Strukture podataka i složenost algoritama u Javi

Marelja, Josipa

PRIRODOSLOVNO–MATEMATIČKI FAKULTET
SVEUČILIŠTA U SPLITU

JOSIPA MARELJA

STRUKTURE PODATAKA I SLOŽENOST ALGORITAMA U JAVI

DIPLOMSKI RAD

Split, rujan 2022.

PRIRODOSLOVNO–MATEMATIČKI FAKULTET
SVEUČILIŠTA U SPLITU

JOSIPA MARELJA

STRUKTURE PODATAKA I SLOŽENOST ALGORITAMA U JAVI

DIPLOMSKI RAD

Studentica:
Josipa Marelja

Mentor:
izv. prof. dr. sc. Jurica Perić

Split, rujan 2022.

TEMELJNA DOKUMENTACIJSKA KARTICA

Diplomski rad

Prirodoslovno-matematički fakultet

Odjel za Matematiku

STRUKTURE PODATAKA I SLOŽENOST ALGORITAMA U JAVI

Josipa Marelja

Sažetak: *Ovaj rad pokriva ključne ideje za dizajniranje algoritama. Vidimo kako upotreba određenih struktura podataka pri dizajniranju algoritma može biti efikasnija od upotrebe drugih struktura podataka. Proučili smo standardne probleme poput sortiranja, dodavanja elemenata u određenu strukturu podataka kao i brisanje elemenata. Prošli smo kroz ključne strukture podataka poput listi, stogova, redova i binarnih stabala te pogledali njihov utjecaj na složenost prilikom dizajniranja algoritma za standardne probleme. Pokazali smo implementaciju navedenih algoritama u Java programskom jeziku te pokazali primjere u kojima odabir struktura podataka ne utječe na složenost algoritma i primjere kada odabir ispravne structure podataka za dani problem smanjuje složenost algoritma za dani problem.*

Ključne riječi:

rekurzija, sortiranje, stabla, složenost

Specifikacija:

broj stranica 122, broj slika i tablica 48, broj literaturnih navoda 8, jezik izvornika: hrvatski

Mentor: *izv. prof. dr. sc. Jurica Perić*

Ocjenjivači:

prof. dr. sc. Milica Klaričić Bakula

TEMELJNA DOKUMENTACIJSKA KARTICA

doc. dr. sc. Vesna Gotovac Đogaš

Rad prihvaćen: *21.rujna 2022.g.*

BASIC DOCUMENTATION CARD

FACULTY OF SCIENCE, UNIVERSITY OF SPLIT
DEPARTMENT OF MATHEMATICS

MASTER'S THESIS

Data structures and complexity of algorithms in Java

Josipa Marelja

ABSTRACT: *This paper covers the key ideas for designing algorithms. We see how the use of certain data structures when designing an algorithm may be more efficient than the use of other data structures. We studied standard problems such as sorting, adding and deleting elements to a certain data structure. We went through key data structures such as lists, stacks, queues and binary trees and looked at their impact on complexity when designing algorithms for standard problems. We showed the implementation of the mentioned algorithms in the Java programming language and showed examples in which the selection of data structures does not affect the complexity of the algorithm and examples when the selection of the correct data structure for a given problem reduces the complexity of algorithm for the given problem.*

Key words:

recursion, sorting, trees, complexity

Specifications:

Number of pages 122, number of pages and tables 48, references 8, language: croatian

Mentor: *associate professor Jurica Perić*

Committee:

professor Milica Klaričić Bakula

assisstant professor Vesna Gotovac Đogaš

This thesis was approved by a Thesis commettee on *21.rujna 2022.g.*

Sadržaj

BASIC DOCUMENTATION CARD

BASIC DOCUMENTATION CARD

Poglavlje 1

Uvod

Prikaz informacija je osnova računalne znanosti. Primarni cilj većine programa nije izvođenje matematičkih operacija, nego spremanje i izvlačenje informacija, najčešće na najbrži mogući način. Iz ovog razloga, učenje o strukturama podataka i algoritmima za manipulaciju programa je srce računalne znanosti. Ovaj rad ima dva glavna cilja:

- Prezentiranje najčešće korištenih struktura podataka
- Procjena koristi i složenosti pojedine strukture podataka kroz upotrebu u algoritmima

Primjeri unutar računalne znanosti:

1. Rad s mrežama: Uzimimo primjer slanja poruke s jedne virtualne mašine na drugu virtualnu mašinu. Trebamo spremiti mrežne puteve kojima prolazi poruka od jedne mašine na drugu. Pitanje na koje odgovor može dati pravilan odabir strukture podataka te odabir ispravnog algoritma je: Kako možemo smanjiti vremensku složenost slanja poruke korištenjem skupa putova kojima prolazi poruka?

2. Prikupljanje podataka: Kako pretraživač poput Googlea sprema podatke prilikom korisnikovog pretraživanja? Kako odabrati najbolji izbor stranica koje možemo ponuditi korisniku pri njegovoj pretrazi? Ovo su sve pitanja koje rješava odabir ispravne strukture podataka i algoritma.

Jedino kroz ovakva mjerenja korisnosti i složenosti algoritma možemo se odlučiti za pravu strukturu podataka za određeni problem. Koncentrirat ćemo se na nekoliko osnovnih zadataka kao što su sortiranje, pretraga i spremanje podataka, koji obuhvaćaju velik dio računalne znanosti. Upoznat ćemo se s različitim strukturama podataka kao što su liste, redovi, stogovi i stabla, zatim ćemo uspoređivati njihovu upotrebljivost u različitim algoritmima za pretragu i sortiranje.

Poglavlje 2

Strukture podataka i algoritmi

2.1 Strukture podataka

Za većinu problema, formuliranje korisnih algoritama ovisi o mogućnosti dobrog organiziranja podataka za dani problem. Izraz "strukture podataka" se koristi za opis specifičnog načina organiziranja podataka. Proći ćemo kroz razne strukture podataka i vidjeti kako odabir utječe na složenost određenih algoritama. Kroz ovaj rad želim se posvetiti strukturama podataka, zanemarujući količinu dostupne memorije u računalu. Više ću se posvetiti gledanju struktura podataka kao matematičkih modela pojedine klase strukture podataka ili tipova podataka koji su im zajednički. Način pristupa tipovima podataka kao matematičkim modelima koji su definirani semantikom, mogućim vrijednostima i mogućim operacijama koje se mogu izvoditi na njima te ponašanjem određene operacije, naziva se apstraktni tip podataka (eng. "abstract data type"). Specificirat ćemo primitivne tipove podataka od kojih su izgrađene strukture podataka, kako izvući određene tipove podataka iz njih i osnovne provjere za kontrolu u obradi algoritma. Ideja te implementacije je skrivanje podataka od korisnika i "zaštita" od vanjskih pristupa,

što zovemo enkapsulacija. Veći nivo apstrakcije su "design patterns" koji su opisani kao dizajn algoritma. Oni generaliziraju osnovne koncepte dizajna koji se ponavljaju u raznim problemima. Omogućuju osnovnu strukturu algoritma i pružaju dokazane algoritamske strukture koji se mogu direktno primijeniti na nove probleme. Kad god radimo sa određenom strukturom podataka moramo uzeti u obzir sljedeće stvari:

1. Modeliranje: način oblikovanja stvarnih podataka u matematičke modele koje možemo programski implementirati.

2. Operacije: sagledati potrebne operacije za spremanje, obradu podataka i formalizirati načine implementacija operacija unutar algoritma.

3. Reprezentacija: utjecaj reprezentacije stvarnih podataka kao matematičkih modela na memoriju računala.

4. Algoritmi: precizirati algoritme koji mogu odraditi definirane operacije.

Primijetimo da su prve dvije stavke u osnovi matematički problemi, dok su zadnje dvije problemi same implementacije algoritma u obliku računalnog programa. Pogledajmo npr. uzimanje redova kao strukture podataka za određeni algoritam. Redovi su niz objekata (nedefiniranog tipa). Objekte možemo ubaciti u red operacijom "push". Uzmimo npr. redove koji se mogu implementirati na razne načine, poput niza ili vezane liste. Koja implementacija daje brži algoritam? Koja implementacija zauzima manje memorije? Koja implementacija je fleksibilna na promjene? U primjeru reda, odgovori na pitanja memorijskog zauzeća i fleksibilnosti na promjene operacije "push" su jednostavani. No, s kompleksnijim operacijama, povećava se sama složenost algoritama te stoga odgovor postaje složeniji. Kroz iduća poglavlja promatrat ćemo razne strukture podataka, pogledati njihovu implementaciju unutar programskog jezika Java i dati alate potrebne za dizajniranje

i implementaciju struktura podataka kako bi riješili problem reprezentiran algoritmom.

2.1.1 Potreba za strukturama podataka

Smatrate da se s većom moći računala, programska složenost postaje manje važna? Ipak, procesorska brzina i memorijska veličina se konstantno popravljaju. Hoće li se problem brzine izvođenja složenijih algoritama riješiti za nekoliko godina novim napretkom procesora i povećanjem količine memorije? Proizvodnjom računala s većom procesorskom moći i memorijskom veličinom, uspjeli smo postići izvođenje pojedinih složenijih računskih radnji kao i složenijih algoritma koji prije nisu bili mogući. No, složeniji algoritmi zahtijevaju veći broj računskih radnji, što potrebu za efikasnijim programima čini još većom. Do efikasnijih programa dolazimo smanjivanjem složenosti. Za rješenje problema kažemo da je efikasno ako rješava problem za dana ograničenja resursa. Primjer ograničenja resursa je dostupna memorija za spremanje podataka. Trošak rješenja je količina resursa koje određeno rješenje koristi. Najčešće pod troškom rješenja problema uzima se vrijeme ili složenost algoritma. Odabiru strukture podataka za rješavanje problema možemo pristupiti na sljedeći način:

- Analiziranjem problema za određivanje osnovnih operacija koje moraju biti podržane,
- Ograničiti resurse za svaku operaciju,
- Odabrati strukturu podataka koja je najbolja za dane uvjete.

2.1.2 Mane i prednosti

U praksi je jako teško reći da je jedna struktura podataka bolja od druge strukture podataka u svim situacijama, tj. za svaki problem. Ako je jedna struktura podataka bolja od druge strukture podataka u svim slučajevima onda se druge strukture podataka ne bi koristile i već bi bile davno zaboravljene. Struktura podataka zahtijeva određenu količinu memorije za spremanje svake njegove instance, određenu količinu vremena za obavljanje osnovne operacije. Obzirom da implementacija programa mora završiti u određenom vremenu (ne može biti beskonačan), i kako je količina dostupne memorije na svakom računalu ograničena, svaki problem implementiran u računalnom obliku je ograničen dostupnom memorijom i vremenom. Stoga je za svaki problem potrebna detaljna analiza kako bi se odredila najbolja struktura podataka za nju i prethodno navedena ograničenja.

Napomena 2.1 *Instanca neke klase je objekt, poznatijim pod nazivom objekt klase ili instanca neke klase. Obzirom na navedeno, instanca se može smatrati konstrukcijom ili izvedbom same klase. Kad god objekti iste klase poprimaju drugačiji oblik (bilo u smislu tipa vrijednosti ili metoda definiranim u njima) nazivaju se različitim instancama iste klase.*

Primjer 2.2 *Uzmimo primjer klase pas, koja ima varijable boja, ime, vrsta, prehrana, itd. Instanca ove klase bio bi pas zvani Rubi koji je smeđe boje i hrani se suhom hranom.*

2.2 Problemi, algoritmi i programi

Programeri se često susreću s problemima, algoritmima i računalnim programima. Problem je zadatak koji se treba riješiti. Najbolje ga možemo opisati sa zadanim skupom ulaznih podataka i odgovarajućim skupom izlaznih podataka. Definicija problema ne bi smjela biti ograničena s načinom rješavanja problema. Problemima možemo pristupiti u matematičkom terminu. Rješenje problema možemo definirati kao funkciju, koja za određen skup ulaznih podataka (kojih može biti nula, jedan ili više), tj. domenu, daje skup izlaznih podataka.

