i to iz operacije *f*. To znači da okvir aktivacije za operaciju *f* mora biti sačuvan u memoriji jer će se izvršavanje u njoj nastaviti nakon povratka iz operacije *g*. Za poziv operacije *g* formira se novi okvir (na sljedećem slobodnom segmentu memorije) gdje on također sadrži vrijednosti parametara i lokalnih varijabli, što je pokazano na slici c). Izvršavanje se sada odvija u operaciji *g* gdje se koriste vrijednosti varijabli *x*, *y* i *a* isključivo iz tog okvira. Sada iz operacije *g* slijedi poziv operacije *h* tako da se ovdje događa isto što i u prethodnom slučaju – sačuva se okvir za *g* i formira novi za *h* nakon čega imamo situaciju kao na slici d). Nakon povratka iz operacije *h* njen okvir više nije potreban. Izvršavanje se sada nastavlja u operaciji *g*, ponovno koristeći vrijednosti varijabli u njenom okviru, što je prikazano na slici e). Nakon toga, opet iz operacije *g*, slijedi poziv operacije *i* za koju se formira novi okvir. S obzirom da ova operacija nema parametre ni lokalne varijable, njen okvir ne sadrži te podatke, što se vidi na slici f). Nakon povratka iz *i* izvršavanje se opet nastavlja u operaciji *g* kako je pokazano na slici g) gdje se koristi njen okvir aktivacije, dok okvir za *i* više nije potreban. Slijedi povratak iz *g* u *f*, što znači da se opet vraćamo na okvir za *f* koristeći tamošnje vrijednosti za *x* i *a* što se vidi na slici h). Nakon povratka iz *f* okvir te operacije više nije potreban, pa se time vraćamo na prvobitno stanje pokazano na slici i) koje je identično onome na slici a).

Jedan niz okvira aktivacije, kao na slici d), zove se *stog poziva* (engl. *call stack*) i po njemu se vidi kako je tekao poziv operacija do određenog trenutka. Primjerice, na slici f) na osnovu stoga poziva vidimo da se u operaciju *i* došlo iz operacije *g*, a u operaciju *g* iz operacije *f*. Stog poziva ne pokazuje koja se operacija prethodno izvršavala. Primjerice, stog poziva na slici e) ne pokazuje da smo se iz operacije *h* vratili u operaciju *g*. U ovom slučaju stog poziva izgleda isto kao i oni na slikama c) i g). Stog poziva pokazuje samo niz poziva operacija do određenog trenutka, ali ne i povratke iz njih. Svi moderni alati za programiranje mogu pokazati ovakav stog poziva operacija u nekom trenutku što



Slika 5.17: Upotreba stoga za implementaciju poziva funkcija i procedura.



Slika 5.18: Primjer reda implementiranog pomoću jednostruko povezane liste.

programeru znatno olakšava analizu njihovog izvršavanja.

5.5.2 Red

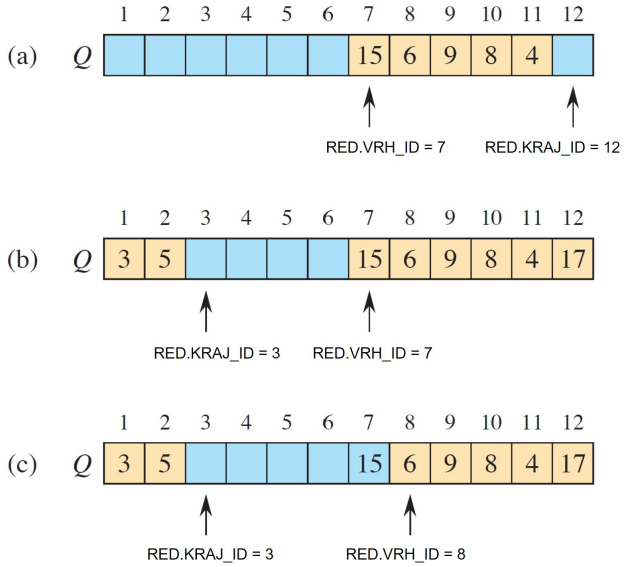
Još jedna jednostavna struktura podataka slična stogu je *red*, a razlikuje se od stoga po tome što elemente dodajemo na jednom kraju, a vadimo ih na drugom. Red radi po načelu FIFO (engl. *First-In, First-Out)*, kao što je prikazano na Slici 5.19. Za redove su obično definirane dvije ključne operacije:

- dodavanje u red, koja se označava s *enqueue*
- vađenje iz reda, koja se označava s *dequeue*.

Tip ove strukture podataka možemo definirati kao što je pokazano u definiciji tipa 5.8.

Tip 5.8 Definicija tipa red.

1: **type** RED
2: ENQUEUE(e) ▷ dodaje novi element e na kraj reda



Slika 5.19: Primjer reda implementiranog pomoću niza[1].

```
 3:    DEQUEUE()          ▷ vraća i uklanja element na početku reda
 4:    POČETAK()          ▷ vraća element na početku reda, ali ga ne uklanja
 5:    KRAJ()             ▷ vraća element na kraju reda, ali ga ne uklanja
 6:    PUN()              ▷ vraća TRUE ako je red pun
 7:    PRAZAN()           ▷ vraća TRUE ako je red prazan

 8:    VRH_ID ← 1         ▷ indeks ili pokazivač na početak reda
 9:    KRAJ_ID ← 1        ▷ indeks ili pokazivač na kraj reda
10:    VELIČINA           ▷ veličina niza u koji se spremaju elementi
11:    NIZ[1..VELIČINA]   ▷ niz u koji se spremaju elementi
12: end type
```

Na Slici 5.19 prikazana je implementacija reda pomoću niza fiksne veličine. Na slici a) prikazan je red koji sadrži pet vrijednosti (označeno žutom bojom) gdje je ona na početku reda na indeksu 7, a kraj reda je na indeksu 12. Slika b) prikazuje taj red nakon što su u njega dodane vrijednosti 17, 3 i 5 (tim redoslijedom). Ovdje vidimo da kada kraj reda dođe do kraja niza, novi se elementi dodaju počevši od početka niza. Sada je kraj reda na indeksu 3. Na slici c) vidimo stanje reda nakon pozivanja operacije DEQUEUE. Ta je operacija pomakla indeks početka reda na idući indeks i vratila element koji se prethodno nalazio na početku reda, odnosno element 15. Novi početak reda sada je na indeksu 8.

Operacije ENQUEUE i DEQUEUE možemo implementirati kako je pokazano u algoritmima 5.17 i 5.18.

Red je pun kada je VRH_ID = KRAJ_ID + 1 ili kada je VRH_ID = 1 i KRAJ_ID = VELIČINA, a prazan kada je VRH_ID = KRAJ_ID. Operacije PUN i PRAZAN implementirane su prema ovim pravilima u algoritmima 5.19 i 5.20.

Algoritam 5.17 Dodavanje elementa u red.

Parametri: e - novi element

```
1:  procedure RED.ENQUEUE(e)
2:      if PUN() then
3:          greška "red je pun"
4:      end if
5:      NIZ[VRH_ID] ← e
6:      if KRAJ_ID = VELIČINA then
7:          KRAJ_ID ← 1
8:      else
9:          KRAJ_ID ← KRAJ_ID + 1
10:     end if
11: end procedure
```

Algoritam 5.18 Uzimanje elementa iz reda.

Rezultat: element s vrha reda

```
1:  function RED.DEQUEUE()
2:      if PRAZAN() then
3:          greška "red je prazan"
4:      end if
5:      e ← NIZ[VRH_ID]
6:      if VRH_ID = VELIČINA then
7:          VRH_ID ← 1
8:      else
9:          VRH_ID ← VRH_ID + 1
10:     end if
11:     return e
12: end function
```

Algoritam 5.19 Provjera je li red pun.

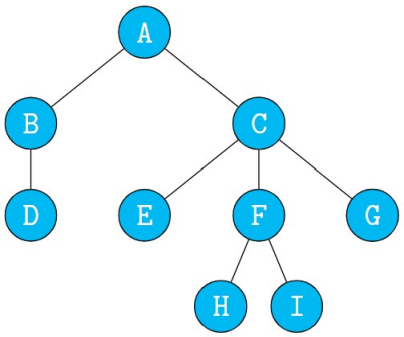
Rezultat: $TRUE$ ako je red pun, inače $FALSE$

```
1:  function RED.PUN()
2:      return VRH_ID = KRAJ_ID + 1  or  (VRH_ID = 1  and  KRAJ_ID = VELIČINA)
3:  end function
```

Algoritam 5.20 Provjera je li red prazan.

Rezultat: $TRUE$ ako je red prazan, inače $FALSE$

```
1:  function RED.PRAZAN()
2:      return VRH_ID = KRAJ_ID
3:  end function
```

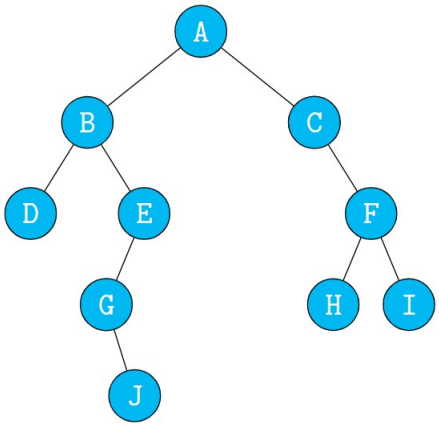


Slika 5.20: Primjer stabla.

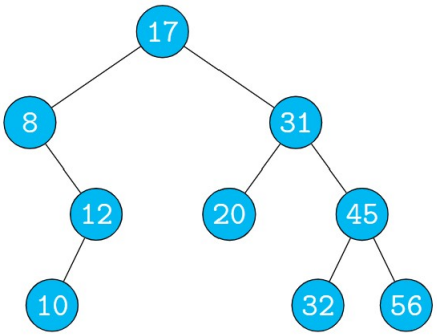
5.6 Stablo

Kao što smo vidjeli u 5.3.4, binarno pretraživanje nizova ima značajno bolje performanse od linearnog. Dodavanje i uklanjanje elemenata otežava činjenica da su nizovi kompaktna struktura podataka koja zahtijeva da se postojeći elementi često pomiču na druge pozicije unutar niza. Povezane liste, s druge strane, omogućavaju učinkovito dodavanje i uklanjanje elemenata, ali s cijenom neučinkovitog indeksiranja koje usporava binarno pretraživanje.

U ovom poglavlju proučit ćemo strukturu podataka koja se zove *stablo* jer izgledom podsjeća na njega, što je vidljivo iz Slike 5.20. Uobičajeno je da se struktura stabla prikazuje s korijenom na vrhu tako da se ono grana odozgo prema dolje. Iako po svom obliku jednostavna, ova struktura podataka ima iznenađujuće veliki značaj u računarstvu. U poglavlju 3 vidjeli smo kako se pozivi funkcija mogu prikazati pomoću stabla kada smo govorili o rekurzivnim funkcijama. Stablo se može koristiti i kao model nekog procesa ili odnosa među objektima i podacima, a ne samo kao zbirka podataka. U ovom ćemo dijelu strukturu stabla proučavati isključivo s gledišta organiziranja i pretraživanja podataka. Na Slici 5.20 vidimo stablo koje se sastoji od jednog čvora A koji predstavlja *korijen* (engl. *root*) tog stabla i skupine drugih čvorova koji se iz njega neposredno ili posredno granaju. Ovakvo se stablo naziva *ukorijenjenim stablom* (engl. *rooted tree*). Također vidimo da se stablo sastoji od više drugih, manjih stabala, gdje svako od njih nazivamo *podstablom* (engl. *subtree*). Primjericw, čvor C i čvorovi koji se iz njega granaju predstavljaju jedno podstablo. Isto važi i za čvor F, ali i za krajnje čvorove kao što su D, E, H, I i G, koji se nazivaju *listovima* (engl. *leaf*) ovog stabla. Ako se neki čvor grana u dva podstabla kaže se da taj čvor ima lijevo i desno podstablo, kao čvorovi A i F na slici. Čvorovi koji se granaju iz jednog čvora nazivaju se *djecom* ili *potomcima* (engl. *child node* ili *successor node*) tog čvora, a sam taj čvor je njihov *roditelj* ili *predak* (engl. *parent node* ili *predecessor node*). Na Slici 5.20 čvor D je dijete čvora B, a čvor B je njegov roditelj. Roditeljem ćemo nazivati čvor koji je izravan predak nekog čvora, a djetetom čvor koji je izravan potomak. Primjerice, čvorovi H i I su potomci (ali ne djeca) čvora C, a čvor A je predak (ali ne roditelj) čvora H.



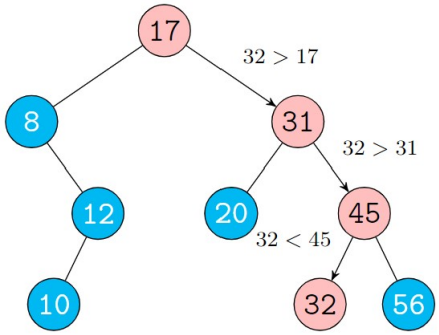
Slika 5.21: Primjer binarnog stabla.



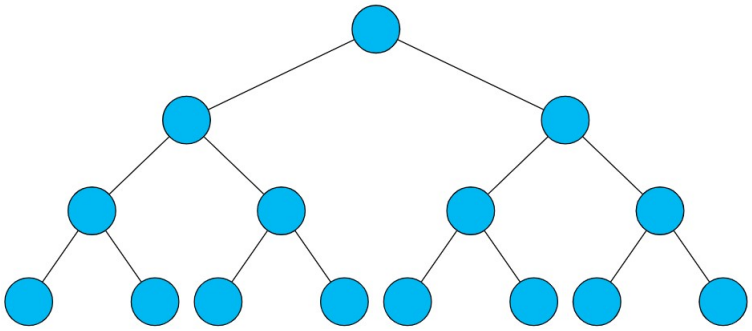
Slika 5.22: Primjer stabla za binarno pretraživanje.

5.6.1 Binarno stablo

Jedna vrsta stabla s velikom primjenom u računarstvu zove se *binarno stablo* (engl. *binary tree*) [1, 2]. Kod te vrste stabla svaki se čvor grana u najviše dva druga čvora. Na Slici 5.21 prikazano je jedno binarno stablo. Za razliku od njega, stablo na Slici 5.20 nije binarno jer se čvor C grana u tri čvora, E, F i G. Binarna stabla često se koriste kao struktura podataka pogodna za binarno pretraživanje. U binarnom stablu, ako je svako lijevo dijete manje ili jednako od svog roditelja, a svako desno veće ili jednako, takvo se stablo zove *stablo za binarno pretraživanje* (engl. *binary search tree*). Drugim riječima, za svaki čvor stabla za binarno pretraživanje s njegove lijeve strane nalaze se elementi koji nisu veći od njega, a s njegove desne strane oni koji nisu manji. Na Slici 5.22 prikazano je jedno stablo za binarno pretraživanje. Vidimo da su čvorovi koji su manji od korijena s njegove lijeve strane, a svi oni veći s desne. To također važi i za sve ostale čvorove ovakvog stabla, odnosno za svako podstablo. Ako, primjerice, pogledamo čvor 31, vidimo da je 20 s njegove lijeve strane, a 32, 45 i 56 s desne. Ako sada zamislimo kako pronaći neku vrijednost u ovakvom stablu možemo uočiti sličnost s binarnim pretraživanjem koje smo primijenili kod nizova. Načelo binarnog pretraživanja kod ovakvog je stabla isto. Ako tražimo vrijednost 32, počinjemo od korijena. S obzirom da je $32 > 17$ nastavljamo tražiti u desnom podstablu. Sljedeća vrijednost je 31 tako da opet nastavljamo u desnom podstablu čvora 31 gdje se nalazi vrijednost 45. Sada tražimo dalje u lijevom podstablu čvora 45 jer je $32 < 45$, čime dolazimo do tražene vrijednosti 32. Ovaj je postupak ilustriran na Slici 5.23.



Slika 5.23: Traženje broja 32 u stablu za binarno pretraživanje.

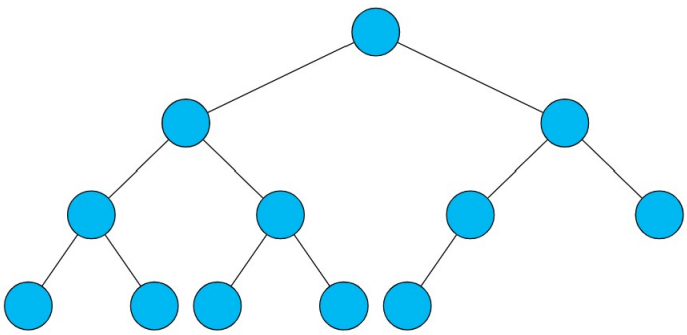


Slika 5.24: Primjer potpunog binarnog stabla.

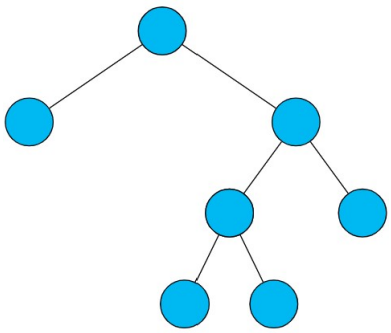
5.6.1.1 Osnovni pojmovi i svojstva binarnog stabla

U ovom dijelu opisani su neki pojmovi povezani sa stablima općenito ili binarnim stablima. To su visina stabla, dubina čvora, stupanj čvora, potpuno, puno, balansirano i degenerirano binarno stablo.

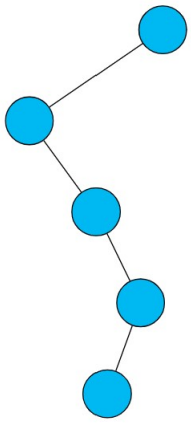
- ***Visina stabla***: broj čvorova na putu od korijena do najdaljeg lista stabla, s tim da se korijen nalazi na visini 0. Na primjer, stablo na Slici 5.21 ima visinu 4, dok ono na Slici 5.23 ima visinu 3. Visinu stabla ćemo u matematičkoj notaciji označavati slovom h.
- ***Dubina čvora***: duljina puta od korijena do tog čvora.
- ***Stupanj čvora***: broj djece čvora. Primjerice, na Slici 5.20 stupanj čvora C je 3.



Slika 5.25: Primjer skoro potpunog binarnog stabla.



Slika 5.26: Primjer punog binarnog stabla.



Slika 5.27: Primjer degeneriranog binarnog stabla.

- ***Potpuno binarno stablo***: binarno stablo kod kojeg svaki list ima istu dubinu, a svaki unutrašnji čvor dva djeteta (Slika 5.24). *Skoro potpuno binarno stablo* (Slika 5.25) ono je kod kojeg posljednja razina ne mora biti do kraja popunjena, a svi listovi na toj razini poslagani su jedan do drugoga bez praznih mjesta, promatrajući ih slijeva nadesno. Potpuno binarno stablo visine h ima $2^h - 1$ unutrašnjih čvorova i 2^h listova. Potpuno stablo (ne nužno binarno) s unutrašnjim čvorovima stupnja k sadrži $\frac{k^h-1}{k-1}$ unutrašnjih čvorova.

- ***Puno binarno stablo***: binarno stablo kod kojeg svaki čvor ima dva djeteta ili je list stabla odnosno nema djece (Slika 5.26). Broj čvorova punog binarnog stabla najmanje je $2h + 1$, a najviše $2^{h+1} - 1$.

- ***Balansirano binarno stablo***: binarno stablo kod kojeg se visina lijevog i desnog podstabla svakog čvora ne razlikuje za više od 1 (iako se definicije balansiranog binarnog stabla mogu ponešto razlikovati zavisno od algoritma balansiranja). Prema ovoj definiciji, stabla na Slikama 5.24 i 5.25 su balansirana, dok ono na Slici 5.26 nije jer je visina lijevog podstabla korijena 0, a desnog 2.

- ***Degenerirano binarno stablo***: binarno stablo u kojem svaki roditelj ima samo jedno dijete (Slika 5.27). U tom slučaju stablo je u stvari povezana lista.

Kao i povezana lista, stablo je povezana struktura podataka jer je svaki čvor stabla povezan s nekim drugim čvorom. Pored toga, svaki čvor sadrži i ključ po kojem ga možemo naći u stablu. Prema tome, tip podatka kojim možemo prikazati čvor binarnog stabla može se definirati kako je prikazano u definiciji tipa 5.9.

Tip 5.9 Definicija elementa binarnog stabla.

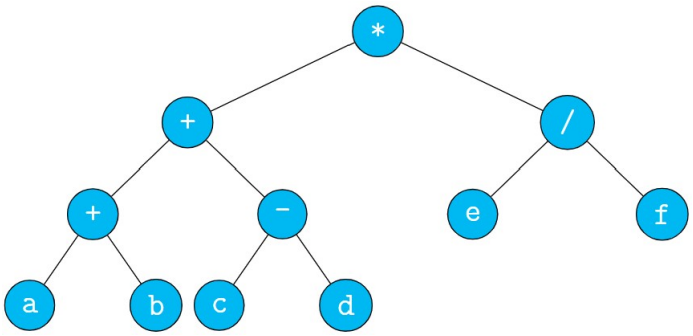
```
1: type ELEMENT-BS
2:     RODITELJ: ELEMENT-BS        ▷ adresa roditelja čvora
3:     LIJEVO: ELEMENT-BS          ▷ adresa lijevog dijeteta
4:     DESNO: ELEMENT-BS           ▷ adresa desnog dijeteta
5:     KLJUČ                       ▷ ključ po kojem pronalazimo čvor
6: end type
```

S obzirom da će se elementi ovog tipa koristiti i za stablo za binarno pretraživanje, pretpostavit ćemo da podaci spremljeni u instance ovog tipa podržavaju operacije uspoređivanja kao što su $<$ i $=$.

5.6.2 Obilazak binarnog stabla

U ovom dijelu opisan je algoritam obilaska binarnog stabla, što se odnosi na poredak pristupa njegovim čvorovima. Postoje tri osnovna načina obilaska takvog stabla:

- prefiksni obilazak
- postfiksni obilazak
- infiksni obilazak.



Slika 5.28: Aritmetički izraz $((a+b)+(c-d))*(e/f)$ u obliku binarnog stabla.

Prefiksni obilazak. Redoslijed čvorova ovog obilaska je roditelj, lijevo dijete, desno dijete. Prefiksni obilazak prikazan je Algoritmom 5.21.

Algoritam 5.21 Prefiksni obilazak binarnog stabla.

Parametri: $cvor$ - adresa polaznog čvora

```
1: procedure PREFIKS(cvor: ELEMENT-BS)
2:     if cvor ≠ NIL then
3:         ispiši cvor.KLJUČ
4:         PREFIKS(cvor.LIJEVO)
5:         PREFIKS(cvor.DESNO)
6:     end if
7: end procedure
```

Za stablo na Slici 5.28 ovaj bi algoritam ispisao $*++ab-cd/ef$. Ovakav obilazak može se primijeniti na bilo kakvo stablo, ne samo na stablo za binarno pretraživanje, iako ima veliku primjenu kod binarnih stabala. Jedna od takvih primjena je u implementaciji programskih jezika gdje se aritmetički izraz kao što je $a*(b+c)$ može prevesti na tzv. *prefiksni oblik*. Taj oblik nema zagrade pa je time i jednostavniji za interpretiranje ili prevođenje. Primjerice, za izraz kao što je $((a+b)+(c-d))*(e/f)$ prevoditelj može prvo formirati stablo prikazano na Slici 5.28. Nakon toga, nad tim stablom može primijeniti proceduru kao što je PREFIKS čime bi dobio prefiksni oblik tog izraza odnosno $*++ab-cd/ef$. Ovakav se izraz sada može jednostavno izračunati tako da se uzastopce pronalazi podniz koji se sastoji od jednog operatora i dva operanda, nakon čega se na njihovo mjesto umetne rezultat tog operatora nad tim operandima. Ovaj se postupak ponavlja dok ne ostane samo rezultat.

Primjerice, neka je zadan izraz $4*(2+7)$. Prevođenjem ovog izraza na prefiksni oblik dobili bismo $*, 4, +, 2, 7$ (zarezi su dodani radi bolje čitljivosti). Sada pronalazimo prvi podniz koji se sastoji od jednog operatora i dva operanda, što je ovdje $+, 2, 7$, i na njegovo mjesto stavimo rezultat tog podizraza nakon čega nam ostaje $*, 4, 9$. Sada ovaj postupak ponovimo i dobijemo 36 što je rezultat izraza $4*(2+7)$. Vidimo da je izračunavanje aritmetičkih izraza u prefiksnom obliku praktički trivijalno jer ne moramo voditi računa o zagradama.

Postifksni obilazak. Redoslijed čvorova ovog obilaska je lijevo dijete, desno dijete, roditelj (Algoritam 5.22).

Algoritam 5.22 Postfiksni obilazak binarnog stabla.

Parametri: $cvor$ - adresa polaznog čvora

```
1: procedure POSTFIKS(cvor: ELEMENT-BS)
2:     if cvor ≠ NIL then
3:         POSTFIKS(cvor.LIJEVO)
4:         POSTFIKS(cvor.DESNO)
5:         ispiši cvor.KLJUČ
6:     end if
7: end procedure
```

Obilaskom stabla na Slici 5.28 ova bi procedura ispisala $ab+cd-+ef/*$. Vidimo da je ovaj ispis po strukturi sličan prefiksnom ispisu. Ovakav bismo izraz izračunali na isti način kao i prefiksni, s razlikom da ovdje tražimo dva operanda iza kojih se nalazi operator.

Infiksni obilazak. Redoslijed čvorova ovog obilaska je lijevo dijete, roditelj, desno dijete (Algoritam 5.23).

Algoritam 5.23 Infiksni obilazak binarnog stabla.

Parametri: $cvor$ - adresa polaznog čvora

```
1: procedure INFIKS(cvor: ELEMENT-BS)
2:     if cvor ≠ NIL then
3:         INFIKS(cvor.LIJEVO)
4:         ispiši cvor.KLJUČ
5:         INFIKS(cvor.DESNO)
6:     end if
7: end procedure
```

Obilaskom stabla na Slici 5.28 ova bi procedura ispisala $a + b + c - d * e/f$. Ovaj se obilazak zove *infiksni* jer se u ispisu operator nalazi između dva operanda. Vidimo da kod ispisa ovog obilaska gubimo informaciju o redoslijedu operacija. Primjerice, nije jasno dolazi li množenje prije zbrajanja, oduzimanja i dijeljenja. Ovakav je obilazak koristan u radu sa stablom za binarno pretraživanje kada elementima takvog stabla želimo pristupati u sortiranom redoslijedu. Primjerice, za stablo na Slici 5.22 ovaj bi algoritam ispisao $8, 10, 12, 17, 20, 31, 32, 45, 56$.