Primjer 2.3 *Neka je dan algoritam koji od danog skupa podataka prirodnih brojeva, vraća broj prostih brojeva. Tada bi domena bila neki podskup prirodnih brojeva, npr. {3,4,5,6,7,8,9,10}, a izlazni skup podataka bio bi prirodan broj n. Algoritam definiran za ovaj problem bi izgledao ovako:*

```
broj_prostih_brojeva_u_nizu=0
neka je dan ulazni niz brojeva
za svaki broj u nizu
    neka je brojač br=0;
        za svaku poziciju i manju od duljine niza,
            ako je ostatak pri djeljenju s i jednak nuli
                brojač br povećaj za jedan,
            inače povećaj vrijednost i za jedan
    ako je br==0 onda je broj broj_prostih_brojeva_u_nizu++
```

Ovaj algoritam će nam za dani skup podataka {3,4,5,6,7,8,9,10} vratiti izlaz 3, što je prirodni broj.

Algoritam laički možemo opisati kao metodu, postupak ili niz pravila za rješavanje neke klase problema. Ako rješenje problema gledamo kao funk-

ciju, onda algoritam možemo smatrati implementacijom za funkciju koja pretvara ulazni skup podataka u odgovarajući izlazni skup podataka. Jedan problem se može riješiti s više algoritama. Prednost poznavanja različitih algoritama za rješavanje određenog problema daje nam mogućnost usporedbe složenosti, vremena izvođenja i korisnosti, kako bi odabrali najbolji algoritam za rješavanje određenog problema. Naprimjer, jedan algoritam može biti dobar za rješavanje problema sortiranja malog broja podataka, dok drugi algoritam može biti bolji za veći broj podataka. Postupak možemo smatrati algoritmom ako zadovoljava određena svojstva:

- Mora dati očekivani skup podataka, za dani ulazni skup podataka. Objasnit ćemo što smatramo očekivanim skupom podataka na sljedećem primjeru.

- **Primjer 2.4** *Neka imamo algoritam koji za ulazni skup podataka vraća listu parnih brojeva. Tada, ako je ulazni skup podataka niz brojeva* $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$*, očekivani skup podataka će biti lista* $\{2, 4, 6, 8, 10\}$.

  Napomena 2.5 *Svaki algoritam (ujedno i implementacija algoritma u obliku računalnog programa) implementira neku funkciju, zato što svaki algoritam mapira svaki ulazni podatak za određeni izlazni podatak. Ovim rečeno, tražimo da određeni algoritam implementira određenu funkciju.*

  Primjer 2.6 *Algoritam koji svakom prirodnom broju pridružuje broj relativno prostih s n koji su manji od n, zapravo implementira Eulerovu funkciju.*

- Algoritam treba imati određeni niz koraka. Koraci trebaju biti razumljivi i računalu i osobi koja čita algoritam.

- Algoritam ne smije biti dvosmislen, tj. u svakom koraku mora biti točno određeno koji korak će biti sljedeći.
- Broj koraka u algoritmu treba biti konačan.

Računalne programe često smatramo instancom ili konkretnom implementacijom algoritma u nekom programskom jeziku. Nastavno na ovo postoji više instanci jednog algoritma, jer mogu biti implementirani u različitim programskim jezicima. No, koja je glavna razlika između algoritma i programa? Algoritam mora pružiti dovoljno detalja kako bi implementacija u program bila moguća. Zahtjev da algoritam završi u konačnoj jedinci vremena (algoritam se ne smije vrtjeti beskonačno) zapravo objašnjava da ne zadovoljavaju svi računalni programi tehničku definiciju algoritma. Operativni sustav je primjer takvog programa. Iako, svaki različit zadatak operativnog sustava može se smatrati kao individualni problem, koji se može riješiti s različitim algoritmima.

Ukratko,

- Problem je funkcija koja za jedan skup podataka daje odgovarajući skup podataka.
- Algoritam je funkcija koja implementira niz koraka potrebnih za rješavanje problema.
- Program je instanca algoritma u određenom programskom jeziku.

2.3 Uvod u Javu

U ovom poglavlju opisat ćemo dijelove programskog jezika Java. Programski jezik Java ima dosta korisnih značajki koje su idealne za implementaciju struktura podataka. Java sadrži veliki skup biblioteka i klasa koje možemo koristiti. Naravno, nije potrebno znati implementaciju svake strukture podataka nego pogledati biblioteku vezanu za strukturu podataka koju koristimo. Navedimo nekoliko primjera biblioteka koje se koriste:

- java.lang: sadrži nizove znakova (eng. "string")
- java.io: sadrži podršku za upis i ispis podataka
- java.util: sadrži klase struktura podataka

Navedimo nekoliko tipova koji se koriste prilikom programiranja:

- Integer: engleski naziv za cijele brojeve
- Float: engleski naziv za realne brojeve
- Char: engleski naziv za znak
- Bool: engleski naziv za logičke vrijednosti istina ili laž

Ovisno da li radimo da brojevima, nizovima ili logičkim vrijednostima odabrat ćemo navedene tipove. Varijable definiramo navodeći tip podataka koji koristimo te vrijednost koju ta varijabla poprima.

Primjer 2.7

```
int i = 10*4
float pi = 3.14f
char c = 'a'
boolean b =  i < pi
```

Sučelja programskog jezika Java za određene strukture podataka navodit ćemo prilikom predstavljanja svake od struktura podataka kroz iduća poglavlja.

2.3.1 Nasljeđivanje i podklase

Java podržava nasljeđivanje od roditelj klase pomoću izraza 'extend'. Pogledajmo primjer nasljeđivanja:

Primjer 2.8

```
class Osoba { ... }
class Student extends Osoba { ... }
```

Student nasljeđuje strukturu od klase Osoba i može dodavati nove instance varijable i nove metode unutar svoje klase. Ako se metoda naziva isto kao i nasljeđenja metoda sa jednakim brojem parametara i drugačijom svrhom, smatra se da poništava (eng. "override") prethodnu metodu definiranu u roditeljskoj klasi.

Metode se ponašaju kao funkcije. Metode mogu biti privatne, javne ili zaštićene (eng. "public", "protected" ili "private") ovisno o odabiru prilikom definiranja. Klase također mogu biti tipa "final", "abstract" ili "public", gdje se "public" klasa definira kao javna, jer se mogu dodatno definirati nove metode unutar nje. Klasa tipa "final" ne dozvoljava podklase, dok klasa tipa "abstract" ima apstraktne metode definirane unutar sebe.

2.3.2 Sučelja struktura podataka

Kako su definirane strukture podataka unutar Javinog sučelja? Svaka klasa strukture podataka nasljeđuje generičnu klasu "Objects". Obzirom da je

klasa Objects roditelj klasa svih klasa struktura podataka, možemo spremati objekte iz bilo koje klase unutar naše strukture podataka. No, da bi sama klasa strukture podataka imala smisla, moramo u nju stavljati ono čime smo ju definirali. Pokažimo navedeno na primjeru.

Primjer 2.9 *Što ako želimo spremati cijele brojeve u klasu ArrayVector definiranu ispod? Koristit ćemo tip Integer da bi svaki objekt pridružen pretvorio u cijeli broj.*

```
Arrayvector A = new Arrayvector();
//dodati 999 na nultoj poziciji
A.insertAtRank(0,Integer(999));
// pristupiti vrijednosti na nultoj poziciji
Integer x= (Integer) A.elementAtRank(0);
```

Primjetimo da ArrayVector sadrži Objects. Stoga, kada pristupamo nečemu iz ove strukture podataka moramo mu prvo pridružiti tip podatka.

Poglavlje 3

Analiza algoritma

Koliko vremena je potrebno za izvršavanje programa? Da li je određeni program spor zato što je loše implementiran? Pitanja poput ovih tjeraju nas da uzmemo u obzir složenost algoritma, efikasnost i razmislimo da li je pristup rješavanju bio ispravan. Postoji li možda lakši, manje složen i samim time efikasniji način rješavanja istog problema? Ovo nas vodi pristupu poznatoj kao asimptotska analiza algoritama ili kraće, asimptotska analiza. Asimptotska analiza se bavi procjenom i usporedbom dvaju algoritama pri rješavanju istog problema.

Jasno je da algoritme ne možemo promatrati zasebno, bez struktura podataka. Stoga ćemo prilikom promatranja algoritma uzeti u obzir različite strukture podataka za spremanje i obradu. No, uzmimo u obzir da, iako je određena struktura podataka dobra za algoritam, nije ujedno najbolja za sve algoritme tog problema. Npr. za problem sortiranja podataka, različiti algoritmi imaju različite strukture podataka koje su za njih najbolje.

3.1 Uvod u asimptotsku analizu

Kako usporediti dva algoritma koji rješavaju isti problem u smislu efikasnosti? Jedan od načina pristupa je implementiranje algoritma u računalni program, kojeg ćemo pokretati s različitim brojem ulaznih podataka te mjeriti vrijeme potrebno za svaki od njih. No, ovaj pristup nije dobar iz tri razloga.

1. Pišemo program za dva algoritma, kad ćemo uistinu trebati samo jedan od njih, stoga vrijeme i trud potrebni za implementaciju algoritma, kojeg nećemo koristiti, u računalni program su uzalud bačeni.

2. Kad empirijski uspoređujemo algoritme, uvijek postoji mogućnost da je jedan program napisan bolje od drugog, stoga analiza neće donijeti ispravnu odluku boljeg algoritma.

3. Odabir testnih ulaznih podataka može ići u korist određenom algoritmu, npr. ako je sortiran niz podataka jedan algoritam će ih sortirati brže nego drugi, dok kod nesortiranog niza će drugi biti bolji.

Navedeni razlozi se mogu izbjeći korištenjem asimptotske analize, koja mjeri složenost algoritma i njegove programske implementacije kroz povećavanje broja ulaznih podataka. Asimptotska analiza nam neće direktno dati vrijeme potrebno za izvođenje algoritma, u smislu "da li je ovaj algoritam brži od drugog?", nego će nam dati odgovor na pitanje, može li se i isplati li se implementirati algoritam kao računalni program?

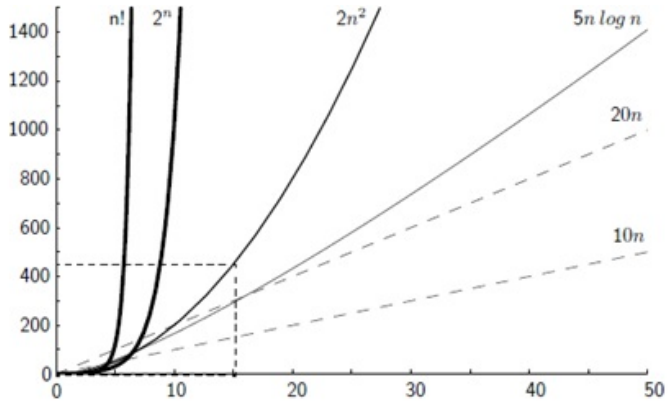
n	$loglogn$	$logn$	n	$nlogn$	n^2	n^3
16	2	4	16	32	256	4096
256	3	256	2048	65536	16777216	$1,16x10^7$

Tablica nam prikazuje za ulazni podatak veličine n kolika je vrijednost ovisno o složenosti algoritma. Npr. ako je složenost algoritma n^2 za n ulaznih podataka vrijednost složenosti za 16 ulaznih podataka, bit će znatno veća od algoritma čija je složenost $logn$. No, korisnike programa često zanima vrijeme potrebno za izvođenje određenog programa, kao i količina memorije potrebna za izvođenje i spremanje određene strukture podataka. Dosta faktora utječe na vrijeme potrebno za izvođenje programa osim same implementacije, poput centralne jedinice za obradu podataka samog računala, programskog jezika itd.

Ako imamo ograničene resurse vremena i memorije, onda trebamo uzeti sve navedeno u obzir. No, niti jedan od navedenih faktora ne daje nam odgovor koji od dva algoritma je bolji. Kako bi usporedba programa koji implementiraju dva algoritma ili različite strukture podataka bila ispravna, treba biti izvedena na istom računalu, te pisana u istom programskom jeziku. No, tu nam ostaje još uvijek problem pisanja samog programa. Da li je jedan program napisan bolje od drugoga? Stoga nam ne preostaje ništa drugo nego usporediti na neki drugi način algoritme.

Prvi način uspoređivanja dvaju algoritama može biti količina obavljenih osnovnih operacija za definiranu količinu ulaznih podataka. Pojam osnovne operacije može biti matematičko uspoređivanje brojeva, a može biti i premještanje brojeva unutar niza. Zbrajanje sadržaja niza od n brojeva nije operacija o kojoj govorimo, jer ona ovisi o broju ulaznih podataka i samim time bi "trošak algoritma" ovisio u ulaznom podatku, što ne želimo. Uzmimo u obzir jednostavan kod, često korišten u programiranju, poput:

```
zbroj=0;
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
```



Slika 3.1: Ilustracija stope rasta za različite funkcije .

```
zbroj++
```

Koje je vrijeme izvođenja ovog dijela programa? Očito je da će vrijeme biti veće s odabirom većeg n. Osnovna operacija je povećanje operacijske varijable zbroj. Pretpostavimo da povećanje vrijednosti za jedan (tj. metoda vrijednost++) ima složenost c_2. Ignorirat ćemo vrijeme potrebno za postavljanje varijable zbroj. Ukupno vrijeme potrebno za povećanje varijable je reda veličine $c_2 n^2$.

Os x predstavlja broj ulaznih podataka, dok os y prezentira vrijeme, memoriju ili drugu vrstu potrošnje.

3.2 Najbolji, najgori i prosječni slučajevi

Pokažimo na primjeru problema računanja faktorijela od n (ulazni skup podataka je $\{1, 2, \ldots, n\}$), količinu vremena potrebnu za izvršavanje. Za ovaj problem, imamo samo jedan ulazni podatak, a to je veličina niza n. Neovisno o poretku brojeva od 1 do n u nizu (poredak može biti [1,2,. . . ,n] ili [2,n,n-1,1,3,4,5,. . . ,n-2]), prilikom linearne pretrage ili računanja faktorijela vrijeme potrebno za izvršavanje će biti isto. Kod navedenog problema, pore-

dak elemenata u nizu nije bitan, stoga je rezultat isti i vrijeme potrebno za izvršavanje će biti jednako.

U drugim slučajevima, poput traženja broja K unutar niza, uz pretpostavku da se broj K nalazi samo jedan put u nizu, poredak elemenata će biti itekako bitan. Pogledajmo linearni algoritam koji pretražuje jedan po jedan element i kad pronađe broj zaustavlja se. Ako je broj K na prvom mjestu, algoritam će ga odmah naći. No, u slučaju da se broj K nalazi na posljednjem mjestu, istom algoritmu će biti potrebno znatno više vremena, jer mora pregledati prije njega još $n - 1$ vrijednosti.

Navedeni slučaj, kada se traženi broj nalazi na prvom mjestu, je najbolji slučaj za algoritam linearne pretrage, dok slučaj kad se dani broj nalazi na zadnjem mjestu najgori. Kada bi isti algoritam, za istu veličinu niza, implementirali u računalni program i izvršili pretragu za danom vrijednosti K, koja bi se svaki put nalazila na različitoj poziciji ili kada bi svaki put tražili različitu vrijednost K, prosječno vrijeme pretrage bi bio prosječni slučaj i znamo da bi bio približno jednak $\frac{n}{2}$.

Kad analiziramo algoritam, bi li trebali gledati najbolji, najgori ili prosječni slučaj algoritma?

Najčešće nismo zainteresirani za najbolji mogući slučaj algoritma, jer je vjerojatnost da se dogodi jako mala. Najbolji slučaj ćemo uzeti u obzir samo onda kada znamo da je velika vjerojatnost da će se dogoditi, na primjer u slučajevima kada znamo da će najmanja vrijednost niza biti na prvom mjestu, jer je niz sortiran.

Prednost uzimanja najgoreg slučaja u obzir je što znamo da se algoritam mora izvršiti brzinom najviše toliko. Aplikacija praćenja slijetanja aviona na pistu, koje koriste zrakoplovne kompanije su jako dobar primjer, u smislu kad svi avioni trebaju sletjeti u isto vrijeme. U ovom primjeru nam je bitno

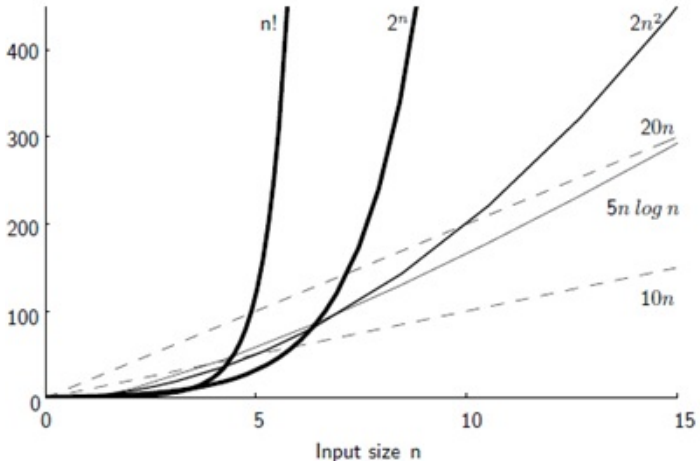
promatrati najgori slučaj, tj. kad svi avioni slijetaju na pistu u isto vrijeme, neće nas pretjerano zanimati slučaj kad niti jedan avion ne sleti na pistu.

U većini slučajeva, korisnike ipak zanima prosječan slučaj za određeni algoritam. No, za izračunavanje prosječnog slučaja, potrebno je uzeti u obzir strukturu podataka, raspored u strukturama podataka, jesu li svi slučajevi jednako vjerojatni,... Stoga nekad nije moguće dati prosječan slučaj za određeni algoritam.

Ukratko, za aplikacije koje se koriste u stvarnom životu, preferira se izračun za najgori slučaj kako bi se znala donja granica za memoriju ili vrijeme potrebno za izvođenje algoritma. Nakon najgoreg, preferira se prosječni slučaj, ako znamo strukturu podataka koja se koristi i raspodjelu podataka unutar nje.

3.3 Asimptotska analiza

Neka su dane funkcije $f(n) = 10n$ i $g(n) = 2n^2$. Vrijednost funkcije g biti će veća od vrijednost funkcije g čim argument (koji predstavlja broj ulaznih podataka) prijeđe vrijednost 5. Ako povećamo multiplikativnu konstantu funkcije f s 10 na 20, potrošnja koju prezentira vrijednost funkcije g će prijeći potrošnju koju prezentira vrijednost funkcije f, kada broj ulaznih podataka ima vrijednost veću od 10. Prilikom kupovanja novog računala ili izvođenja programa koji implementira algoritam na računalu s bržim kompajlerom, potrošnja će za veći broj ulaznih podataka za funkciju f biti znatno manja, dok za funkciju g neznatno manja. Iz ovih razloga se često ignorira konstanta, kada želimo procijeniti stopu rasta za određeni faktor potrošnje (bilo vrijeme ili memorija), ovim se bavi asimptotska analiza. Točnije, asimptotska analiza bavi se proučavanjem stope porasta potrošnje, kada broj ulazni podataka



postane jako velik ili dosegne svoju gornju granicu. No, koliko je god korisno ignorirati konstantu kada se radi o velikom broju ulaznih podataka, to ne možemo primijeniti pri malom broju ulaznih podataka. Pri zanemarivanju konstante u malom broju ulaznih podataka, primjerice 5 ulaznih podataka, analiza algoritma bi mogla prikazati veliku grešku pri procjeni. Stoga asimptotsku analizu koristimo samo u slučajevima kada nam je potrebno shvaćanje ponašanja algoritma pri velikom broju podataka kako bi bili svjesni gornje granice stope potrošnje.

3.3.1 Gornja granica, donja granica, Theta notacija

Dosta termina i simbola koristi se za opisivanje vremena potrebnog za izvršavanje operacija algoritma. Jedan od termina je i gornja granica. Ona opisuje gornju ili najvišu stopu rasta koju jedan algoritam može imati. Radi lakše komunikacije, često se za gornji termin funkcije f za ulazne podatke n, uzima notacija $O(f(n))$, čitamo „veliko O notacija“. Često nas ne zanima točna vrijednost funkcije $f(n)$ koja opisuje vrijeme složenosti nekog algoritma za n ulaznih podataka, nego samo klasa složenosti. Pristup promatranja klase složenosti i zanemarivanja konstanti, nam daje uvid u složenost algoritma

pri velikom broju ulaznih podataka.

Definicija 3.1 *Neka su $f, g\colon D \to \mathbb{R}$ realne funkcije na odozgo neograničenom podskupu $D \subseteq \mathbb{R}$. Funkcija f ne raste brže od g, u oznaci $f(x) = O(g(x))$ ako postoje $c \in \mathbb{R}^+$ i $x_0 \in D$ tako da je $|f(x)| \leq c * |g(x)|, \forall x \geq x_0 \in D$.*

Objasnimo malo navedenu definiciju.

1. Mi ne trebamo znati kada točno vrijednost funkcije f postanje manja od c*g. Nas zanima postojanje n_0 takvog da za sve vrijednosti n veće od njega, vrijednosti funkcije f su manje od vrijednosti c*g.

2. Želimo uzeti u obzir efikasnost algoritma neovisno o brzini računala na kojem će se izvoditi. Iz ovog razloga vrijednost funkcije g se množi s konstantom c. Sama ideja je da ne preciziramo točno vrijeme izvođenje pojedine funkcije unutar koraka, nego gledamo vremensku složenost ukupnog algoritma.

Definicija 3.2 *Neka su $f, g\colon D \to \mathbb{R}$ realne funkcije na odozgo neograničenom podskupu $D \subseteq \mathbb{R}$. Funkcija f je manjeg reda veličine od g ili f raste sporije od g, u oznaci $f(x) = o(g(x))(x \to \infty)$ ako postoji $lim_{x\to\infty}\frac{f(x)}{g(x)}$ i jednak je 0.*

Napomena 3.3 *Funkcije složenosti su uvijek nenegativne, pa apsolutne vrijednosti u definiciji možemo izostaviti.*

Ako pretpostavimo da su svi koraci algoritma traže jednaku vremensku potrošnju, onda se klasa kompleksnosti algoritma odlučuje na temelju petlji i koliko često se koraci unutar petlje izvode. Ovim pristupom dobivamo mjeru koja je primjenjiva za svaki ulazni podatak, samim time možemo procijeniti mjeru za jako velik broj ulaznih podataka.

Konstanta n_0 je vrijednosti ulaznog broja podataka za koju vrijedi gornja granica. Često je n_0 mala vrijednost (možda i 1), ali ne nužno. Također

moramo biti u mogućnosti izabrati proizvoljnu konstantu c kako bi za svaki broj ulaznih podataka, algoritam trebao završiti za $cf(n)$. "Veliko O" notacija nam opisuje gornju granicu, tj. daje nam tvrdnju o najvećem broju resursa potrebnih za izvođenje algoritma neke klase za broj ulaznih podataka n. Slična notacija koristi se za najmanju količinu resursa potrebnih za izvođenje klase algoritma za broj ulaznih podataka n. Donja granica algoritma se opisuje sa notacijom Ω , čitamo „Veliko omega“.

Definicija 3.4 *Neka su $f, g\colon D \to \mathbb{R}$ realne funkcije na odozgo neograničenom podskupu $D \subseteq \mathbb{R}$. Funkcija f nije manjeg reda veličine od g ili f raste barem jednako brzo kao i $f(x) = \Omega(g(x))$ $(x \to \infty)$ ako nije $f(x) = o(g(x))$.*

Primjer 3.5 $x^2 = o(x^5)$ $lim_{x\to\infty}\frac{x^2}{x^5} = lim_{x\to\infty}\frac{1}{x^3} = 0$.