Obilazak stabla u dubinu Možemo primjetiti da ova tri algoritma rade tako da uvijek idu do kraja na jednu stranu, odnosno do krajnjeg lijevog čvora, a nakon toga do krajnjeg desnog. Zbog toga ih svrstavamo u algoritme *obilaska stabla u dubinu*. Iako smo ove algoritme implementirali pomoću rekurzije, to nije nužno. Algoritam 5.24 prikazuje iterativni način za prefiksni obilazak stabla. Za taj algoritam koristimo petlju gdje u svakoj iteraciji

Tablica 5.2: Stanja stoga tijekom izvršavanja Algoritma 5.24.

Korak	**Operacija**	**Stog**
0	Dodaj početni element na stog, 17	$17 \rightarrow$
1	Uzmi i ispiši idući element sa stoga, 17 Dodaj prvo desno pa onda lijevo dijete elementa 17 na stog	*prazan stog* $31, 8 \rightarrow$
2	Uzmi i ispiši idući element sa stoga, 8 Dodaj desno dijete elementa 8 na stog	$31 \rightarrow$ $31, 12 \rightarrow$
3	Uzmi i ispiši idući element sa stoga, 12 Dodaj lijevo dijete elementa 12 na stog	$31 \rightarrow$ $31, 10 \rightarrow$
4	Uzmi i ispiši idući element sa stoga, 10	$31 \rightarrow$
5	Uzmi i ispiši idući element sa stoga, 31 Dodaj prvo desno pa onda lijevo dijete elementa 31 na stog	*prazan stog* $45, 20 \rightarrow$
6	Uzmi i ispiši idući element sa stoga, 20	$45 \rightarrow$
7	Uzmi i ispiši idući element sa stoga, 45 Dodaj prvo desno pa onda lijevo dijete elementa 45 na stog	*prazan stog* $56, 32 \rightarrow$
8	Uzmi i ispiši idući element sa stoga, 32	$56 \rightarrow$
9	Uzmi i ispiši idući element sa stoga, 56	*prazan stog*

uzmemo čvor sa stoga, ispišemo vrijednost koju sadrži, i zatim na stog stavimo njegovo desno i lijevo dijete (ako postoje). Ovaj postupak ponavljamo sve dok stog ne postane prazan. Stog na početku sadrži samo početni čvor koji je zadan (što je u ovom primjeru korijen) jer moramo početi s nekim čvorom. Ovaj je postupak ilustriran u tablici 5.2 na stablu sa Slike 5.22. Vidimo da stog ovdje ima istu ulogu kao i onaj kod rekurzivnog poziva procedure PREFIKS: svaki takav poziv bi u pozadini na stog stavio mjesto (redak 5) od kojeg treba nastaviti izvršavanje nakon povratka iz prethodnog rekurzivnog poziva u retku 4, a to bi bilo mjesto od kojeg se nastavlja obilazak stabla. Upravo to radimo i u ovoj iterativnoj inačici, samo što u njoj umjesto rekurzivnih poziva na stog (onaj koji smo mi instancirali) smještamo čvor od kojeg treba nastaviti obilazak stabla.

Vremenska složenost ova tri algoritma obilaska u dubinu je $\Theta(n)$ jer se mora doći do svakog elementa i to samo jednom.

Obilazak stabla u širinu. Za razliku od obilaska stabla u dubinu, *obilazak stabla u širinu* radi tako da se čvorovima stabla pristupa po razinama. Isto tako, za razliku od obilaska stabla u dubinu, obilazak u širinu tipično se implementira upotrebom reda. Ovo je prikazano Algoritmom 5.25. Možemo primjetiti da su ovaj algoritam i PREFIKS-ITER gotovo identični, s tim da PREFIKS-ITER koristi stog, a OBILAZAK-U-ŠIRINU red.

Za ilustraciju obilaska stabla u širinu opet ćemo uzeti stablo na Slici 5.22. Obilaskom u širinu ispis čvorova bio bi $17, 8, 31, 12, 20, 45, 10, 32, 56$. Ovakav obilazak možemo implementirati tako da prvo u red stavimo korijen stabla. Nakon toga u petlji uzimamo idući element iz reda, a odmah nakon toga u red dodamo njegovu djecu, recimo slijeva nadesno. Ovo ponavljamo dok u redu ima čvorova. U tablici 5.3 ilustriran je ovaj postupak.
Algoritmi kao PREFIKS-ITER i OBILAZAK-U-ŠIRINU primijenjivi su za bilo kakva stabla, ne samo binarna. Pored toga, takvi su algoritmi primijenjivi i na strukture podataka koje se

Algoritam 5.24 Prefiksni obilazak binarnog stabla iterativno.

Parametri: $cvor$ - adresa polaznog čvora

```
1:  procedure PREFIKS-ITER(cvor: ELEMENT-BS)
2:      stog ← instanca tipa STOG                          ▷ prazan stog
3:      stog.PUSH(cvor)                                    ▷ polazni čvor
4:      while not stog.PRAZAN() do
5:          x ← stog.POP()
6:          ispiši x.KLJUČ
7:          if x.DESNO ≠ NIL then
8:              stog.PUSH(x.DESNO)
9:          end if
10:         if x.LIJEVO ≠ NIL then
11:             stog.PUSH(x.LIJEVO)
12:         end if
13:     end while
14: end procedure
```

Algoritam 5.25 Obilazak binarnog stabla u širinu.

Parametri: $cvor$ - adresa polaznog čvora

```
1:  procedure OBILAZAK-U-ŠIRINU(cvor: ELEMENT-BS)
2:      red ← instanca tipa RED                            ▷ prazan red
3:      red.ENQUEUE(cvor)                                  ▷ polazni čvor
4:      while not red.PRAZAN() do
5:          x ← red.DEQUEUE()
6:          ispiši x.KLJUČ
7:          if x.LIJEVO ≠ NIL then
8:              red.ENQUEUE(x.LIJEVO)
9:          end if
10:         if x.DESNO ≠ NIL then
11:             red.ENQUEUE(x.DESNO)
12:         end if
13:     end while
14: end procedure
```

Tablica 5.3: Stanja reda tijekom izvršavanja Algoritma 5.25.

Korak	Operacija	Red
0	Dodaj početni element u red, 17	$17 \rightarrow$
1	Uzmi i ispiši idući element iz reda, 17 Dodaj prvo lijevo pa onda desno dijete elementa 17 u red	*prazan red* $31, 8 \rightarrow$
2	Uzmi i ispiši idući element iz reda, 8 Dodaj desno dijete elementa 8 u red	$31 \rightarrow$ $12, 31 \rightarrow$
3	Uzmi i ispiši idući element iz reda, 31 Dodaj prvo lijevo pa onda desno dijete elementa 31 u red	$12 \rightarrow$ $45, 20, 12 \rightarrow$
4	Uzmi i ispiši idući element iz reda, 12 Dodaj lijevo dijete elementa 12 u red	$45, 20 \rightarrow$ $10, 45, 20 \rightarrow$
5	Uzmi i ispiši idući element iz reda, 20	$10, 45 \rightarrow$
6	Uzmi i ispiši idući element iz reda, 45 Dodaj prvo desno pa onda lijevo dijete elementa 45 u red	$10 \rightarrow$ $56, 32, 10 \rightarrow$
7	Uzmi i ispiši idući element iz reda, 10	$56, 32 \rightarrow$
8	Uzmi i ispiši idući element iz reda, 32	$56 \rightarrow$
9	Uzmi i ispiši idući element iz reda, 56	*prazan red*

zovu *grafovi*[5].

Kao i kod algoritama za obilazak u dubinu, vremenska složenost algoritma OBILAZAK-U-ŠIRINU je $\Theta(n)$.

5.6.3 Osnovne operacije nad stablom za binarno pretraživanje

Kao i druge strukture podataka, stablo za binarno pretraživanje podržava operacije dodavanja, brisanja i traženja elementa pa se tip podatka za ovakvo stablo može definirati kako je pokazano u definiciji tipa 5.10.

Tip 5.10 Definicija tipa za stablo za binarno pretraživanje.

```
1:  type SBP
2:      NAJMANJI(cvor: ELEMENT-BS)
3:      NAJVEĆI(cvor: ELEMENT-BS)
4:      PRETHODNIK(cvor: ELEMENT-BS)
5:      SLJEDBENIK(cvor: ELEMENT-BS)
6:      NAĐI(v)                                  ▷ traženje čvora s podatkom v
7:      DODAJ(e: ELEMENT-BS)
8:      OBRIŠI(e: ELEMENT-BS)
9:      PREMJESTI(stablo: ELEMENT-SBP, u: ELEMENT-SBP, v: ELEMENT-SBP)

10:     KORIJEN: ELEMENT-BS ← NIL                ▷ adresa korijena stabla
11: end type
```

[5] *Graf* je struktura podataka koja se sastoji od čvorova povezanih lukovima. Stablo je vrsta grafa.

Da bismo znali gdje stablo počinje moramo imati adresu elementa koji predstavlja korijen stabla, za što će nam poslužiti atribut KORIJEN.

Najmanji i najveći element. S obzirom na način na koji su elementi u stablu za binarno pretraživanje organizirani, u takvom stablu jednostavno je naći najmanji i najveći element. Da bismo došli do najmanjeg elementa jednostavno slijedimo lijevu stranu stabla dok ne dođemo do krajnjeg lijevog lista koji je ujedno i najmanji element. Za najveći element na isti način slijedimo desnu stranu. Pronalaženje najmanjeg i najvećeg elementa prikazano je u Algoritmima 5.26 i 5.27.

Algoritam 5.26 Najmanji element stabla za binarno pretraživanje.

Parametri: $cvor$ - adresa polaznog čvora
Rezultat: adresa najmanjeg čvora

```
1: function SBP.NAJMANJI(cvor: ELEMENT-BS)
2:     while cvor.LIJEVO ≠ NIL do
3:         cvor ← cvor.LIJEVO
4:     end while
5:     return cvor
6: end function
```

Algoritam 5.27 Najveći element stabla za binarno pretraživanje.

Parametri: $cvor$ - adresa polaznog čvora
Rezultat: adresa najvećeg čvora

```
1: function SBP.NAJVECI(cvor: ELEMENT-BS)
2:     while cvor.DESNO ≠ NIL do
3:         cvor ← cvor.DESNO
4:     end while
5:     return cvor
6: end function
```

Vremenska složenost ova dva algoritma je $O(h)$ jer ne podrazumijevamo potpuno ili balansirano binarno stablo (primjerice, najmanji element može biti lijevo dijete korijena, dok desna strana stabla može biti puno dublja). Ako bi se radilo o potpunom ili balansiranom binarnom stablu vremenska složenost bila bi $\Theta(h)$ jer bi tada duljina puta od korijena do krajnjeg lijevog ili desnog lista bila oko h.

Sljedbenik i prethodnik. Prethodnik čvora x čvor je s najvećom vrijednošću manjom od x.KLJUČ, a sljedbenik onaj s najmanjom većom. Drugim riječima, sljedbenik je idući čvor koji bismo ispisali infiksnim obilaskom stabla za binarno pretraživanje, a prethodnik onaj prethodni. Primjerice, na Slici 5.22 sljedbenik čvora 12 je čvor 17, a sljedbenik čvora 17 je čvor 20. Algoritmom 5.28 prikazano je određivanje sljedbenika zadanog čvora. Za

zadani čvor x taj algoritam radi na sljedeći način: ako postoji desno podstablo čvora x, vrati minimum tog podstabla. U suprotnom, traži najnižeg pretka od x čije je lijevo dijete predak od x (ili sam x). U primjeru sa Slike 5.22 najniži predak od 12 čije je lijevo dijete predak od 12 je 17 jer lijevo dijete od 17 je 8, a taj je čvor predak čvora 12. Ovaj postupak možemo i opisati tako da kažemo da ako ne postoji desno podstablo, penjemo se po stablu dok ne dođemo do lijevog djeteta nekog čvora s; s je tada sljedbenik čvora x. Kao još jedan primjer s iste slike, čvor 10 nema desno dijete. S obzirom da je 10 lijevo dijete svog roditelja, to jest čvora 12, čvor 12 sljedbenik čvora 10.

Algoritam 5.28 Sljedbenik čvora stabla za binarno pretraživanje.

Parametri: $cvor$ - adresa polaznog čvora
Rezultat: sljedbenik polaznog čvora

```
1: function SBP.SLJEDBENIK(cvor: ELEMENT-BS)
2:     if cvor = NIL then
3:         return NIL
4:     end if
5:     if cvor.DESNO ≠ NIL then
6:         return NAJMANJI(cvor.DESNO)
7:     else
8:         sljedbenik ← cvor.RODITELJ
9:         while sljedbenik ≠ NIL and cvor = sljedbenik.DESNO do
10:            cvor ← sljedbenik
11:            sljedbenik ← sljedbenik.RODITELJ
12:        end while
13:    end if
14:    return sljedbenik
15: end function
```

Određivanje prethodnika prikazano je Algoritmom 5.29. Taj algoritam radi na simetričan način za zadani čvor x: ako čvor ima lijevo podstablo, prethodnik je maksimum tog podstabla. U suprotnom, penjemo se po stablu dok ne dođemo do desnog djeteta nekog čvora p. p je tada prethodnik od x. Primjerice, prethodnik čvora 10 na Slici 5.22 je 8 jer 8 je roditelj desnog djeteta 12.