Definicije „velikog O" i Ω daju nam način za opisati gornju (ako možemo naći jednadžbu maksimalne potrošnje za pojedinu klasu pri ulaznom broju podataka n) i donju granicu za algoritam (ako možemo naći jednadžbu minimalne potrošnje za pojedinu klasu pri ulaznom broju podataka n). Kada su gornja i donja granica jednake zanemarujući konstantni faktor, to zapisujemo u obliku Θ notacije, čitamo „veliko Theta notacije“.

Definicija 3.6 *f je istog reda veličine kao i g ili f raste istom brzinom kao i g, u oznaci $f(x) = \Theta(g(x))$ $(x \to \infty)$, ako postoje $c_1, c_2 \in \mathbb{R}^+$ i $x_0 \in D$ tako da je $c_1 * |g(x)| \leq |f(x)| \leq c_2 + |g(x)|$, $\forall x \geq x_0$, $x \in D$, gdje je D odozgo neograničen podskup od $\mathbb{R}$.*

Za primjer algoritma linearne provjere, stopa rasta pretraga je unutar skupova O(n) i Ω(n) u prosjeku, pa kažemo da je složenosti Θ(n) u prosjeku. Pri promatranju algebarskih jednadžba koje opisuju vrijeme potrebno za izvedbu algoritma, gornja i donja granica se uvijek poklapaju. To je zato što

imamo, pri razumnom promatranju, savršenu analizu za algoritam, prikazanu pomoću prethodno navedene jednadžbe. Za dosta algoritama jako je lako pronaći jednadžbu koja opisuje ponašanje kroz određeni period. Iako se dosta programera koristi izrazom „veliko O“ ili O pri opisu složenosti algoritma, ipak ponašanje složenosti algoritma bolje se izražava Θ notacijom, kad god imamo dovoljno znanja i informacija o algoritmu koji se odvija.

Nakon određivanja jednadžbe koja opisuje vrijeme izvođenja algoritma, obično je lako odrediti Θ, Ω i O izraz za danu jednadžbu. Ne trebamo se koristiti definicijama za odrediti izraz, nego se možemo koristiti sljedećim pravilima ili svojstvima.

1. Ako je $f(n)$ u skupu O(g(n)) i g(n) u skupu O(h(n)), onda je $f(n)$ u skupu O(h(n)). (Svojstvo tranzitivnosti)

2. Ako je $f(n)$ u skupu O(kg(n)), za bilo koju konstantu $k > 0$, onda je $f(n)$ u skupu O(g(n)).

3. Ako je$f_1(n)$ u skupu $O(g_1(n))$ i $f_2(n)$ u skupu $O(g_2(n))$, onda je $f_1(n)+f_2(n)$ u skupu $O(max\{g_1(n), g_2(n)\})$.

4. Ako je $f_1(n)$ u skupu $O(g_1(n))$ i $f_2(n)$ u skupu $O(g_2(n))$, onda je $f_1(n)f_2(n)$ u skupu $O(g_1(n)g_2(n))$.

Prvo pravilo nam sugerira ako je neka funkcija g(n) gornja granica za funkciju $f(n)$, onda je gornja granica za g(n) također gornja granica za $f(n)$. Slično svojstvo ima i notacija Ω, ako je donja granica za funkciju f, onda je bilo koja donja granica funkcije g(n) također donja granica funkcije f. Isto vrijedi i za notaciju Θ.

Važnost drugog pravila je sama mogućnost ignoriranja multiplikativnih konstanti u jednadžbama prilikom korištenja O notacije. Isto vrijedi i za Ω i Θ.

Treće pravilo nam govori ako imamo dva dijela algoritma koja se izvode istovremeno, u obzir trebamo uzeti samo dio s najvećom potrošnjom (vremena ili drugačijih resursa), tj. onaj s većom složenošću. Četvrto svojstvo se koristi određen broj puta za određivanje složenosti petlji, tj. ako se neka radnja ponavlja određen puta i svako ponavljanje ima jednaku potrošnju, tj. u našem slučaju složenost. Time dobivamo ukupnu složenost jednog dijela algoritma tako da pomnožimo dio koji se ponavlja s brojem ponavljanja. Isto svojstvo vrijedi i za Ω te Θ.

Uzevši u obzir prva tri svojstva, možemo ignorirati sve konstante i dijelove manje složenosti koji se izvode istovremeno kako bi dobili asimptotsku stopu rasta za bilo koju funkciju. Opasnost od greške radi ignoriranja konstanti opisali smo prethodno, no malo ćemo sad reći nešto o ignoriranju manje složenih radnji kada se obavlja neka složenija radnja.

Ignoriranje manje složenih radnji u korist složenije je uobičajeno kod asimptotske analize. Složenije radnje dobivaju na većoj važnosti kako raste broj ulaznih podataka te za jako malu razliku u ulaznim podacima, postaje značajno veća nego manje složena radnja.

Primjer 3.7 *Ako je* $T(n) = 3n^4 + 5n^2$*, onda je T(n) u skupu* $O(n^4)$*. Izraz* n^2 *pridonosi jako malo složenosti za jako velike brojeve. Za dane funkcije f, g i broj ulaznih podataka n, pri izražavanju stope rasta kao algebarskih jednadžbi, možda poželimo odlučiti koja ima brži rast. Pri tome se koristimo limesom količnika, kada n raste prema beskonačnosti, tj.* $\lim_{n\to\infty} \frac{f(n)}{g(n)}$*. Kada je vrijednost limesa beskonačno, za n koji ide prema beskonačno, onda je* $f(n)$ *u skupu* $\Omega(g(n))$*, jer vrijednost* $f(n)$ *raste brže. Kada je vrijednost limesa 0, onda je* $f(n)$ *u skupu O(g(n)), jer vrijednost g(n) raste brže. Ako je vrijednost limesa konstanta različita od nule, onda je* $f(n)=\Theta(g(n))$*.*

Primjer 3.8 *Ako je* $f(n) = 2nlogn$ *i* $g(n) = n^2$*, da li* $f(n)$ *u skupu* $O(g(n))$*,*

$\Omega(g(n))$ *ili* $\Theta(g(n))$*?*

Kako je $\frac{n^2}{2nlogn}=\frac{n}{2logn}$ *, onda je* $\lim_{n\to\infty}\frac{n^2}{2nlogn}=\infty$*, jer n raste brže nego* $2logn$*. Stoga je* n^2 *u skupu* $\Omega(2nlogn)$*.*

3.4 Analiza problema

Često se koristimo analizom algoritma ili neke implementacije algoritma u računalnom programskom obliku kako bi odredili složenosti ili određenu vrstu potrošnje. Iste tehnike se mogu primijeniti i za analizu problema. Ima smisla reći da gornja granica za problem ne može biti gora od gornje granice najboljeg algoritma za problem koji promatramo. No što bi značilo dati donju granicu za problem?

Napomena 3.9 *Pod pojmom najboljeg algoritma smatramo onaj koji ima najmanju složenost u najgorem slučaju.*

Uzmimo u obzir graf s određenom vrstom potrošnje za sve moguće ulazne podatke duljine n. Neka je dan problem P te A kolekcija algoritama (teoretski gledano postoji beskonačan broj takvih algoritama) koji rješavaju dani problem P. Promotrimo kolekciju grafova svih algoritama u A. Najgori slučaj donje granice je najmanja od svih najviših točaka na grafu. Stoga je puno lakše pokazati da je algoritam (ili program) u skupu $\Omega(f(n))$, nego pokazati da li je problem u skupu Ω(f(n)). Da bi problem bio u skupu $\Omega(f(n))$, tada bi svaki algoritam, koji rješava dani problem trebao biti u skupu $\Omega(f(n))$, čak i algoritmi kojih se ljudi još nisu dosjetili.

Razmotrit ćemo problem sortiranja kako bi malo bolje objasnili razliku između spomenutih notacija Θ, Ω i O. Koliko je najmanja složenost za algoritam sortiranja u najgorem slučaju?

3.4. Analiza problema

Algoritam mora barem pogledati svaki ulazni element u nizu kako bi se uvjerio da je niz uistinu sortiran. Dakle, bilo koji algoritam za sortiranje ima složenost barem cn, gdje je n broj ulaznih podataka. Ovo nas dovodi do donje granice, koja bi za pregledavanje elemenata niza, trebala biti Ω(n). Bubble Sort i Insertion sort imaju složenost $O(n^2)$ u najgorem slučaju. Dakle za problem sortiranja, može se reći da je gornja granica $O(n^2)$. No što je s razmakom između Ω(n) i $O(n^2)$? Postoji li bolji algoritam za sortiranje? Ako se ne možemo dosjetiti algoritma koji u najgorem slučaju ima bolju složenost od $O(n^2)$ i ako ne možemo pronaći tehniku koja će pokazati da je najmanja složenost sortiranja u najgorem slučaju veća od $\Omega(n)$, onda ne možemo sa sigurnošću tvrditi da postoji bolji algoritam za sortiranje od ovoga. No, o algoritmima sortiranja ćemo se više pozabaviti u idućim odlomcima.

3.5 Empirijska analiza

Do sad smo objasnili princip rada asimptotske analize i vidjeli smo da postoji dosta ograničenja u takvoj analizi, poput efekta na mali broj ulaznih podataka, određivanje preciznijih razlika između algoritama s istom stopom rasta i naslijeđene poteškoće pri definiranju matematičkih modela za kompleksnije probleme. Alternativa asimptotskoj analizi je empirijska analiza. Najočitija empirijska analiza je napraviti implementaciju programa i time odrediti koji algoritam je bolji. No, ovdje imamo poteškoća pri određivanju složenosti, jer veliku ulogu pri testiranju boljeg algoritma imaju kapacitet memorije, procesorska moć te sami programski jezik u kojem je implementacija algoritma napravljena. Također veliku ulogu ima i sama kvaliteta koda napisanog u programu, tj. pitanje da li jedan program napisan bolje od drugoga? Pojam "napisan bolje od drugog"odnosi se na kvalitetu samog programa, ne na njegovu ispravnost. Program treba biti ispravan, no njegova kvaliteta odnosi se na implementaciju koja može biti funkcijski program, proceduralna vrsta programa ili bilo koja druga. No, za određene probleme ukoliko je program ispravan, funkcijski način programiranja može biti dosta kvalitetniji, jer ga određeni programski jezici bolje podržavaju.

Poglavlje 4

Osnovne strukture podataka

Prije nego što pređemo na određene strukture podataka bitno je naglasiti da se većina složenih struktura podataka ne koristi (poput binarnih stabla pretrage), uglavnom se koriste jednostavnije strukture podataka (poput listi). Većinom se koriste kada se radi o malom broju podataka u kojima efikasnost, stoga i složenost pri odabiru određene strukture podataka nije bitna, primjer je prototip za aplikaciju.

Za "dobru" strukturu podataka jako je bitno prepoznati razliku funkcionalne definicije strukture podataka i njihove implementacije. Pod apstraktnom strukturom podataka (eng. "abstract data structure") smatramo skup objekata i operacija definiranih na tim objektima. Naprimjer, stog je apstraktna sruktura podataka koja podržava operacije "push" i "pop". Stog se može implementirati na razne načine, poput niza ili vezane liste. Također je jako bitno prezentirati korisniku strukturu podataka pomoću apstraktne definicije na njenim funkcijama bez da otkrivamo metode na koje se odnosi. Objasnimo malo pojam liste, s kojima ćemo se susretati u idućim poglavljima. Linearna lista ili kraće lista je osnovni apstraktni tip podatka. Listu smatramo nizom elemenata $(a_1, \ldots, a_n)$. Pritom ovdje ne specificiramo tip

elementa, tj. ne otkrivamo da li se radi o cijelom broju, znaku ili nečem trećem, obzirom da nije bitno za objašnjavanje samog pojma. Implementiranje u Javi bez navođenja tipa bilo bi korištenjem klase Objects. Duljina navedene liste je n, obzirom da sadrži n elemenata.

Osnovne operacije koje lista ima su:

- get(i) vraća element na i-toj poziciji, tj. a_i,
- set(i) postavlja vrijednost i-tog elementa na x,
- length() vraća duljinu liste,
- insert(i,x) ubacuje element x prije i-tog elementa u nizu, tj. prije a_i,
- delete(i) briše i-ti element.