Oba ova algoritma imaju vremensku složenost $O(h)$ jer je broj operacija određen duljinom puta od jednog čvora do drugog prema gore ili dolje u stablu.

Pretraživanje. Za ovu operaciju koristimo prethodno opisano načelo binarnog pretraživanja: počnemo od korijena s kojim uspoređujemo traženu vrijednost. Na osnovu te usporedbe nastavljamo s lijevim ili desnim podstablom. Ovo ponavljamo sve dok ne naiđemo na traženi element ili dok ne dođemo do nekog lista stabla. U tom slučaju operacija vraća NIL kao rezultat, što znači da traženi element ne postoji u stablu. Ako nađemo čvor koji sadrži traženu vrijednost, kao rezultat vraćamo adresu tog čvora jer bismo ga mogli koristi za druge operacije sa stablom, kao što je brisanje. Ovaj je postupak prikazan Algoritmom

Algoritam 5.29 Prethodnik čvora stabla za binarno pretraživanje.

Parametri: $cvor$ - adresa polaznog čvora
Rezultat: prethodnik polaznog čvora

```
1: function SBP.PRETHODNIK(cvor: ELEMENT-BS)
2:     if cvor = NIL then
3:         return NIL
4:     end if
5:     if cvor.LIJEVO ≠ NIL then
6:         return NAJVEĆI(cvor.LIJEVO)
7:     else
8:         prethodnik ← cvor.RODITELJ
9:         while prethodnik ≠ NIL and cvor = prethodnik.LIJEVO do
10:            cvor ← prethodnik
11:            prethodnik ← prethodnik.RODITELJ
12:        end while
13:    end if
14:    return prethodnik
15: end function
```

5.30. Sličan postupak možemo primijeniti za dodavanje novog elementa u stablo za binarno pretraživanje.

Teorem 2 *Za stablo za binarno pretraživanje visine h, operacije* NAJMANJI, NAJVEĆI, PRETHODNIK, SLJEDBENIK *i* NAĐI *mogu se implementirati tako da im je vremenska složenost $O(h)$.* [1]

Dodavanje elemenata. S obzirom da u stablu već mogu postojati čvorovi koji su prethodno dodani, da bismo dodali novi čvor moramo pronaći odgovarajući list u stablu na koji ćemo ga nadovezati. Primjerice, ako u stablo prikazano na Slici 5.23 želimo dodati vrijednost 14, ta bi se vrijednost nadovezala na čvor s brojem 12 kojem bi bila desno dijete. Radimo isto kao da tražimo broj 12, uzimajući u obzir njegovu vrijednost na osnovu koje u svakom čvoru odlučujemo nastavljamo li dalje s lijevim ili desnim podstablom, sve dok ne dođemo do odgovarajućeg krajnjeg čvora (redovi $6..13$). Varijabla *roditelj* sadrži adresu roditelja svakog čvora do kojeg se došlo i služi za povezivanje posljednjeg čvora s novim. Ovaj je postupak prikazan Algoritmom 5.31.

Algoritam 5.31 vremenske je složenosti $O(h)$ za stablo visine h. Ovdje opet treba imati na umu da ovisimo o visini stabla, a ne o broju elemenata u njemu, tako da balansiranost stabla ima utjecaja na učinkovitost algoritma.

Brisanje elemenata. Brisanje elemenata iz stabla za binarno pretraživanje malo je složenije od dodavanja jer elemente koji su povezani s onim koji se briše moramo povezati s nekim drugim elementom. Da bismo obrisali element z iz stabla za binarno pretraživanje moramo u obzir uzeti sljedeće slučajeve:

Algoritam 5.30 Pretraživanje stabla za binarno pretraživanje.

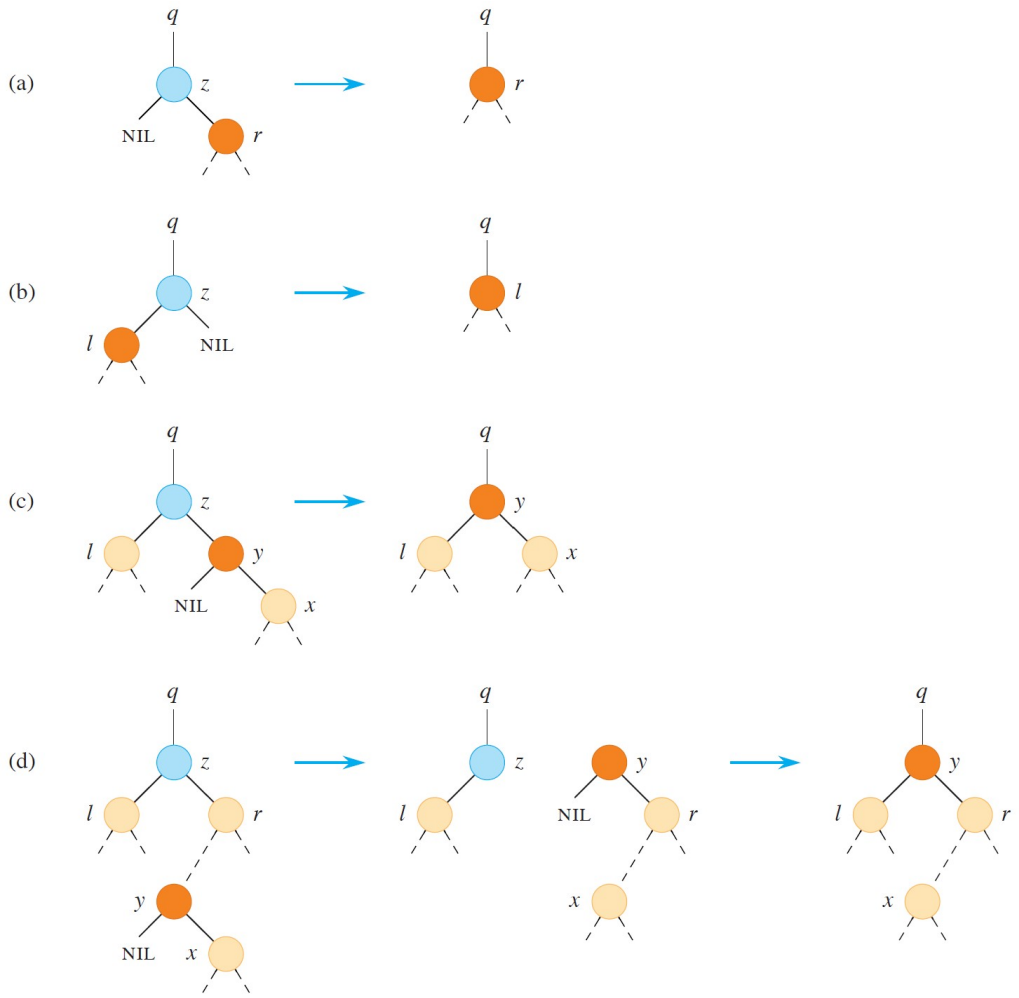
Parametri: k - ključ koji tražimo
Rezultat: adresa čvora s ključem k ili NIL ako takav čvor ne postoji u stablu

```
1: function SBP.NAĐI(k)
2:     cvor ← KORIJEN
3:     while cvor ≠ NIL do
4:         if cvor.KLJUČ = k then
5:             return cvor
6:         else
7:             if k < cvor.KLJUČ then
8:                 cvor ← cvor.LIJEVO
9:             else
10:                cvor ← cvor.DESNO
11:            end if
12:        end if
13:    end while
14:    return NIL
15: end function
```

Algoritam 5.31 Dodavanje elementa u stablo za binarno pretraživanje.

Parametri: e - element koji dodajemo

```
1: procedure SBP.DODAJ(e: ELEMENT-SBP)
2:     cvor ← KORIJEN
3:     if cvor = NIL then
4:         KORIJEN ← cvor
5:     else
6:         while cvor ≠ NIL do
7:             roditelj ← cvor
8:             if e.KLJUČ < cvor.KLJUČ then
9:                 cvor ← cvor.LIJEVO
10:            else
11:                cvor ← cvor.DESNO
12:            end if
13:        end while
14:        e.RODITELJ ← roditelj
15:        if e.KLJUČ < roditelj.KLJUČ then
16:            roditelj.LIJEVO ← e
17:        else
18:            roditelj.DESNO ← e
19:        end if
20:    end if
21: end procedure
```



Slika 5.29: Brisanje elementa iz stabla za binarno pretraživanje [1]. Briše se element označen plavom bojom, a na njegovo mjesto dolazi podstablo s korijenom označenim narančastom bojom.

1. Ako z nema djece, odgovarajući atribut njegovog roditelja postavimo na NIL.
2. Ako z ima jedno dijete, z zamijenimo tim djetetom tako da postavimo odgovarajući atribut roditelja na to dijete.
3. Ako z ima oba djeteta, zamijenimo ga njegovim sljedbenikom (prema vrijednosti) koji će se nalaziti u desnom podstablu čvora z.

Postupak brisanja elementa iz stabla za binarno pretraživanje ilustriran je na Slici 5.29. Slučajevi a) i b) su trivijalni jer za njih samo "podignemo" lijevo ili desno podstablo čvora z. Slučajevi c) i d) su malo složeniji jer čvor z koji brišemo ima oba djeteta pa moramo pronaći čvor koji će zamijeniti z i koji ima najviše jedno dijete. Drugim riječima, trebamo naći sljedbenika čvora z jer on neće imati lijevo dijete. U slučaju c) vidimo da čvor y nema lijevo dijete, odnosno y je sljedbenik od z, pa u tom slučaju y dolazi na mjesto z. U slučaju d) desno dijete od z (čvora koji brišemo), odnosno r, ima oba djeteta pa stoga r nije sljedbenik od z. Čvor y nema lijevo dijete pa je on sljedbenik od z i njega postavljamo na mjesto z. Desno dijete od y tada postaje lijevo dijete od r.

Kao što se vidi iz prethodnog opisa, brisanje elementa iz stabla za binarno pretraživanje uključuje premještanja nekog podstabla na drugo mjesto u stablu. Zbog toga ćemo definirati pomoćnu operaciju PREMJESTI (Algoritam 5.32) koja ima tri parametra: $stablo$ (korijen stabla), u (mjesto na koje se umeće podstablo), v (korjien podstabla koje premještamo na poziciju u).

Algoritam 5.32 Premještanje podstabla unutar stabla za binarno pretraživanje.

Parametri: $stablo$ - adresa korijena stabla; u - mjesto na koje se umeće podstablo; v - korijen podstabla koje premještamo na poziciju u

```
1:  procedure SBP.PREMJESTI(stablo: ELEMENT-SBP, u: ELEMENT-SBP, v: ELEMENT-SBP)
2:      if u.RODITELJ = NIL then
3:          stablo.KORIJEN ← v
4:      else
5:          if u = u.RODITELJ.LIJEVO then
6:              u.RODITELJ.LIJEVO ← v
7:          else
8:              u.RODITELJ.DESNO ← v
9:          end if
10:         if v ≠ NIL then
11:             v.RODITELJ ← u.RODITELJ
12:         end if
13:     end if
14: end procedure
```

S obzirom da svaki redak algoritma OBRIŠI osim poziva operacije NAJMANJI u redku 8 treba konstantno vrijeme, vremenska složenost algoritma OBRIŠI je $O(h)$ za stablo visine h.

Za stablo za binarno pretraživanje visine h vrijedi Teorem 3.

Teorem 3 *Za binarno stablo pretraživanja visine h, operacije* DODAJ *i* OBRIŠI *mogu se implementirati tako da im je vremenska složenost $O(h)$* [1].

Algoritam 5.33 Brisanje čvora iz stabla za binarno pretraživanje.

Parametri: $stablo$ - korijen stabla; z - adresa čvora koji brišemo

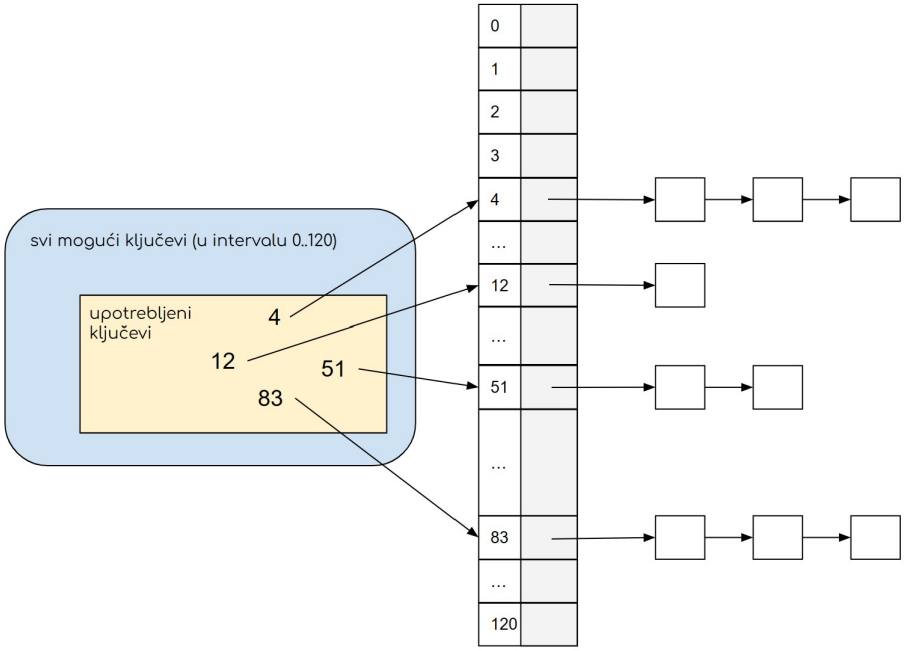
```
1: procedure SBP.OBRIŠI(stablo, z)
2:     if z.LIJEVO = NIL then
3:         PREMJESTI(stablo, z, z.DESNO)
4:     else
5:         if z.DESNO = NIL then
6:             PREMJESTI(stablo, z, z.LIJEVO)
7:         else
8:             y ← NAJMANJI(z.DESNO)
9:             if y ≠ z.DESNO then
10:                PREMJESTI(stablo, y, y.DESNO)
11:                y.DESNO ← z.DESNO
12:                y.DESNO.RODITELJ ← y
13:            end if
14:            PREMJESTI(stablo, z, y)
15:            y.LIJEVO ← z.LIJEVO
16:            y.LIJEVO.RODITELJ ← y
17:        end if
18:    end if
19: end procedure
```

5.7 Tablica s izravnim adresiranjem i tablica raspršivanja

Prethodno smo proučili dvije tehnike traženja zadanog ključa: linearno i binarno pretraživanje. Binarno pretraživanje dalo nam je znatno bolje rezultate od linearnog jer smo, u najgorem slučaju, mogli izbjeći pretraživanje velike količine podataka. Primjerice, za milijun vrijednosti u nizu binarnim bismo pretraživanjem morali usporediti oko dvadesetak ključeva da bismo saznali da neki od njih ne postoji u nizu. U ovom dijelu proučit ćemo još dvije strukture podataka koje podržavaju učinkovite operacije pronalaženja ključa: tablica s izravnim adresiranjem i njena generalizacija, tablica raspršivanja.

5.7.1 Tablica s izravnim adresiranjem.

Tablica s izravnim adresiranjem (engl. *direct address table*) [1] generalizacija je običnog niza čija veličina odgovara broju svih mogućih ključeva. Za primjer pretpostavimo da u memoriji želimo spremati podatke o studentima prema broju ostvarenih ECTS bodova na studiju tako da na najučinkovitiji način imamo pristup svim studentima koji su ostvarili određeni broj bodova. Ključ nam je ovdje broj koji predstavlja broj ECTS bodova (između 0 i 120). Pretpostavit ćemo da svaki indeks niza sadrži adresu liste u kojoj se nalaze podaci o studentima s ostvarenim istim brojem bodova koji odgovara tom indeksu. Ova struktura podataka prikazana je na Slici 5.30. Sada je jednostavno doći do svih studenata koji su ostvarili, recimo, 51 ECTS bod – jednostavno dohvatimo listu koja se nalazi na indeksu 51. Vidimo da ne moramo tražiti odgovarajuću listu jer podatak o njezinoj lokaciji dobiva se



Slika 5.30: Izravno adresiranje.

izravno adresiranjem niza.

U definiciji tipa 5.11 prikazan je tip podatka za tablicu s izravnim adresiranjem. Taj se tip sastoji od triju operacija: dohvaćanja podatka na osnovu ključa k, dodavanja novog ključa k s vrijednošću v i brisanja ključa k. Atribut *niz* je niz koji sadrži sve podatke tablice i inicijalno je svaki indeks tog niza postavljen na NIL.

Tip 5.11 Definicija tipa za tablicu s izravnim adresiranjem.

```
1: type TABLICA
2:     DOHVATI(k)                          ▷ vraća element na indeksu k
3:     DODAJ(k, v)                         ▷ dodaje novi ključ k s vrijednošću v
4:     OBRIŠI(k)                           ▷ briše ključ k

5:     NIZ[0..MAKS] ← NIL                  ▷ svaki je atribut incijaliziran na NIL
6: end type
```

Operacije DOHVATI, DODAJ i OBRIŠI trivijalne su za implementirati i prikazane su u Algoritmima 5.34, 5.35 i 5.36.
Vidimo da je vremenska složenost ovih triju operacija O(1).

Tablica s izravnim adresiranjem izuzetno je učinkovita struktura podataka, ali ima i nedostatke. Kao što Slika 5.30 pokazuje, postoji mogućnost da velik dio niza u koji se smještaju podaci ostane neiskorišten. To se može dogoditi kada je broj upotrebljenih ključeva znatno manji od broja svih mogućih ključeva. Ovo nije problem kada je broj svih mogućih ključeva mali u odnosu na raspoloživu memoriju računala, ali ako je velik značajan dio niza može ostati neiskorišten, nepotrebno zauzimajući memoriju. Drugi

Algoritam 5.34 Dohvaćanje elementa tablice s izravnim adresiranjem.

Parametri: k - indeks elementa koji dohvaćamo
Rezultat: element na indeksu k

```
1: function TABLICA.DOHVATI(k)
2:     return NIZ[k]
3: end function
```

Algoritam 5.35 Dodavanje elementa u tablicu s izravnim adresiranjem.

Parametri: k - indeks elementa; v - novi element

```
1: procedure TABLICA.DODAJ(k, v)
2:     NIZ[k] ← v
3: end procedure
```

problem tablica s izravnim adresiranjem je što ključ mora biti isključivo pozitivni cijeli broj ili nula jer predstavlja indeks niza.

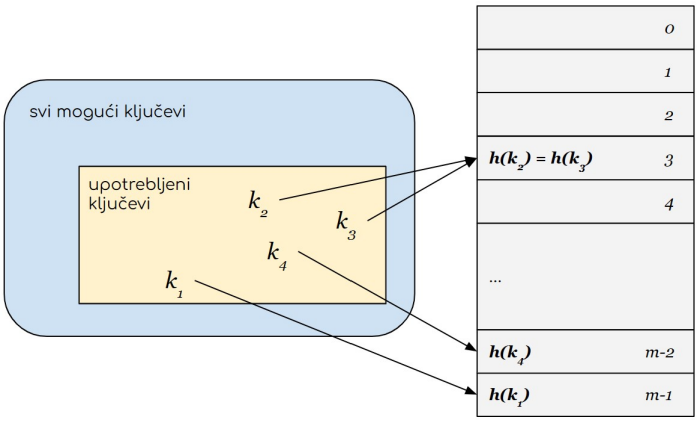
5.7.2 Tablica raspršivanja

Tablica raspršivanja (engl. *hash table*) još je jedna temeljna struktura podataka koja se koristi za učinkovito pohranjivanje i dohvaćanje podataka [1, 13, 24]. Tablicu raspršivanja možemo promatrati kao generalizaciju tablice s izravnim adresiranjem, ali kod koje elementima pristupamo putem ključa umjesto numeričkog indeksa. Za razliku od tablice s izravnim adresiranjem, ključ ovdje ne predstavlja indeks polja u nizu. Da bismo ostvarili podjednako dobre performanse kao one indeksiranja niza, potrebno je u što većoj mjeri izbjeći traženje ključa. Osnovna ideja iza tablice raspršivanja je da, umjesto da ga se traži, položaj ključa u nizu se izračuna se na osnovi samog ključa. Time se također izbjegava nepotrebno pretraživanje velikog broja ključeva. Za razliku od tablice s izravnim adresiranjem, tablica raspršivanja je struktura podataka koja može učinkovito raditi s brojem aktivnih (upotrebljenih) ključeva koji je puno manji od broja svih mogućih ključeva. Nadalje, ključ tablice raspršivanja ne mora isključivo biti numeričkog tipa. Kod tablice raspršivanja to se postiže upotrebom *funkcije raspršivanja* (engl. *hash function*).

Algoritam 5.36 Brisanje elementa iz tablice s izravnim adresiranjem.

Parametri: k - indeks elementa koji brišemo

```
1: procedure TABLICA.OBRIŠI(k)
2:     NIZ[k] ← NIL
3: end procedure
```



Slika 5.31: Tablica raspršivanja kapaciteta m.

Funkcija raspršivanja

S obzirom da i tablica raspršivanja u pozadini upotrebljava niz za smještanje ključeva i vrijednosti, ključ se mora transformirati u numeričku vrijednost koja odgovara nekom indeksu tog niza. To se može postići na više načina, zavisno od karakteristika ključa. Ako, primjerice, koristimo tekstualne ključeve na osnovi njihove duljine možemo odrediti indeks na kojem se nalazi vrijednost pridružena takvom ključu. Algoritmom 5.37 prikazan je primjer fukcije raspršivanja koja za tekstualni ključ vraća njegovu duljinu.

Algoritam 5.37 Primjer jednostavne funkcije raspršivanja.

Parametri: k - ključ nad kojim treba primjeniti funkciju raspršivanja
Rezultat: vrijednost funkcije raspršivanja (engl. *hash code*)

```
1: function INDEKS(k)
2:     return duljina(k)
3: end function
```

Kod tablice raspršivanja podatak s ključem k i funkcijom raspršivanja h nalazi se na indeksu $h(k)$. Na Slici 5.31 prikazana je struktura tablice raspršivanja koja se temelji na nekoj funkciji raspršivanja h. Vidimo da su ključevi k_2 i k_3 smješteni na isti indeks tablice. Pojavu kada se dva ili više ključeva moraju smjestiti na isti indeks tablice raspršivanja nazivamo *kolizija*. One mogu nastati iz više razloga. Jedan od njih je da funkcija raspršivanja za dva ili više ključeva vraća isti kôd. Ovdje se opet možemo poslužiti primjerom funkcije INDEKS – ako postoje dva ili više ključeva s istom duljinom, ova će ih funkcija smjestiti na isti indeks u nizu.

Ako je broj upotrebljenih ključeva veći od kapaciteta niza u koji se oni smještaju, neki će indeksi morat sadržavati više ključeva. Ovo je drugi razlog iz kojeg nastaju kolizije. Primjerice, za veličinu tablice 10 i broj ključeva 20 koji su u njoj smješteni, na nekim indeksima nalazit će se više ključeva. Jedno od riješenja je da se tablica poveća tako da je broj ključeva dovoljno mali u odnosu na njenu veličinu. Važno je imati na umu da je ovdje

i dalje važna kvaliteta funkcije raspršivanja, odnosno njeno svojstvo da za svaki ključ vrati jedinstven kôd.

Nezavisno ravnomjerno raspršivanje

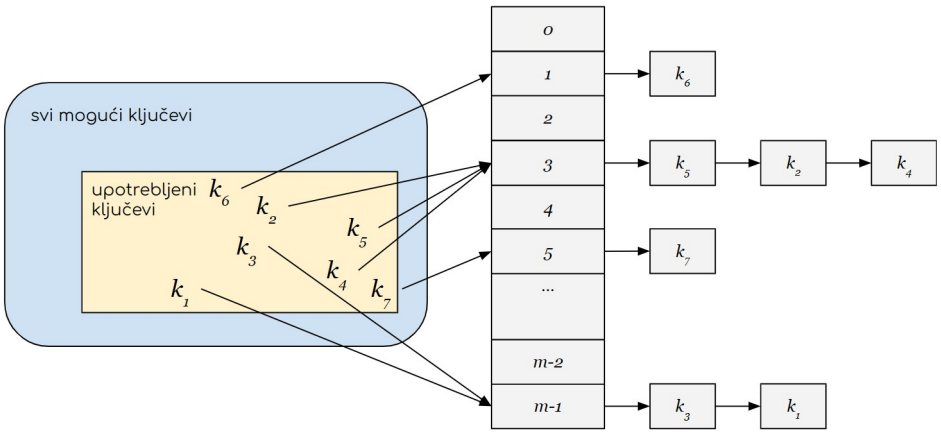
Pretpostavimo da u tablicu raspršivanja želimo spremati podatke o europskim gradovima gdje je ključ ime grada, upotrebom funkcije INDEKS kao funkcije raspršivanja. Očito je da bi ova funkcija većinu naziva gradova grupirala oko nekoliko indeksa niza jer se većina naziva gradova sastoji od podjednakog broja slova, recimo između 4 i 8. Rekli bismo da funkcija INDEKS ne distribuira ključeve ravnomjerno. Ravnomjerna distribucija ključeva tablice raspršivanja poželjna je karakteristika funkcije raspršivanja. Jedan pojam koji je više od teorijskog značaja je *nezavisna ravnomjerna funkcija raspršivanja* (engl. *independent uniform hash function*). To je "idealna" funkcija raspršivanja kod koje je za svaki ključ k (iz skupa svih mogućih ključeva) svaka vrijednost $h(k)$ iz domene $\{0, 1, ..., m-1\}$ jednako vjerojatna. To možemo reći i ovako: $h(k)$ je element koji je slučajno i nezavisno od drugih elemenata odabran iz skupa elemenata $\{0, 1, ..., m-1\}$. Ovakvu apstraktnu funkciju raspršivanja možemo predočiti na sljedeći način. Neka imamo šest mjesta u tablici, s indeksima $0..5$. Pretpostavimo da za neki ključ kao što je $zagreb$ bacamo kocku da bismo dobili indeks na koji ćemo smjestiti taj ključ. Jasno je da bacanjem kocke svaki indeks tablice ima jednaku vjerojatnost, odnosno $1/6$. Sada, ako ponovo dobijemo isti ključ, ovakva bi funkcija trebala vratiti isti indeks koji je dobiven prvi put za taj ključ. S praktičnog stajališta, ovakvu bi funkciju raspršivanja bilo teško učinkovito implementirati jer bismo na neki način morali pamtiti koji je indeks dobiven za koji ključ, što bi poništilo prednosti tablice raspršivanja. Zbog toga je ovakva funkcija samo od teorijskog značaja i koristi se u analizi performansi tablice raspršivanja gdje se često pretpostavlja da su elementi nezavisno i ravnomjerno raspoređeni. U praksi nam je cilj dobiti dobru aproksimaciju ovakve funkcije.