4.1 Liste, stogovi i redovi

Iz potrebe za spremanjem stvari poput brojeva, obračuna plaće ili opisa poslova, javila se potreba za raznim strukturama podataka. Najefektivniji i najčešće upotrebljiv način spremanja činilo se spremanje u liste. No, prilikom sortiranja i pretrage za podacima, stvorila se potreba i za drugim strukturama podataka. Dosta aplikacija nema potrebe za sortiranjem ili poretkom spremanja podataka, dok druge zahtijevaju da se podaci spremaju u poretku u kojem su došle. Ovisno o potrebama aplikacije odabrat ćemo određenu strukturu podataka. Kroz ovo poglavlje, kao što sam naziv sugerira, bavit ćemo se listama, stogovima i redovima.

Težit ćemo prema sljedećem:

- Dati primjere odvajanja logičkih reprezentacija u obliku ADT-a ("Abstract data type") od fizičke implementacije za svaku strukturu poda-

taka

  - ADT je matematički model za strukture podataka koji specificira tip podatka koji se sprema, operacije koje podržava i tipove parametara za operacije

- Ilustrirati primjenu asimptotske analize u kontekstu nekih osnovnih, često korištenih operacija

Počet ćemo s definiranjem ADT-a za liste. Prezentirat ćemo dvije vrste implementacija liste: lista bazirana na nizu i „linked-list“ ili povezane liste, te zatim pokazati jednostavne implementacije stogova i redova.

4.1.1 Liste (eng. List)

Iako intuitivno znamo što znači pojam liste, definirat ćemo pojam liste i operacije koje liste koriste. Najbitniji koncept liste jest pozicija. Prema poziciji određujemo da li se neki element nalazi na prvom, drugom ili zadnjem mjestu. Gledat ćemo liste kao matematički koncept niza. Definiramo liste kao konačan, poredan niz podataka, koje zovemo elementima. U smislu poredan, smatramo da svaki element liste ima svoju poziciju.

Napomena 4.1 *Pod izrazom „poredan“ ne smatramo da je niz sortiran, tj. da elementi niza dobivaju poziciju na temelju svoje vrijednost i nekog uređaja* $\leq$.

Svaki element liste ima tip podatka. Mi ćemo promatrati liste u kojima svi elementi imaju isti tip podatka. Operacije definirane kao dio ADT-a ne ovise o tipu podataka elementa. Pod prethodnim, smatramo da se operacija može izvesti na nizu elemenata, čiji je tip podatka cijeli broj (eng. "Integer"), realni broj (eng. "Float"), znak (eng. "Char") ili niz znakova (eng. "String").

Za listu kažemo da je prazna ako ne sadrži niti jedan element. Broj elemenata u nizu nazivamo duljinom liste. Početni element niza naziva glavom, a krajnji element repom liste. Mi ćemo promatrati nesortirane liste, obzirom da ćemo kroz iduća poglavlja uvesti algoritme za sortiranje podataka. Prilikom prezentiranja elemenata niza koristit ćemo se vrijednostima pozicije kako ih vrednuje programski jezik Java. Dakle, prvi element liste ima poziciju nula, drugi poziciju jedan, n-ti element niza ima poziciju $n-1$. Prije odabira liste kao strukture podataka za korištenje prvo trebamo vidjeti koje operacije lista podržava, tj. koje operacije se s njom mogu izvesti. Intuicija nam sugerira da liste možemo smanjivati i proširivati po potrebi. Također možemo pristupiti elementu na temelju njegove pozicije te kreirati ili inicijalizirati listu te ju obrisati.

Sljedeći korak koji se definira po pravilima ADT-a je definiranje osnovnih operacija na listama.

Napomena 4.2 *Većina operacija koje će biti navedene su već definirane unutar sučelja List u programskom jeziku Java.*

Sučelje za programiranje je skup određenih pravila, kojima se programeri mogu služiti. Sastoji se od raznih bibiloteka (funkcija, procedura i metoda) kao i strukturama podataka, objektima i protoklima kojima se programeri moraju služiti. Prikaz sučelja liste. Kao što vidimo na slici 4.1.1, pored svake metode, prikazano je objašnjenje što pojedina metoda radi te koji su izlazni podaci. Varijabla E je prozvoljni tip ulaznog podatka.

Kroz programsko sučelje List, razne funkcije su već implementirane kao insert, moveToPos, koje ubacuju određenu vrijednost na određenu poziciju unutar liste. Funkcije poput next ili prev prebacuju pokazivač na poziciju koja je sljedeća ili prethodna. Funkcija getValue vraća referencu na element

Slika 4.1: Slika 4.1.1

```
/** List ADT */
public interface List<E> {
  /** Remove all contents from the list, so it is once again
      empty. Client is responsible for reclaiming storage
      used by the list elements. */
  public void clear();

  /** Insert an element at the current location. The client
      must ensure that the list's capacity is not exceeded.
      @param item The element to be inserted. */
  public void insert(E item);

  /** Append an element at the end of the list. The client
      must ensure that the list's capacity is not exceeded.
      @param item The element to be appended. */
  public void append(E item);

  /** Remove and return the current element.
      @return The element that was removed. */
  public E remove();

  /** Set the current position to the start of the list */
  public void moveToStart();

  /** Set the current position to the end of the list */
  public void moveToEnd();

  /** Move the current position one step left. No change
      if already at beginning. */
  public void prev();

  /** Move the current position one step right. No change
      if already at end. */
  public void next();

  /** @return The number of elements in the list. */
  public int length();

  /** @return The position of the current element. */
  public int currPos();

  /** Set current position.
      @param pos The position to make current. */
  public void moveToPos(int pos);

  /** @return The current element. */
  public E getValue();
}
```

na kojem se pokazivač trenutno nalazi, ako element na toj poziciji ne postoji program će vratiti grešku, tj. Exception. Pored prethodno navedenih postoji i funkcija find koja traži poziciju u nizu na kojoj se nalazi vrijednost koju tražimo. Operacije implementirane u sučelju liste u progranskom jeziku Java:

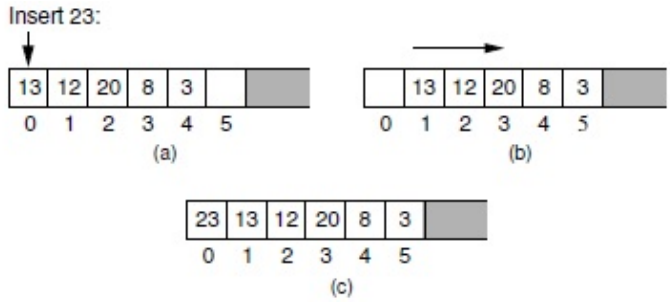
- Clear() – briše sve vrijednosti liste i pridružuje listi praznu listu
- Insert(vrijednost) – ubacuje vrijednost na trenutnu poziciju u listi
- Append(vrijednost) – ubacuje vrijednost na zadnju poziciju u listi
- Remove () – izbacuje element na određenoj poziciji i ispisuje njegovu vrijednost
- moveToStart() – miče pokazivač na početnu poziciju
- moveToEnd() – miče pokazivač na zadnju poziciju
- prev() – ispisuje vrijednost na poziciji prije trenutne
- next() – ispisuje vrijednost na poziciji iza trenutne
- length() – ispisuje broj elemenata liste
- currPos() – ispisuje trenutnu poziciju

4.1.2 Lista bazirana na nizu (eng. Array-Based List)

Pogledajmo programersko sučelje liste bazirane na nizu i usporedimo sa sučeljem obične liste. Lista bazirana na nizu nasljeđuje sve operacije definirane u sučelju List, koje smo prethodno naveli. No, iako su nasljeđene njihova implementacija je znatno drugačija radi različitog načina rada s listama baziranim na nizu u usporedbi s običnim listama. U programskom jeziku Java, ova klasa se označava sa AList koja uključuje klasu listArray, unutar koje su

```
/** Array-based list implementation */
class AList<E> implements List<E> {
  private static final int defaultSize = 10; // Default size
  private int maxSize;        // Maximum size of list
  private int listSize;       // Current # of list items
  private int curr;           // Position of current element
  private E[] listArray;      // Array holding list elements

  /** Constructors */
  /** Create a list with the default capacity. */
  AList() { this(defaultSize); }
  /** Create a new list object.
      @param size Max # of elements list can contain. */
  @SuppressWarnings("unchecked") // Generic array allocation
  AList(int size) {
    maxSize = size;
    listSize = curr = 0;
    listArray = (E[])new Object[size];  // Create listArray
  }

  public void clear()         // Reinitialize the list
    { listSize = curr = 0; }  // Simply reinitialize values

  /** Insert "it" at current position */
  public void insert(E it) {
    assert listSize < maxSize : "List capacity exceeded";
    for (int i=listSize; i>curr; i--) // Shift elements up
      listArray[i] = listArray[i-1];  //   to make room
    listArray[curr] = it;
    listSize++;               // Increment list size
  }

  /** Append "it" to list */
  public void append(E it) {
    assert listSize < maxSize : "List capacity exceeded";
    listArray[listSize++] = it;
  }

  /** Remove and return the current element */
  public E remove() {
    if ((curr<0) || (curr>=listSize))  // No current element
      return null;
    E it = listArray[curr];   // Copy the element
    for(int i=curr; i<listSize-1; i++) // Shift them down
      listArray[i] = listArray[i+1];
    listSize--;               // Decrement size
    return it;
  }
```

```
public void moveToStart() { curr = 0; } // Set to front
public void moveToEnd() { curr = listSize; } // Set at end
public void prev() { if (curr != 0) curr--; }   // Back up
public void next() { if (curr < listSize) curr++; }

/** @return List size */
public int length() { return listSize; }

/** @return Current position */
public int currPos() { return curr; }

/** Set current list position to "pos" */
public void moveToPos(int pos) {
  assert (pos>=0) && (pos<=listSize) : "Pos out of range";
  curr = pos;
}

/** @return Current element */
public E getValue() {
  assert (curr>=0) && (curr<listSize) :
         "No current element";
  return listArray[curr];
}
```



definirane sve metode (osim onih definiranih u klasi List) za rad s listama baziranim na nizu. Razlika od standardnih lista je samo stvaranje liste. Prilikom stvaranja liste bazirane na nizu potrebno je znati koliko elemenata će imati lista. Poredak elemenata unutar niza se podudara s poretkom unutar liste bazirane na nizu. Stoga je složenost dohvaćanja vrijednosti na i-toj poziciji koristeći naredbe moveToPos i getValue jednaka $\Theta(1)$.

Prikom ubacivanja elementa na zadnju poziciju potrebno je samo usmjeriti pokazivač na novu vrijednost. Dakle ne trebamo raditi ostale radnje poput zamjene elemenata unutar liste. Stoga je složenost ubacivanja elementa na zadnju poziciju jednaka $\Theta(1)$. No, ako poželimo ubaciti element na prvo mjesto, svi elementi u listi trebaju se pomaknuti za jedno mjesto s obzirom na svoju trenutnu poziciju. Stoga je složenost takve operacije Θ(n), ako je n elemenata već upisano u listu. Ako želimo ubaciti element na i-to mjesto tada $n-i$ elemenata se mora pomaknuti na sljedeću poziciju od svoje trenutne.

Ako ubacujemo element na početak liste bazirane na nizu, trebamo pomaknuti sve elemente za jedno mjesto prema "repu", tj. zadnjem elementu.