Rad s kolizijama

Iako je glavni cilj definiranja dobre funkcije raspršivanja izbjeći kolizije, one su u praksi moguće pa je stoga potrebno imati strategiju njihovim upravljanjem. U ovom dijelu opisan je jednostavan način upravljanja kolizijama koji se zove *ulančavanje* (engl. *chaining*), a u nastavku će biti opisana još jedna tehnika rješavanja kolizija koja se zove *otvoreno adresiranje* (engl. *open addressing*).

Rješavanje kolizija ulančavanjem. Ulančavanje je tehnika rješavanja kolizija kod koje se svi ključevi koje treba smjestiti na isti indeks niza tablice raspršivanja spreme u povezanu listu, zajedno s adresom pripadajuće vrijednosti. Ako treba pronaći zadani ključ pretražuje se ta povezana lista, obično linearnim pretraživanjem. Ovakva struktura tablice raspršivanja prikazana je na Slici 5.32. Pristup ključu kod ovakve strukture podataka sastoji se u tome da se prvo odredi indeks na kojem se on nalazi upotrebom funkcije raspršivanja. Nakon toga unutar liste na tom se indeksu linearnim pretraživanjem pronađe traženi ključ.

Faktor opterećenja. Prilikom analize performansi tablice raspršivanja koristit ćemo vrijednost koja se zove *faktor opterećenja* (engl. *load factor*). Za tablicu veličine m i broj



Slika 5.32: Tablica raspršivanja kod koje se kolizije rješavaju ulančavanjem. Kolizije u ovom primjeru nastaju jer je $h(k_2) = h(k_4) = h(k_5)$ i $h(k_1) = h(k_3)$.

ključeva n faktor raspršivanja α je n/m. Ta vrijednost može biti veća od, manja od ili jednaka 1.

Performanse. Ako pretpostavimo da želimo koristiti tablicu raspršivanja s ulančavanjem, kakve performanse možemo očekivati od takve strukture podataka sa stajališta vremena potrebnog da se pronađe zadani ključ? Učinkovitost ovakvog pristupa ključevima ovisi o načinu na koji su distribuirani. Primjerice, za milijun vrijednosti koje su raspoređene u samo dvije liste od $500,000$ vrijednosti, pretraživanje neće biti učinkovito kao kada bi bile raspoređene u 20.000 lista od 50 vrijednosti. U ovom drugom slučaju, nakon što bismo izračunali indeks liste u kojoj se nalazi vrijednost sa zadanim ključem, pretraživali bismo puno manji skup vrijednosti. U najgorem slučaju svi su ključevi smješteni na isti indeks pa se pretraživanje svodi na linearno pretraživanje. Time bi vremenska složenost bila $\Theta(n)$ u najgorem slučaju.

Kakve su prosječne performanse tablice pretraživanja? One prvenstveno ovise o "kvaliteti" funkcije raspršivanja, odnosno koliko ravnomjerno ona raspoređuje ključeve u tablici. Za analizu prosječne vremenske složenosti pretpostavit ćemo da funkcija raspršivanja raspoređuje ključeve po načelu nezavisnog ravnomjernog raspršivanja. Ovdje ćemo uzeti u obzir dva slučaja: kada traženi ključ ne postoji i kada postoji. Teorem 4 pokriva prvi slučaj, odnosno kada traženi ključ ne postoji.

Teorem 4 *U tablici raspršivanja u kojoj se kolizije rješavaju ulančavanjem, neuspješno pretraživanje ima prosječnu vremensku složenost $\Theta(1+\alpha)$, pod pretpostavkom nezavisnog ravnomjernog raspršivanja.* [1]

Teorem 5 pokriva drugi slučaj, odnosno kada traženi ključ postoji.

Teorem 5 *U tablici raspršivanja u kojoj se kolizije rješavaju ulančavanjem, uspješno pretraživanje ima prosječnu vremensku složenost $\Theta(1+\alpha)$, pod pretpostavkom nezavisnog ravnomjernog raspršivanja* [1].

Oba teorema intuitivno su jasna – vremenska složenost računanja funkcije raspršivanja i pristupa indeksu tablice je $O(1)$. S obzirom da će duljina liste u prosjeku biti α, to će ujedno biti i očekivani broj elemenata koje treba ispitati, pa će ukupna vremenska složenost biti $\Theta(1+\alpha)$.

Uz pretpostavku nezavisnog ravnomjernog raspršivanja, u tablici u kojoj broj elemenata n nije veći od veličine tablice m (odnosno, $\alpha = n/m = 1$), ključevi u tablici bit će ravnomjerno raspoređeni pa će time i liste sadržavati mali broj elemenata (najčešće jedan ako je n blizu m). Možemo zaključiti da je vremenska složenost pronalaženja ključa u tablici raspršivanja s ulančavanjem u prosjeku $O(1)$.

Otvoreno adresiranje

Otvoreno adresiranje (engl. *open addressing*) tehnika je rješavanja kolizija kod koje se svi elementi nalaze unutar same tablice raspršivanja. Svaki indeks, prema tome, sadrži ili element tablice ili NIL (ili neku drugu oznaku praznog mjesta). Posljedica ovoga je to da α (faktor opterećenja) ne može biti veći od 1 kao kod tehnike ulančavanja.

Rješavanje kolizija. Pri dodavanju novog elementa prvo se odredi početni indeks u koji se on može spremiti. Ako je taj indeks zauzet (kolizija) odredi se novi, sljedeći indeks koji nije nužno idući po redu. Ovaj se postupak ponavlja dok se ne pronađe slobodan indeks. Kod traženja ključa ispitujemo indekse tablice po istom ovom načelu dok ne nađemo zadani ključ ili dok ne dođemo do praznog indeksa, što bi značilo da traženi ključ ne postoji u tablici.

Dodavanje ključa. Za dodavanje novog ključa kod otvorenog adresiranja koristi se funkcija raspršivanja koja ima dva parametra – ključ i broj *ispitivanja* ili *proba*. Ispitivanje (engl. *probe*) je provjera je li neki indeks u tablici slobodan (sadrži NIL). Da bismo dodali novi element potrebno je doći do slobodnog mjesta u tablici nizom ispitivanja $h(k,0), h(k,1), ..., h(k,m-1)$ (gdje je *m* veličina tablice). Svaki ovakav niz ispitivanja mora biti jedna permutacija niza svih indeksa tablice $[0,1,...,m-1]$ čime se osigurava da je eventualno svaki indeks tablice ispitan. Dodavanje ključa izvodi operacija DODAJ prikazana Algoritmom 5.38.

Traženje ključa. Traženje ključa odvija se na isti način – ispitujemo indekse dok ne dođemo do praznog mjesta, što znači da se traženi ključ ne nalazi u tablici. Traženje ključa izvodi operacija DOHVATI prikazana Algoritmom 5.39.

Brisanje ključa. Brisanje ključa kod otvorenog adresiranja složenije je nego kod ulančavanja. Indeks na kojem se nalazi ključ koji brišemo ne možemo jednostavno označiti s NIL jer bismo time poremetili traženje ključa. Brisanje ključa bit će opisano kasnije.

Funkcije raspršivanja za otvoreno adresiranje

Kod otvorenog adresiranja idealna funkcija raspršivanja ona koja zadovoljava svojstvo *nezavisnog ravnomjernog raspršivanja permutacije* (engl. *independent uniform permutation hashing*). To znači da je ispitivanje za svaki ključ jedna jednako vjerojatna permutacija

Algoritam 5.38 Dodavanje ključa u tablicu raspršivanja s otvorenim adresiranjem [1].

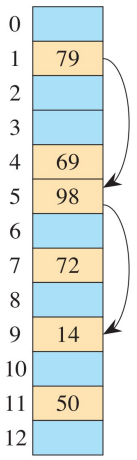
Parametri: k - ključ koji dodajemo
Rezultat: indeks na koji se smješta dodani ključ

```
1: function TABLICA.DODAJ(k)
2:     proba ← 0
3:     repeat
4:         indeks ← h(k, proba)
5:         if niz[indeks] = NIL then
6:             niz[indeks] ← k
7:             return indeks
8:         else
9:             proba ← proba + 1
10:        end if
11:    until proba = m
12:    greška "prepunjena tablica"
13: end function
```

Algoritam 5.39 Traženje ključa u tablici raspršivanja s otvorenim adresiranjem [1].

Parametri: k - ključ koji tražimo
Rezultat: indeks na kojem se nalazi traženi ključ

```
1: function TABLICA.DOHVATI(K)
2:     proba ← 0
3:     repeat
4:         indeks ← h(k, proba)
5:         if niz[indeks] = k then
6:             return indeks
7:         end if
8:         proba ← proba + 1
9:     until niz[indeks] = NIL or proba = m
10:    return NIL
11: end function
```



Slika 5.33: Primjer umetanja ključeva pomoću dvostrukog raspršivanja. Veličina tablice je 13, $h_1(k) = k \, mod \, 13$, $h_2(k) = 1 + (k \, mod \, 11)$. Prikazana je kolizija za ključ 14 koji se smješta na indeks 9 jer su indeksi 1 i 5 već zauzeti [1].

niza $[0, 1, ..., m - 1]$. Iako je ovo teško postići u praksi, moguće je napraviti prihvatljivu aproksimaciju.

Dvostruko raspršivanje

Dvostruko raspršivanje smatra se jednom od najboljih metoda za otvoreno adresiranje jer daje permutacije koje dobro aproksimiraju gore opisano svojstvo. Dvostruko raspršivanje koristi funkciju raspršivanja oblika

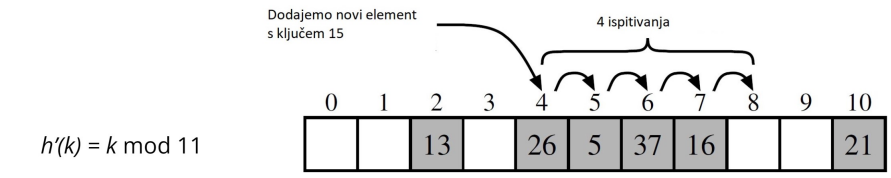
$$h(k, i) = (h_1(k) + ih_2(k)) \textbf{ mod } m$$

gdje su h_1 i h_2 *pomoćne funkcije raspršivanja*. Početno ispitivanje ide na indeks $niz[h_1(k)]$, dok su naredna ispitivanja pomaknuta za $h_2(k)$, i sve **mod** m.

Slika 5.33 prikazuje primjer dodavanja ključeva pomoću dvostrukog raspršivanja. Isprobavanja teku ovako:

```
h(79, 0) = 1
h(69, 0) = 4
h(72, 0) = 7
h(98, 0) = 7;  h(98, 1) = 5
h(14, 0) = 1;  h(14, 1) = 5;  h(14, 2) = 9
h(50, 0) = 11
```

Vidimo da za ključ 98 dolazi do kolizije jer je indeks 7 već zauzet pa se u idućem ispitivanju ($proba = 1$) taj ključ sprema na indeks 5. Isto tako, za ključ 14 dolazi do kolizije jer su indeksi 1 i 5 već zauzeti pa se u trećem ispitivanju ($proba = 2$) on sprema na indeks 9.



Slika 5.34: Dodavanje elementa 15 u tablicu raspršivanja linearnim ispitivanjem. Hash funkcija je $h'(k) = k\, mod\, 11$ [13].

Linearno ispitivanje

Linearno ispitivanje jednostavna je inačica dvostrukog raspršivanja gdje je funkcija raspršivanja oblika

$$h(k,i) = (h'(k) + i) \textbf{ mod } m$$

Prvo se ispituje indeks $niz[h'(k)]$, slijede ispitivanja (ako je taj indeks zauzet) $niz[h'(k)+1]$, $niz[h'(k)+2]$, ..., $niz[m-1]$, $niz[0]$, $niz[1]$, ..., sve do indeksa $niz[h'(k)-1]$. Na Slici 5.34 prikazan je primjer dodavanja novog elementa linearnim ispitivanjem.

Brisanje elementa tablice raspršivanja s linearnim ispitivanjem. U ovom paragrafu opisan je algoritam za brisanje elementa iz tablice raspršivanja s linearnim ispitivanjem. Taj se algoritam temelji na "inverznoj" funkciji raspršivanja u odnosu na $h(k,i) = (h'(k) + i)$ **mod** m. Funkcija h preslikava ključ k i broj ispitivanja i na indeks tablice. Inverzna funkcija g preslikava ključ k i indeks tablice q ($0 \leq q < m$) na broj ispitivanja kojim se dolazi do q:

$$g(k,q) = (q - h'(k)) \textbf{ mod } m$$

S obzirom da je g inverzna funkcija raspršivanja u odnosu na h, vrijedi sljedeće: ako je $h(k,i) = q$ onda je $g(k,q) = i$ i $h(k, g(k,q)) = q$.