1. Lista sadrži 5 elemenata prije ubacivanja elementa 23.

2. Svaki element liste bazirane na nizu pomaknemo za jedno mjesto udesno.

3. Ubacujemo element 23 na nultu poziciju.

Pokažimo na primjeru bitnu razliku liste od liste bazirane na nizu.

```
public class ArrayListPrimjer {

   private ArrayList<Putnici> putnici = new ArrayList<>(20);

     public ArrayList<Putnici> dodajPutnika(Putnik putnik) {

     putnici.add(putnik);

     return putnici;
     }

 public ArrayList<Putnik> nadjiPutnika(String izvor) {

         return new ArrayList<Putnici>(putnici.stream()

             .filter(it -> it.getSource().equals(izvor))

             .collect(Collectors.toList()));

     }

}
```

Ovdje se koristimo listama baziranim na nizu za spremanje i vraćanje liste putnika. Obzirom da je najveći kapacitet putnika 20, početni kapacitet je postavljen na 20. Navedena implementacija radi ispravno, sve dok ne budemo morali mijenjati tip liste koju koristimo. Ako želimo nadograditi program s

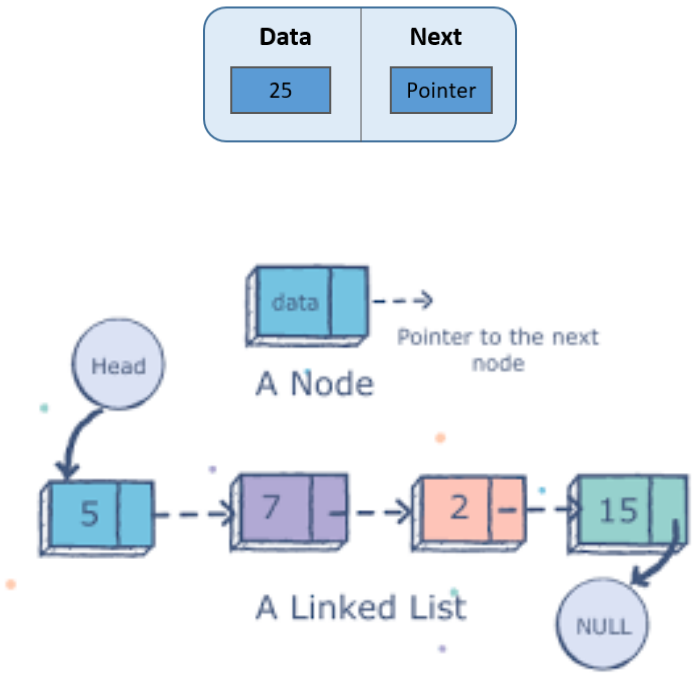
nekim drugim funkcionalnostima, naprimjer želimo upotrijebiti naš program za avio-tvrtke koje imaju znatno više putnika, trebali bi se koristiti strukturama podataka koje zauzimaju znatno manje memorije, npr. povezanim listama (eng. "LinkedList").

```
private LinkedList<Passenger> putnici = new LinkedList<>();
```

Povezane liste se koriste čvorovima za spremanje i dohvaćanje podataka. To bi značilo da se velik broj metoda treba promijeniti jer očekuju da rade sa listama baziranim na nizu. No, ako iz strukture podataka List želimo prebaciti na LinkedList, to radimo bez problema, jer su sve metode Liste, naslijeđene u LinkedList iz klase List.

Brisanje elementa na početnoj poziciji povezane liste sličan je dodavanju elementa na početnu poziciju, jer se mora pomaknuti n-i-1 element kako bi oslobodili mjesto (u slučaju dodavanja elementa) ili kako bi popunili prazno mjesto (u slučaju brisanje elementa), pomicanje se izvodi prema poziciji elementa koji je izbrisan.

U prosječnom slučaju za obje prethodno opisane radnje, složenost je Θ(n), obzirom da brisanjem ili dodavanjem elementa potrebno je zamjeniti poziciju elemenata preostalih u odnosu na poziciju koja je ostala prazna, ako se radi o brisanju elementa) ili kako bi oslobodili poziciju za novi element, ako se radi o dodavanju elementa. Većina elemenata za klasu Alist jednostavno pristupa ili pomiče element na zadanu vrijednost, jer pomoću vrijednosti pozicije znamo o kojem se elementu radi i lako je dohvatljiv jer ne moramo raditi dodatne radnje premještanja elemenata. Stoga je njihova složenost $\Theta(1)$.



Slika 4.2: Povezana lista, koja se sastoji od elemenata 5,7,2,15. Pokazivač prvog čvora pokazuje na iduću. Ukoliko se čvor nalazi na zadnjem mjestu, tada će pokazivač zadnjeg čvora pokazivati na null čvor.

4.1.3 Povezane liste (eng. LinkedList)

Sljedeća tradicionalna implementacija lista je uz pomoć vezanih listi koji se koriste dinamičkom memorijskom dodjelom, koja dodjeljuje memoriju za novi element liste ako je potrebno. Povezana lista se sastoji od skupa objekata, koji se nazivaju čvorovi liste. Čvor povezane liste se sastoji od vrijednosti elementa i pokazivača na iduću poziciju. Čvor koji nema vrijednost i ne pokazuje ni na šta naziva se null čvor.

S obzirom na to da je čvor liste različit od ćelije liste (pod ćelijom liste smatramo mjesto gdje je spremljena vrijednost čvora) u koju se spremaju elementi, dobro je imati još jednu listu gdje će biti spremljeni čvorovi liste.

```
/** Singly linked list node */
class Link<E> {
  private E element;        // Value for this node
  private Link<E> next;     // Pointer to next node in list

  // Constructors
  Link(E it, Link<E> nextval)
    { element = it;  next = nextval; }
  Link(Link<E> nextval) { next = nextval; }

  Link<E> next() { return next; }  // Return next field
  Link<E> setNext(Link<E> nextval) // Set next field
    { return next = nextval; }     // Return element field
  E element() { return element; }  // Set element field
  E setElement(E it) { return element = it; }
}
```

Slika 4.3: Prikaz sučelja za povezane liste u programskom jeziku Java

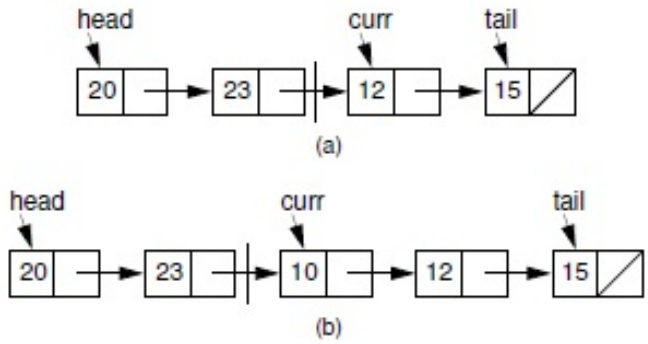
Dobrobit dodatne liste je ponovna upotreba u implementacijama stogova i redova.

Klasa u sučelju programskog jezika Java je Link. Objekti u klasi Link sadrže elemente u koja se spremaju vrijednosti i pokazivač na sljedeću poziciju.

Lista koncipirana na prethodno opisanom postupku naziva se jednostruko vezana lista. Kroz ovu strukturu podataka objasnit ćemo pojam „pokazivač“ koji se često koristi pri služenju sa strukturama podataka. Pokazivač, vizualno prikazan kao strelica, pokazuje na mjesto koje se trenutno razmatra, obrađuje, itd. Ako nema vrijednosti pokazivač dobiva, u Java programskom jeziku, naziv null. Ovo se najčešće događa kada se nalazimo na kraju vezanih listi, stogova i redova. Kako bi odredili što se nalazi na sljedećem mjestu koristimo se naredbom next koja pokazivač usmjeri na sljedeće mjesto s obzirom na poredak. Korištenjem pokazivača i spremanjem čvorova dobivamo lakši način ubacivanja elementa na određenu poziciju pomoću naredbe append.

Pojam „head“ i „tail“ odnosno prevedeno na hrvatski, glava i rep, također zadržavaju prethodno opisan smisao. Trenutni čvor koji se obrađuje dobivamo naredbom curr.

Razlika od liste bazirane na nizu je u samom instanciranju odnosno stva-



Slika 4.4: Ilustracija dodavanja u povezane liste.

1. curr pokazivač se nalazi na čvoru s vrijednosti 12 i pokazivačem na čvor s vrijednosti 15.

2. Prilikom dodavanja elementa s vrijednosti 10, kreirat ćemo čvor s vrijednosti 10, postaviti pokazivač curr na novi čvor čija je vrijednost na 12.

ranju povezane liste, obzirom da nema fiksan broj elemenata prilikom deklariranja nije potrebno navesti broj elemenata. Kao što smo već naveli operacije se nasljeđuju od klase List, no njihova implementacija je drugačija radi same strukture. Prođimo sad kroz složenost osnovnih operacija ove strukture podataka.

Operacija dodavanja ima složenost $\Theta(1)$. Opišimo postupak radi lakšeg razumijevanja složenosti. Ukoliko želimo ubaciti element na i-to mjesto, potrebno je stvoriti čvor, te pokazati da je vrijednost next pokazivača i-1 elementa upravo i-ti, novi čvor, te postaviti vrijednost next pokazivača novog čvora na čvor koji je prije dodavanja bio na i-tom mjestu.

Operacija brisanja elementa također ima složenost $\Theta(1)$. Zahvaljujući metodama pokazivača curr i next, samo postavimo da je curr, tj. trenutna

```
/** Stack ADT */
public interface Stack<E> {

  /** Reinitialize the stack.  The user is responsible for
      reclaiming the storage used by the stack elements. */
  public void clear();

  /** Push an element onto the top of the stack.
      @param it The element being pushed onto the stack. */
  public void push(E it);

  /** Remove and return the element at the top of the stack.
      @return The element at the top of the stack. */
  public E pop();

  /** @return A copy of the top element. */
  public E topValue();

  /** @return The number of elements in the stack. */
  public int length();
};
```

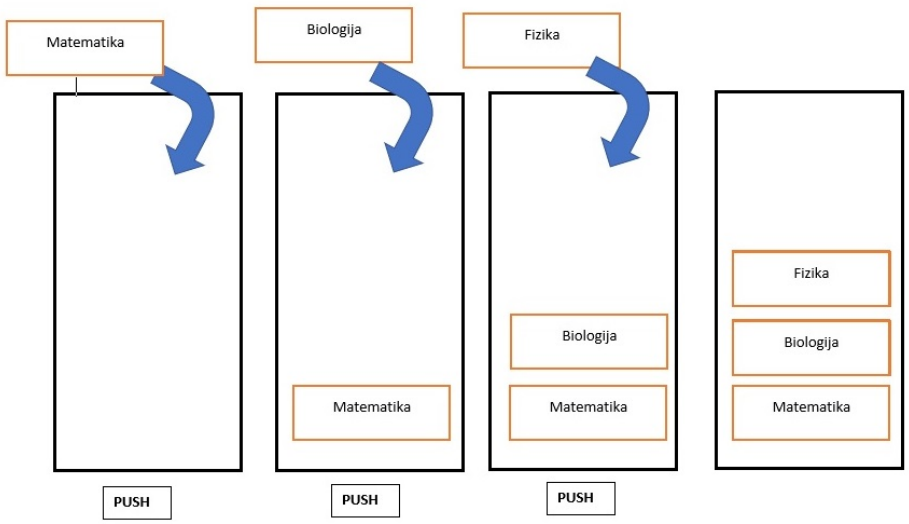
Slika 4.5: Programsko sučelje stoga u programskom jeziku Java

vrijednost pokazivača next, tj. sljedeći čvor. No, ako se ne želimo koristiti sljedećim čvorom, tj. naredbom next, nego prethodnim, tada je složenost znatno veća, tj. Θ (n), uz pretpostavku da je n elemenata ispred mjesta gdje brišemo. Problem je u tome što metoda postavljanja pred na curr pomiče za jednu mjesto bliže glavi, stoga je potrebno obići sve elemente koji se nalaze prije i pamtiti ih u zasebnoj listi.

4.1.4 Stogovi (eng. Stack)

Stogovi su strukture podataka nalik na liste, u kojima je ubacivanje elemenata moguće samo na početku ili na kraju. Iako navedeno ograničenje čini stogove manje fleksibilnim nego listama, pogodne su za određene operacije. Dosta aplikacija zahtjeva ograničenu mogućnost ubacivanja podataka i micanja podataka, što čini stogove pogodnijim nego liste.

Dostupni element u stogu se naziva top element. Za stogove se ne primjenjuje izraz ubacivanje (eng. „insert“) nego guranje (eng. „push“), te imamo



i izraz izbacivanja (eng. „pop“).