Algoritam 5.40 prikazuje brisanje elementa iz tablice raspršivanja s linearnim ispitivanjem. Ovaj algoritam radi tako da prvo (redak 2) postavi indeks na kojem se nalazi ključ za brisanje na NIL. Nakon toga traži indeks q' koji sadrži ključ koji treba pomaknuti na indeks koji je upravo postao slobodan. U redku 11 provjerava se treba li ključ k' na indeksu q' pomaknuti na ispražnjeni indeks q da bi k' ostao dostupan. Ako je $g(k',q) < g(k',q')$ to znači da je prilikom dodavanja ključa k' indeks q je bio ispitan i zauzet. Međutim, taj indeks (q) je sada prazan pa se ključ k' pomiče na indeks q (redak 12). Taj se postupak nastavlja da se vidi treba li sada neki ključ sada staviti na indeks q' koji se upravo oslobodio zbog pomicanja k'. Ovaj postupak ilustriran je na Slici 5.35.

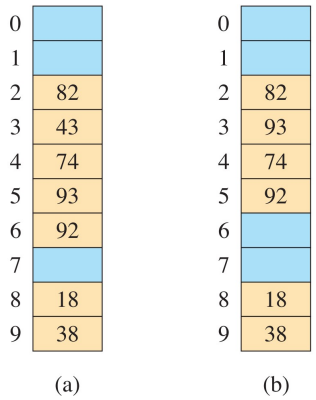
Performanse tablice raspršivanja s linearnim ispitivanjem. Za neuspješno traženje ključa vrijedi sljedeći teorem:

Teorem 6 *Pod pretpostavkom nezavisnog ravnomjernog raspršivanja permutacije i da nema brisanja elemenata, za tablicu raspršivanja s otvorenim adresiranjem i faktorom opterećenja $\alpha = n/m < 1$ kod neuspješnog pretraživanja očekivani broj ispitivanja je najviše $1/(1-\alpha)$ [1].*

Algoritam 5.40 Brisanje ključa iz tablice raspršivanja s linearnim ispitivanjem [1].

Parametri: q - ključ koji brišemo

```
1:  procedure TABLICA.OBRIŠI(q)
2:      while TRUE do
3:          T[q] ← NIL
4:          q' ← q
5:          repeat
6:              q' ← (q' + 1) mod m
7:              k' ← T[q']
8:              if k' = NIL then
9:                  return
10:             end if
11:         until g(k', q) < g(k', q')
12:         T[q] ← k'
13:         q ← q'
14:     end while
15: end procedure
```



Slika 5.35: Brisanje iz tablice raspršivanja s linearnim ispitivanjem. Hash funkcija je $h'(k) = k$ **mod** 10. Na slici (a) nalaze se elementi $74, 43, 93, 18, 82, 38, 92$ dodani po redu. Slika (b) prikazuje sadržaj tablice nakon brisanja elementa 43 s indeksa 3. Ključ 93 pomiče se unazad na indeks 3, a ključ 92 se pomiče na indeks 5 na kojem se nalazio 93. Položaji ostalih ključeva ostaju nepromijenjeni [1].

Primjerice, ako je $n = 5900$, $m = 7919$, onda je $\alpha = 0.75$, pa je broj ispitivanja najviše $1/(1 - 0.75) = 4$ (pod pretpostavkom nezavisnog ravnomjernog raspršivanja permutacije i da nema brisanja elemenata).

Posljedica ovog teorema je i da dodavanje novog elementa u tablicu raspršivanja s otvorenim adresiranjem i s faktorom opterećenja $\alpha < 1$ u prosjeku zahtijeva najviše $1/(1-\alpha)$ ispitivanja, pod pretpostavkom nezavisnog ravnomjernog raspršivanja permutacije i da nema brisanja elemenata.

Za uspješno traženje ključa vrijedi sljedeći teorem:

Teorem 7 *Za tablicu raspršivanja s otvorenim adresiranjem i faktorom opterećenja $\alpha < 1$, očekivani broj ispitivanja kod uspješnog pretraživanja je najviše*

$$\frac{1}{\alpha} ln \frac{1}{1-\alpha}$$

pod pretpostavkom nezavisnog ravnomjernog raspršivanja permutacije i da nema brisanja elemenata, te da svaki ključ u tablici ima jednaku vjerojatnost pretraživanja [1].

Popis slika

2.1 Način rada algoritma u primjeru 2.5. 19
2.2 Primjer grafa funkcije koja predstavlja vremensku složenost. 21
2.3 Odnos $f(n)$ i $g(n)$ kod O-notacije [1]. 22
2.4 Odnos $f(n)$ i $g(n)$ kod Ω-notacije [1]. 23
2.5 Odnos $f(n)$ i $g(n)$ kod Θ-notacije [1]. 25
2.6 Oblici grafova funkcija nekoliko tipičnih vremenskih složenosti. 28
2.7 Prikaz odnosa broja usporedbi i broja ulaznih vrijednosti Algoritma 2.5. . 29
2.8 Lijevo: oblik funkcije 2.1. Desno: vrijeme izvršavanja jedne implementacije Algoritma 2.5. 29

3.1 Rekurzivno računanje faktorijela. 33
3.2 Rekurzivno stablo izraza FIB-REK(4). 35
3.3 Stablo generiranja kombinacija. Svaki čvor sadrži jednu kombinaciju, iako se neke ponavljaju. 39
3.4 Stablo generiranja permutacija. U zagradi se nalazi ostatak elemenata iz prethodnika (roditelja) nakon što je izdvojen element koji se nalazi izvan zagrada. Izdvojeni elementi ulaze u permutaciju koja se dobiva od izdvojenih elemenata na putu od korijena do lista stabla. 40
3.5 Labirint s označenim vraćanjem istim putem. 41
3.6 Labirint s označenim izborima puteva. 41
3.7 Labirint kao stablo. 42
3.8 Labirint kao stablo. 42
3.9 Stablo sa Slike 3.8 prikazano kao lista. 43
3.10 Labirint. 44
3.11 Stablo labirinta na Slici 3.10. 45
3.12 Dio rekurzivnog stabla s odgovarajućim poretkom sjedenja. 46
3.13 Labirint u kojem se može ići ukrug. 47

4.1 Demonstracija Insertion sort algoritma. 55
4.2 Demonstracija Shell sort algoritma – korak 1. 57
4.3 Demonstracija Merge sort algoritma. 59
4.4 Demonstracija Quicksort algoritma. 61
4.5 Binarna gomila kao (a) stablo i (b) niz [1]. 62
4.6 Rad procedure MAKS-GOMILANJE$(a, 2)$ [1]. 64

4.7 Izgradnja gomile [1]. 65
4.8 Rad algoritma *Heapsort* [1]. 67
4.9 Rad Algoritma 4.17 [1]. 70

5.1 Primjer niza znakova koji sadrži tekst "ABC". Okomite crte označavaju segmente memorije od 16 bitova iznad kojih se nalazi njihova adresa. Adresa niza je 102 (prvo slovo) i svako slovo je veličine 2 bajta ili 16 bita. Cijeli niz zauzima 6 bajtova i proteže se od adrese 102 do 106. 76
5.2 Primjer niza koji sadrži adrese elemenata veličine 16 bitova. Radi jednostavnosti adrese su također 16-bitne. Segment adresa niza je od adrese 104 do 108, ali dodatni memorijski prostor na adresama 118, 124 i 128 upotrebljen je za sama slova. 77
5.3 Konceptualni prikaz matrice 5.1. Na slici (a) elementi su poslagani po redcima, a na slici (b) po stupcima. Na slici (c) elementi su poslagani po redcima, ali tako da niz sadrži pokazivač na svaki redak. Na slici (d) elementi su poslagani po stupcima, a niz sadrži pokazivače na svaki stupac [1]. 78
5.4 Primjer matrice 5.1 u 16-bitnoj memoriji. Segment memorije koji zauzima matrica označen je crveno. Na slici (a) matrica je smještena po redcima, a na slici (b) po stupcima. 79
5.5 Prikaz memorije koja sadrži trodimenzionalni niz s dimenzijama $2 \times 3 \times 2$ na adresi 108. Na svakoj adresi nalazi se 16-bitna vrijednost. Zelenom bojom označeni su indeksi dimenzija niza kojima se pristupa svakom tom elementu. 80
5.6 Primjer dinamičkog niza. Adresa niza A je 104, a niza B 114. Slika (a): Niz A je popunjen i nema prostora za proširenje na istom segmentu memorije (odmah iza adrese 112) jer je taj dio memorije već zauzet (nizom B u ovom primjeru). Slika (b): Da bismo dodali novi element u niz A, elementi se kopiraju u veći prostor memorije (adrese 120 do 134) u kojem ima mjesta za tri dodatna elementa, počevši od adrese 130. Nakon kopiranja nova adresa niza A je 120. 82
5.7 Binarno pretraživanje niza od 15 elemenata u kojem tražimo broj 28. . . . 85
5.8 Rad postupka BINARNO-PRETRAŽIVANJE. 86
5.9 Primjer hijerarhije memorije. 87
5.10 Primjer primjene načela prostornog lokaliteta nad nizom. 88
5.11 Primjer primjene načela prostornog lokaliteta nad nizom koji sadrži adrese elemenata. Niz je označen narančastom bojom, a elementi zelenom. . . . 88
5.12 Konceptualni prikaz jednostruko povezane liste s četiri elementa. 90
5.13 Primjer jednostruko povezane liste na razini memorije: svaki se element sastoji od vrijednosti i adrese na kojoj se nalazi idući element liste. Ako je adresa prvog elementa 129, ova lista sadrži elemente s ključevima $7 \to 3 \to 9 \to 5$. 91
5.14 Primjer dvostruko povezane liste. Svaki se element sastoji od podatka i dvije adrese: jedne za prethodni i druge za idući element liste. Poveznice za idući i prethodni element prikazane su istom bojom. Ova se lista sastoji od elemenata s ključevima u poretku $5 \to 9 \to 3 \to 7$. Element s ključem 5 zauzima adrese $133 - 135$, onaj s ključem 9 adrese $126 - 128$, itd. 94

5.15 Primjer stoga implementiranog pomoću jednostruko povezane liste. 96
5.16 Primjer operacije STOG.POP sa stogom implementiranim kao niz [23]. . . . 98
5.17 Upotreba stoga za implementaciju poziva funkcija i procedura. 100
5.18 Primjer reda implementiranog pomoću jednostruko povezane liste. 100
5.19 Primjer reda implementiranog pomoću niza[1]. 101
5.20 Primjer stabla. 103
5.21 Primjer binarnog stabla. 104
5.22 Primjer stabla za binarno pretraživanje. 104
5.23 Traženje broja 32 u stablu za binarno pretraživanje. 105
5.24 Primjer potpunog binarnog stabla. 105
5.25 Primjer skoro potpunog binarnog stabla. 105
5.26 Primjer punog binarnog stabla. 106
5.27 Primjer degeneriranog binarnog stabla. 106
5.28 Aritmetički izraz $((a+b)+(c-d))*(e/f)$ u obliku binarnog stabla. . . . 108
5.29 Brisanje elementa iz stabla za binarno pretraživanje [1]. Briše se element označen plavom bojom, a na njegovo mjesto dolazi podstablo s korijenom označenim narančastom bojom. 117
5.30 Izravno adresiranje. 120
5.31 Tablica raspršivanja kapaciteta *m*. 122
5.32 Tablica raspršivanja kod koje se kolizije rješavaju ulančavanjem. Kolizije u ovom primjeru nastaju jer je $h(k_2) = h(k_4) = h(k_5)$ i $h(k_1) = h(k_3)$. 124
5.33 Primjer umetanja ključeva pomoću dvostrukog raspršivanja. Veličina tablice je 13, $h_1(k) = k \, mod \, 13$, $h_2(k) = 1+(k \, mod \, 11)$. Prikazana je kolizija za ključ 14 koji se smješta na indeks 9 jer su indeksi 1 i 5 već zauzeti [1]. 127
5.34 Dodavanje elementa 15 u tablicu raspršivanja linearnim ispitivanjem. Hash funkcija je $h'(k) = k \, mod \, 11$ [13]. 128
5.35 Brisanje iz tablice raspršivanja s linearnim ispitivanjem. Hash funkcija je $h'(k) = k$ **mod** 10. Na slici (a) nalaze se elementi $74, 43, 93, 18, 82, 38, 92$ dodani po redu. Slika (b) prikazuje sadržaj tablice nakon brisanja elementa 43 s indeksa 3. Ključ 93 pomiče se unazad na indeks 3, a ključ 92 se pomiče na indeks 5 na kojem se nalazio 93. Položaji ostalih ključeva ostaju nepromijenjeni [1]. 129