Primjer 4.3 *Ubacivanje vrijednosti.*

Prvo dodajemo vrijednosti:

```
Stack<String> stog = new Stack<>();

  stog.push("Matematika");

  stog.push("Biologija");

  stog.push("Fizika");
```

Ako želimo ispisati cijeli stog, to radimo ovako:

```
  System.out.println("stog = " + stack);
```

```
  Ispis će biti "stog=["Matematika","Biologija","Fizika"]
```

Ako želimo ispisati samo ono što se nalazi na vrhu
koristimo se metodom peek().

```
  System.out.println("stog.peek() = " + stog.peek())
```

```
  Ispis će biti: stog.peek()=Fizika
```

Ako želimo pronaći na kojoj se poziciji nalazi
riječ Matematika, to ćemo napraviti ovako:

```
  System.out.println("Matematika se nalazi na  = " + stog.search("Matematika"));
```

```
  Ispis: Matematika se nalazi na 3
```

Dakle imamo metodu search koja nam omogućuje pretragu
elemenata stoga i vraća poziciju na kojoj se nalazi.

Ako želimo izbrisati zadnji element, koristimo se metodom pop,
koja će nam osim brisanja i ispisati element koji je obrisan.

```
  System.out.println("stog.pop() = " + stog.pop());
```

```
  Ispis će biti stog.pop()=Fizika
```

Kao i s listama, postoji dosta vrsta stogova poput stogovima baziranim na nizu i vezanih stogova. No, mi ćemo proći samo kroz stogove bazirane na

```
/** Array-based stack implementation */
class AStack<E> implements Stack<E> {

  private static final int defaultSize = 10;

  private int maxSize;            // Maximum size of stack
  private int top;                // Index for top Object
  private E [] listArray;         // Array holding stack

  /** Constructors */
  AStack() { this(defaultSize); }
  @SuppressWarnings("unchecked") // Generic array allocation
  AStack(int size) {
    maxSize = size;
    top = 0;
    listArray = (E[])new Object[size];   // Create listArray
  }

  /** Reinitialize stack */
  public void clear() { top = 0; }

  /** Push "it" onto stack */
  public void push(E it) {
    assert top != maxSize : "Stack is full";
    listArray[top++] = it;
  }

  /** Remove and top element */
  public E pop() {
    assert top != 0 : "Stack is empty";
    return listArray[--top];
  }

  /** @return Top element */
  public E topValue() {
    assert top != 0 : "Stack is empty";
    return listArray[top-1];
  }

  /** @return Stack size */
  public int length() { return top; }
```

Slika 4.6: Programsko sučelje stoga baziranog na nizu u programskom jeziku Java

nizu.

4.1.5 Stogovi bazirani na nizu (eng. Array-based stacks)

Kao i liste bazirane na nizu, stogovi bazirani na nizu se definiraju s fiksnom početnom količinom elemenata. Metoda top koristi se za postavljanje pokazivača na početni element, tj. onoga koji je dostupan.

Operacije

- push ili ubacivanje elementa na vrh stoga
- clear ili brisanje svih elemenata stoga
- pop ili brisanje zadnje vrijednosti stoga
- topValue ili vrijednost elementa koji se nalazi na vrhu
- length ili duljina, tj. broj elemenata unutar stoga

Ako imamo n elemenata u stogu i želimo izbaciti zadnji element, potrebno je izbaciti sve elemente koji se nalaze ispred njega, zatim im pridružiti novu vrijednost u istom poretku, što zahtjeva složenost $\Theta(n)$. U slučaju da smo odlučili da stog kreće od kraja, onda s metodom pop izbacimo element i to zahtjeva složenost $\Theta(1)$.

4.1.6 Redovi (eng. Queue)

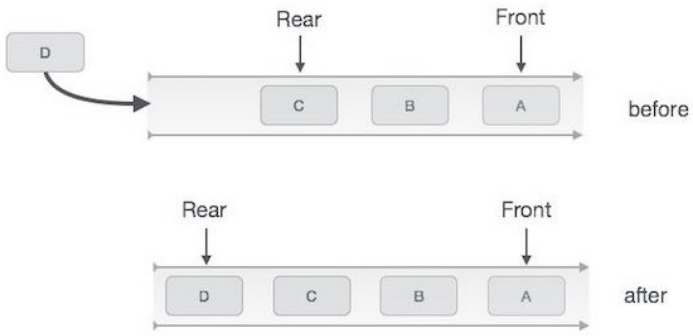
Kao i stogovi, redovi su strukture podataka slične listama koje pružaju ograničeni pristup elementima. U redovima je izbacivanje elemenata moguće samo u slučaju da se element nalazi na početnoj ili krajnjoj poziciji. Za ubacivanje elemenata koristi se implementirana metoda enqueue unutar sučelja Java programskog jezika, dok se za izbacivanje koristi dequeue.

Primjer 4.4 *1. Provjeriti da li je red pun*

2. Ako je red pun, pošalji grešku "Overflow error"

3. Ako red nije pun, povećaj za 1 pokazivač na zadnjem mjestu

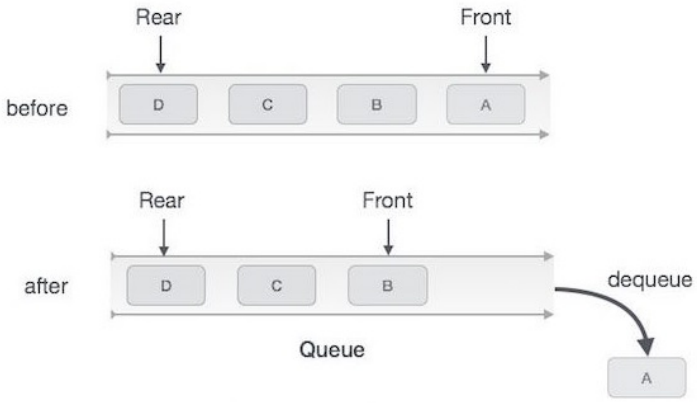
4. Dodaj element na zadnje mjesto gdje smo usmjerili pokazivač.



Slika 4.7: Primjer enqueue operacije.

5. Vrati uspjeh.

Pseudokod algoritma bi izgledao ovako:



Slika 4.8: Primjer dequeue operacije.

```
Početak algoritma enqueue(podatak)

   Ako je red pun
      return overflow
   Inače

   zadnji = zadnji + 1
   red[zadnji] = podatak
   return true

kraj algoritma
```

Primjer 4.5 *1. Provjeriti da li je red prazan*

2. *Ako je red prazan, pošalji grešku "Underflow error" i izađi*
3. *Ako red nije prazan, pristupi podatku na koji prikazuje pokazivač 'front'*
4. *Povećaj vrijednost pokazivača front na podatak koji se nalazi za jedno mjesto iza trenutnog prvog elementa.*

5. Vrati uspjeh.

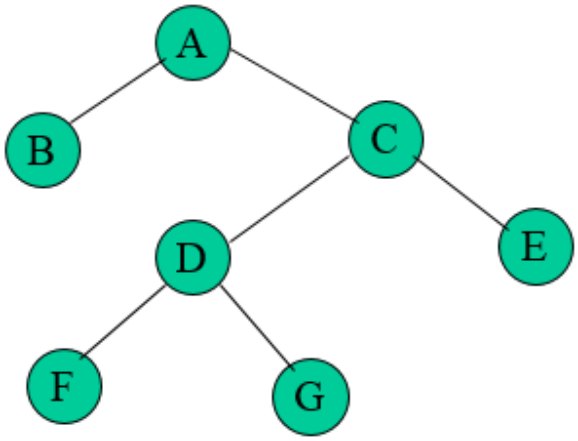
Algoritam bi izgledao ovako:

```
Početak algoritma dequeue

   Ako je red prazan
       return underflow
   inače

   podatak = red[front]
   front = front + 1
   return true

kraj algoritma
```

Redove možemo gledati isto kao što ih gledamo u životu. Kad stojimo u redu, kad nova osoba dođe, ide na kraj reda (uz pretpostavku da osoba neće varati i proći ispred svih).

4.1.7 Implementacija reda pomoću niza (eng. Array implementation of queue)

Implementacija pomoću niza su teški u implementaciji i nisu efikasni. U slučaju ubacivanja novih elemenata potrebno je pomaknuti element za jedno mjesto svaki put kad se prvi element izbriše. Stoga je za brisanje elementa koji se nalazi na kraju potrebna složenost $\Theta(1)$. No, ako želimo izbrisati prvi



Slika 4.9: Primjer binarnog stabla

element, potrebno je svaki idući element pomaknuti za jedno mjesto bliže prvom elementu, što zahtjeva složenost Θ (n).

4.2 Binarna stabla

Binarna stabla se sastoje od konačnog skupa elemenata koje nazivamo čvorovima. Skup elemenata binarnog stabla je ili prazan ili se sastoji od čvora kojeg nazivamo korijen (eng. „root“), te dva binarna stabla koje nazivamo podstabla, koji imaju zajednički korijen. Korijeni podstabla se nazivaju djecom, korijen stabla na kojeg su povezana sva podstabla naziva se roditelj čvor. Ako imamo niz elemenata $n_1, n_2, \ldots, n_k$, gdje je n_i roditelj od n_i+1 za $0 \leq i \leq k-1$, tada se ovaj niz naziva put od n_1 do n_k. Duljina ovog puta je k-1. Ako postoji put od čvora R do čvora M, onda je R predak, a M nasljednik. Samim time, svi čvorovi osim korijena su korjenovi nasljednici, a korijen je predak svima. Uvedimo nekoliko pojmova o stablima:

1. Dubina čvora M u stablu je duljina puta od korijena stabla do M.

Ako putova od korijena do čvora M jednake duljine ima više uzimamo proizvoljni.

2. Visina stabla je za jedan više od najvećeg puta u stablu.

3. Svi čvorovi kojima je dubina n imaju stupanj n (stupanj se definira kao broj djece), osim korijena koji ima stupanj 0.

4. List je čvor koji nema djece.

5. Unutarnji čvor je čvor koji ima barem jedno dijete i nije korijen.

6. U binarnom stablu svaki čvor ima svoje dijete.

7. Potpuno binarno stablo je stablo čijim se čvorovima mogu dati imena $0, \ldots,$ n tako da za svaki čvor i:

   (a) lijevo dijete čvora ima ime $2i+1$, ako je $2i+1 < n$, inače to dijete ne postoji

   (b) desno dijete čvora ima ime $2i+2$, ako je $2i+2 < n$, inače to dijete ne postoji

Teorem 4.6 *Broj listova u nepraznom potpunom binarnom stablu je za jedan veći od broja unutarnjih čvorova.*

Dokaz.

Za dokaz ćemo koristiti matematičku indukciju, gdje je n broj unutarnjih čvorova.

Baza indukcije.

Neprazno binarno stablo s 0 unutarnjih čvorova ima jedan list čvor. Potpuno binarno stablo ima jedan unutarnji čvor s dva lista. Stoga, tvrdnja vrijedi za n=0 i n=1.

Pretpostavka indukcije.

Pretpostavimo da tvrdnja vrijedi za potpuno binarno stablo T s $n-1$ unutarnjih čvorova te n listova.

Korak indukcije.

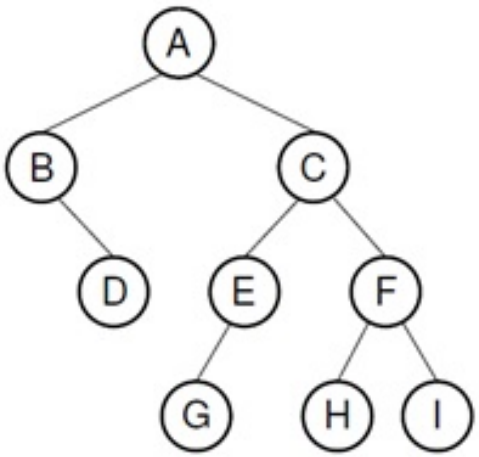
Neka je T' stablo s n unutarnjih čvorova. Odaberimo jedan unutarnji čvor I čija su oba djeteta listovi. Uklonimo oba čvora listova I-tog unutarnjeg čvora. Tada je I list čvor i stablo T' sada ima $n-1$ unutarnjih čvorova, pa vrijedi pretpostavka indukcije, tj. stablo ima n listova. Vratimo sad oba čvora djeteta čvoru I. Sada čvor I nije više list, nego unutarnji čvor i imamo dva lista. Stoga ukupno imamo n+1 list, tj. vrijedi tvrdnja. Po matematičkoj indukciji, teorem vrijedi za svaki n $\in \mathbb{N}_0$. ■

Pri analizi potrebne memorije za implementaciju binarnog stabla, korisno je znati koliko praznih podstabala ima. Stoga ćemo dokazati sljedeći teorem koji slijedi iz teorema o potpunim binarnim stablima te samim time dobiti jasniju sliku o implementaciji stabala u programskom jeziku.