Popis algoritama

1.1 Euklidov algoritam za pronalaženje najvećeg zajedničkog djelitelja dvaju cijelih brojeva. . . . 9
1.2 Euklidov algoritam za pronalaženje najvećeg zajedničkog djelitelja dvaju cijelih brojeva. . . . 9
1.3 Najmanji zajednički množitelj dvaju brojeva. . . . 10
1.4 Općeniti postupak, ali ne i algoritam. . . . 10
1.5 Funkcija za pronalaženje najvećeg zajedničkog djelitelja dvaju pozitivnih cijelih brojeva prema Euklidovom algoritmu. . . . 12
1.6 Primjer označavanja tipa varijable. . . . 13
1.7 Primjer korištenja atributa. . . . 13
1.8 Pseudokod za pronalazak najvećeg elementa u nizu brojeva. . . . 13
2.1 Zbroj cijelih brojeva u intervalu $[1, 10]$. . . . 15
2.2 Zbroj cijelih brojeva u intervalu [1, n]. . . . 16
2.3 Brži zbroj cijelih brojeva u intervalu [1, n]. . . . 16
2.4 Linearno pretraživanje. . . . 17
2.5 Algoritam za sortiranje niza. . . . 18
2.6 Poboljšani algoritam sortiranja 2.5. . . . 19
2.7 Provjera sortiranosti niza. . . . 20
3.1 Oblik rekurzivne funkcije. . . . 30
3.2 Oblik neizravne rekurzije. . . . 31
3.3 Iterativno izračunavanje faktorijela. . . . 31
3.4 Rekurzivno izračunavanje faktorijela. . . . 32
3.5 Rekurzivno izračunavanje *n*-tog elementa Fibonaccijevog niza. . . . 34
3.6 Iterativno izračunavanje *n*-tog elementa Fibonaccijevog niza. . . . 35
3.7 Jednostavna provjera pripadnosti elementa nizu. . . . 37
3.8 Provjera pripadnosti elementa nizu uzajamnom rekurzijom. . . . 37
3.9 Provjera pripadnosti elementa nizu izravnom rekurzijom. . . . 38
3.10 Sve kombinacije elemenata zadanog niza. . . . 39
3.11 Sve permutacije elemenata zadanog niza. . . . 40
3.12 Pronalaženje izlaza iz labirinta. . . . 44
3.13 Pronalaženje poredaka koji zadovoljavaju zadana ograničenja. . . . 46
4.1 Bubble sort. . . . 52
4.2 Bubble sort sa zastavicom. . . . 53
4.3 Selection sort. . . . 54

4.4 Insertion sort. 55
4.5 Shell sort. 56
4.6 Merge. 58
4.7 Merge sort. 58
4.8 Quick sort sa srednjim elementom kao stožernim elementom. 60
4.9 Roditelj čvora gomile. 62
4.10 Lijevo dijete čvora gomile. 63
4.11 Desno dijete čvora gomile. 63
4.12 Održavanje svojstva gomile. 64
4.13 Izgradnja gomile. 66
4.14 Heap sort. 67
4.15 Najveći element gomile. 68
4.16 Izdvajanje i vraćanje maksimuma gomile. 69
4.17 Povećavanje ključa gomile. 69
4.18 Dodavanje elementa u gomilu. 70
5.1 Operacija ZADNJI-DAN-MJESECA. 74
5.2 Adresa elementa na indeksu i. 77
5.3 Dodavanje novog elementa u niz. 80
5.4 Brisanje elementa iz niza. 81
5.5 Pretraživanje sortiranog niza. 84
5.6 Binarno pretraživanje niza. 85
5.7 Pretraživanje jednostruko povezane liste. 92
5.8 Dodavanje elementa u jednostruko povezanu listu. 92
5.9 Brisanje prvog elementa iz jednostruko povezane liste prema ključu. . . . 93
5.10 Dodavanje elementa u dvostruko povezanu listu. 95
5.11 Brisanje elementa iz dvostruko povezane liste 95
5.12 Dodavanje elementa na stog. 97
5.13 Uzimanje elementa sa stoga. 97
5.14 Provjera je li stog pun. 98
5.15 Provjera je li stog prazan. 98
5.16 Element na vrhu stoga. 98
5.17 Dodavanje elementa u red. 102
5.18 Uzimanje elementa iz reda. 102
5.19 Provjera je li red pun. 102
5.20 Provjera je li red prazan. 102
5.21 Prefiksni obilazak binarnog stabla. 108
5.22 Postfiksni obilazak binarnog stabla. 109
5.23 Infiksni obilazak binarnog stabla. 109
5.24 Prefiksni obilazak binarnog stabla iterativno. 111
5.25 Obilazak binarnog stabla u širinu. 111
5.26 Najmanji element stabla za binarno pretraživanje. 113
5.27 Najveći element stabla za binarno pretraživanje. 113
5.28 Sljedbenik čvora stabla za binarno pretraživanje. 114
5.29 Prethodnik čvora stabla za binarno pretraživanje. 115
5.30 Pretraživanje stabla za binarno pretraživanje. 116
5.31 Dodavanje elementa u stablo za binarno pretraživanje. 116

5.32 Premještanje podstabla unutar stabla za binarno pretraživanje. 118
5.33 Brisanje čvora iz stabla za binarno pretraživanje. 119
5.34 Dohvaćanje elementa tablice s izravnim adresiranjem. 121
5.35 Dodavanje elementa u tablicu s izravnim adresiranjem. 121
5.36 Brisanje elementa iz tablice s izravnim adresiranjem. 121
5.37 Primjer jednostavne funkcije raspršivanja. 122
5.38 Dodavanje ključa u tablicu raspršivanja s otvorenim adresiranjem [1]. . . 126
5.39 Traženje ključa u tablici raspršivanja s otvorenim adresiranjem [1]. 126
5.40 Brisanje ključa iz tablice raspršivanja s linearnim ispitivanjem [1]. 129

Popis tablica

2.1 Utjecaj člana x^2 na ukupni rezultat funkcije $f(x) = x^2 + 2x + 1$ kada x raste u beskonačnost. 21

4.1 Algoritmi za sortiranje (vremenska i prostorna složenost). 51

5.1 Vremenske složenosti operacija indeksiranja, dodavanja i brisanja elemenata nesortiranog niza. 89
5.2 Stanja stoga tijekom izvršavanja Algoritma 5.24. 110
5.3 Stanja reda tijekom izvršavanja Algoritma 5.25. 112

Bibliografija

[1] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009. ISBN: 978-0-26203-384-8.

[2] Steven S. Skiena. *The Algorithm Design Manual*. 3rd ed. Springer, 2020. ISBN: 978-3-030-54255-9.

[3] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison Wesley, 2006. ISBN: 0-321-29535-8.

[4] Robert Sedgewick and Kevin Wayne. *Algorithms, Part I*. Addison Wesley, 2014. ISBN: 0-13-379869-0.

[5] Marcello La Rocca. *Advanced Algorithms and Data Structures*. Manning, 2021. ISBN: 9781617295485.

[6] Michael T. Goodrich, Roberto Tamassia, and David Mount. *Data Structures and Algorithms in C++*. Wiley, 2011. ISBN: 978-0-470-38327-8.

[7] Anany Levitin. *The Design and Analysis of Algorithms*. Pearson, 2012. ISBN: 978-0-13-231681-1.

[8] Mark A. Weiss. *Data Structures and Algorithm Analysis in C++*. 4th ed. Pearson, 2014. ISBN: 978-0-13-284737-7.

[9] Mark A. Weiss. *Data Structures and Problem Solving Using C++*. Pearson, 2003. ISBN: 0321 205006.

[10] Thomas H. Cormen. *Algorithms Unlocked*. MIT Press, 2013. ISBN: 978-0-262-51880-2.

[11] Jeff Erickson. *Algorithms*. Jeff Erickson, 2019. ISBN: 978-1-792-64483-2.

[12] John V. Guttag. *Introduction to Computation and Programming Using Python*. 2nd ed. The MIT Press, 2016. ISBN: 978-0-26252-962-4.

[13] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python*. Wiley, 2013. ISBN: 978-1-11829-027-9.

[14] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. The MIT Press, 1996. ISBN: 978-0-26251-087-5.

[15] David Kopec. *Classic Computer Science Problems in Python*. Manning, 2019. ISBN: 978-1-61729-598-0.

[16] Aleksandar Stojanović. *Elementi računalnih programa*. Element, 2012. ISBN: 978-9-53197-616-9.

[17] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 2012. ISBN: 978-0-07338-309-5.

[18] Daniel P. Friedman. *Essentials of Programming Languages*. 3rd ed. The MIT Press, 2008. ISBN: 978-0-26206-279-4.

[19] Alfred Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Addison Wesley, 2006. ISBN: 978-0-32148-681-3.

[20] David M. Harris and Sarah L. Harris. *Digital Design and Computer Architecture*. 2nd ed. Morgan Kaufmann, 2013. ISBN: 978-0-12-394424-5.

[21] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[22] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5th ed. Morgan Kaufmann, 2014. ISBN: 978-0-12-407726-3.

[23] Željko Kovačević. *C++ analiza i primjena*. Školska knjiga, 2004. ISBN: 978-953-0211-54-4.

[24] Robert Sedgewick and Kevin Wayne. *Algorithms, Part II*. Addison Wesley, 2014. ISBN: 0-13-379911-5.

Ključne riječi

Ω-notacija, 23
Θ-notacija, 24

algoritam, 8
algoritam LCM, 10
algoritmi sortiranja
    s kvadratnom v. s., 51
    s log-linearnom v. s., 51
analiza algoritama, 14
apstraktni tip podatka, 72
asimptotska analiza algoritama, 20
asimptotska notacija, 21
asimptotsko ograničenje odozdo, 23
asimptotsko ograničenje odozgo, 22
asimptotsko tijesno ograničenje, 24

binarna gomila, 61
    izgradnja, 63
    maks, 62
    min, 62
    održavanje svojstva, 63
binarno stablo, 104
    balansirano, 107
    degenerirano, 107
    dubina čvora, 105
    infiksni obilazak, 109
    obilazak, 107
    postfiksni obilazak, 109
    potpuno binarno, 107
    prefiksni obilazak, 108
    prefiksni oblik, 108
    puno binarno, 107
    skoro potpuno, 62, 107
    stupanj čvora, 105
    svojstva, 105
    visina, 105
Bubble sort, 52

dvostruko povezana lista, 93
    brisanje, 95
    dodavanje, 94

elementi u poretku s ograničenjima, 45
euklidov algoritam, 9
euklidov algoritam s konkretnim vrijednostima, 9

faktorijel, 31, 32
Fibonaccijev niz, 34

Heap sort, 61
Heap sort., 66

Insertion sort, 54

jednostruko povezana lista, 90
    brisanje, 92
    dodavanje, 92
    operacije, 91
    pretraživanje, 91

kombinacije i permutacije, 38

labirint, 41
labirint kao niz, 41
linearno pretraživanje, 17

Merge sort, 58

najbolji slučaj, 17
najgori slučaj, 17

niz, 76
    binarno pretraživanje, 84
    brisanje, 79
    dinamički, 81
    dodavanje, 79
    heterogen, 77
    homogen, 76
    indeksiranje, 76
    matrica, 78
    operacije, 76
    pretraživanje, 83
    princip lokaliteta, 86
    višedimenzionalni, 78

O-notacija, 22

permutacije, 39
povezana lista, 90
prioritetni red, 63
prostorna složenost, 14
pseudokod, 12
    atributi, 13
    tip varijable, 13

Quick sort, 60

red, 100
    First-In-First-Out (LIFO), 100
    operacije, 100
red prioriteta, 68
    maks, 68
    min, 68
rekurzija, 30
rekurzivan proces, 34
rekurzivna definicija, 32
rekurzivna funkcija, 30
rekurzivno stablo, 34

Selection sort, 53
Shell sort, 56
stablo, 103
    djeca, 103
    korijen, 103
    list, 103
    obilazak u dubinu, 109
    obilazak u širinu, 110
    podstablo, 103
    potomci, 103
    predak, 103
    roditelj, 103
    ukorijenjeno, 103
stablo za binarno pretraživanje, 104
    brisanje, 115
    dodavanje, 115
    najmanji element, 113
    najveći element, 113
    operacije, 112
    prethodnik, 113
    pretraživanje, 114
    sljedbenik, 113
stog, 96
    Last-In-First-Out (LIFO), 96
    okvir aktivacije, 99
    operacije, 96
    pozivi podprograma, 99
struktura podataka, 74
    operacije, 74

tablica raspršivanja, 119
    dvostruko raspršivanje, 127
    faktor opterećenja, 123
    funkcija raspršivanja, 122
    kolizija, 122
    linearno ispitivanje, 128
    nezavisno ravnomjerno raspršivanje, 123
    otvoreno adresiranje, 125
    ulančavanje, 123
tablica s direktnim adresiranjem, 119
Tim sort, 59
tip podatka, 72
    ADT, 72
    apstraktan tip podatka, 72
    atribut, 72
    datum, 72
    definiranje, 72
    definiranje operacija, 73
    instanca, 73
    instanciranje, 73
    objekt, 73
    operacije, 73
    pristup članovima, 73
    član, 72

uzajamno rekurzivne funkcije, 30

vraćanje istim butem (backtracking), 41
vremenska složenost, 14, 15
    eksponencijalna, 27
    faktorijelna, 27
    grafički prikaz, 26
    konstantna, 15, 27
    kubna, 27
    kvadratna, 17, 27
    linearna, 15, 16, 27
    log-linearna, 27
    logaritamska, 27
    tipične funkcije, 26
vremenska složenost algoritma, 20

View publication stats