Teorem 4.7 *Broj praznih podstabala u nepraznom binarnom stablu je za jedan više od broja svih čvorova u stablu.*

Dokaz. Neka je T binarno stablo s n čvorova. Po definiciji, svako binarno stablo T ima dva čvora koja su mu djeca. Stoga, stablo T ima ukupno 2n čvorova djece. Svaki čvor osim korijena ima jednog roditelja, pa imamo $n-1$ čvorova roditelja. Drugim riječima postoji $n-1$ nepraznih čvorova djece. S obzirom na to da je ukupan broj čvorova djece jednak 2n, preostaje nam n+1 čvorova djece koji su prazni.

■



4.2.1 Obilazak binarnog stabla

Često se u programiranju želi promatrati čvorove stabla načinom da se posjećuje svaki čvor i pri svakoj posjeti obavi neku radnju. Navedenu radnju nazovimo obilazak stabla (eng. "traversal“) . Svaki obilazak stabla koji ispisuje svaki čvor koji posjeti nazivamo nabrajanjem čvorova stabla.

Većina aplikacija ne zahtijeva da se čvorovi obilaze po određenom poretku, bitno je samo da se svaki čvor posjeti točno jednom. Neke aplikacije ipak zahtijevaju da se poštuje nekakav poredak, zbog održavanja povezanosti čvorova. Naprimjer, možemo zahtijevati da se prvo posjete roditelj čvorovi, pa zatim djeca čvorovi. Prethodno opisan način obilaska stabla nazivamo unaprijed određeni obilazak (eng. "preorder traversal").

Primjer 4.8 *Recimo da imamo sljedeće stablo i da želimo obići stablo unaprijed određenim obilaskom.*

Ispis bi bio: ABDCEGFHI

Postupak bi išao ovako. Prvo se ispiše korijen čvor, zatim svi čvorovi lijevog podstabla (po unaprijed određenom obilasku), prije bilo kojeg čvora desnog stabla.

Alternativno, možda bi željeli posjetiti svaki čvor nakon što posjetimo njegovu čvor djecu, odnosno podstabla, što bi bilo korisno ako želimo staviti čvor na prvo slobodno mjesto ili ako želimo izbrisati neki čvor dijete.

Prethodno opisan obilazak naziva se naknadno određen obilazak (eng. „postorder traversal“).

Primjer 4.9 *Uzet ćemo binarno stablo prikazano na prethodnom primjeru i pokazati metodu ispisa naknadno određenim obilaskom.*

Ispis: DBGEHIFCA.

„Prirodni“ obilazak smatra se posjećivanje prvo lijeve djece (uključujući cijelo podstablo), zatim posjećivanje čvora te posjećivanje desne djece (uključujući cijelo podstablo).

Primjer 4.10 *Na istom binarnom stablu prikaz „prirodnog“ obilaska.*

Ispis: BDAGECHFI

Obilazak binarnog stabla većinom se implementira pomoću rekurzije. Ulazni parametar je referenca na čvor, kojeg ćemo nazvati rt. Poziv na rekurziju započinje prosljeđivanjem reference na korijen ili stablo. Funkcija obilaska posjeti rt čvor i njegovu čvor djecu.

U primjeru unaprijed određenog obilaska posjetio bi prvo sebe, tj. čvor rt, zatim svoje čvorove djecu. Funkcija unaprijed određenog obilaska prvo provjerava da li je stablo prazno, ako je onda obilazak završava. Inače, posjetit će samog sebe, tj. čvor rt. Nakon toga obilazi rekurzivno lijevo podstablo, posjeti sve čvorove u tom podstablu, zatim obilazi rekurzivno desno podstablo. Na isti način se rade svi obilasci s izmjenom poretka izvršavanja rekurzivnih funkcija.

Ako usporedimo prethodno dva navedena primjera, možda se čini da je metoda unaprijedOdređeniObilazak2 efikasnija nego unaprijedOdređeniObilazak,

```
funkcija unaprijedOdređeniObilazak ( Čvor rt)
{
    Ako je rt jednak nula stani
    Posjeti(rt)
    UnaprijedOdređeniObilazak( Čvor lijevo od rt)
    UnaprijedOdređeniObilazak ( Čvor desno od rt)
}
```

```
Funkcija unaprijedOdređeniObilazak2( Čvor rt)
{
    Posjeti(rt)
    Ako je lijevi čvor različit od null čvora,
        pozovi unaprijedOdređeniObilazak2( lijevi čvor od rt)
    Ako je desni čvor različit od null čvora,
        pozovi unaprijedOdređeniObilazak2(desni čvor od rt)
}
```

jer ima dvostruko manje rekurzivnih poziva. S druge strane unaprijedOdređeniObilazak2 mora pristupiti lijevom i desnom čvor-djetetu dva puta češće. Rezultat usporedbe ova dva algoritma je da zapravo nema značajnog poboljšanja. U stvarnosti, drugi algoritam se ne koristi češće radi upotrebe null čvora, jedan od mnogih razloga je što se pri testiranju dva puta mora testirati da li je čvor null, dok se u prvom algoritmu provjerava samo jednom.

Drugi pristup obilasku stabla je pomoću funkcije obilaska koja za parametar prima posjetitelja, tj. čvor koji je posjećen. Ovaj princip nazivamo dizajn posjetitelja. Veliko ograničenje ovakvog pristupa leži u tome što za sve posjetitelje funkcije povratni parametri moraju biti fiksno određeni na početku. Stoga, dizajner algoritma mora uzeti u obzir sve čvorove kako bi donio adekvatnu odluku za povratni tip podataka i parametre.

Rukovanje s tokom informacija prilikom prolaska kroz čvorove može biti jako zahtjevno ako se radi s rekurzijama. Problemi najčešće nastupaju prilikom prosljeđivanje ispravnih podataka ili pri vraćanju ispravnih podataka.

U prethodno navedenom pristupu prvo ćemo razmotriti jednostavne slučajeve

u kojima računanje zahtjeva komunikaciju od lista do korijena. Još jedan od problema ove metode je samo određivanje koji čvor treba biti posjećen.

Naprimjer, želimo provjeriti imaju li svi čvorovi u lijevom podstablu manju vrijednost od roditelj čvora ili imaju li svi čvorovi desnog podstabla veću vrijednost od roditelj čvora, za svaki roditelj čvor? U ovom slučaju nije dovoljno utvrditi za svako podstablo da je veće od svog roditelj čvora, jer je potrebno sagledati veću sliku, možda naprimjer taj roditelj čvor ne zadovoljava relaciju u odnosu na svog roditelj čvora itd.

4.2.2 Primitivne operacije binarnog stabla

Prevođenjem primitivnih operatora za binarna stabla lako možemo raspoznati što rade.

- Imamo dva konstruktora za kreiranje stabla:
  - EmptyTree, što vraća prazno stablo
  - MakeTree(v,l,r), koji kreira binarno stablo od korijena v sa dva binarna stabla l i r

  Napomena 4.11 *Konstruktor je posebna metoda koja se koristi za inicijaliziranje, tj. stvaranje novih objekata u strukturama podataka.*

- Uvjeti za testiranje da li je stablo prazno:
  - isEmpty(t), koji vraća vrijednost istina ili laž za stablo t
- Selektori za odvajanje stabla na podstabla ili korijen
  - Root(t), koji vraća vrijednost korijena binarnog stabla t
  - Left(t), koji vraća lijevo podstablo binarnog stabla t

– Right(t), koji vraća desno podstablo binarnog stabla t

Navedene operacije se mogu koristiti za kreiranje svih potrebnih algoritama kojima manipuliramo stablima. Za lakše upravljanje možemo sami kreirati svoje konstruktore koji kreiraju binarno stablo od čvora i dva prazna podstabla. Pokažimo na kreiranju vlastite klase unutar Java programskog jezika.

```
  class Cvor {

      int vrijednost;

     Cvor lijevi;

      Cvor desni;

      Cvor(int vrijednost) {

          this.vrijednost = vrijednost;

          desni = null;

          lijevi = null;

      }
}
public class Binarnostablo {

      Cvor korijen;
```

```
}
```

4.2.3 Binarno stablo pretrage

Binarno stablo pretrage je tip stabala koji pružaju efikasniji način spremanja podataka te sami način pronalaska podataka. Za stablo kažemo da je binarno stablo pretrage ako zadovoljava sljedeće svojstvo:

- Za svaki roditelj čvor koji ima vrijednost K, svi čvorovi u lijevom podstablu roditelja imaju vrijednost manju od K.
- Za svaki roditelj čvor koji ima vrijednost K, svi čvorovi u desnom podstablu roditelja imaju vrijednost veću ili jednaku K.

Posljedica navedenog svojstva je da će vrijednosti čvorova biti spremljene u poretku od najmanjeg do najvećeg što pretragu čini manje složenom, jer je lista vrijednosti čvorova sortirana. Zbog sortiranosti vrijednosti čvorova koristit ćemo se samo listom ključeva jer vrijednost prvog ključa odgovara najmanjoj vrijednosti, dok vrijednost zadnjeg ključa liste odgovara najvećem elementu.

4.2.4 Pretraga sa listama ili nizovima

Kao što smo vidjeli , većina aplikacija računalne znanosti uključuje pretragu u nekom smislu, za određeni podatak ili skup podataka. Ako bi podatke spremili u nesortirani niz ili listu, onda bi pretraga određenog podatka zahtjevala pretragu cijelog niza dok se određeni podatak ne pronađe. Za n elemenata niza, u prosječnom slučaju zahtjeva $\Theta(\frac{n}{2})$, a u najgorem slučaju $O(n)$ pretraga.

No, ako se podaci sortiraju prije spremanja u niz, binarna pretraga zahtjeva $\Theta(log_2 n)$ pretraga u najgorem i u prosječnom slučaj. Ali, uz ovo treba uzeti i obzir složenosti i vrijeme potrebno za sortiranje kao i održavanje u slučaju brisanja elementa. Ideja ovog je da uz pomoć binarne pretrage možemo ubrzati proces spremanja kao i pretrage određenog elementa. Rješenje problema može biti u spremanju skupa podataka za pretragu pomoću binarnog stabla, čime bi dobili pretragu s minimalnim gubitcima bilo u vremenskom, memorijskom ili u vidu složenosti.

Ideja je jednostavna, za svaki čvor stabla želimo da nam vrijednost čvora govori da smo našli odgovarajući element ili da se element nalazi u lijevom ili desnom podstablu. Pretpostavit ćemo da su sve vrijednosti čvorova različite. Stoga definiramo:

Definicija 4.12 *Binarno stablo pretrage je binarno stablo koje je ili prazno ili zadovoljava sljedeće uvjete:*

- *Sve vrijednosti u lijevom podstablu su manje nego korijen podstabla*
- *Sve vrijednosti u desnom podstablu su veće nego korijen podstabla*
- *Lijeva i desna podstabla su također binarna stabla pretrage.*

Stoga imamo stablo koje je poseban tip binarnog stabla s vrijednostima čvora koji su „ključevi pretrage“. Samim tim, nasljeđujemo sve operacije koje se koriste u običnim binarnim stablima.

4.2.5 Izgradnja binarnog stabla pretrage

Prilikom izgradnje binarnog stabla, prirodno je krenuti od korijena, zatim dodavati nove čvorove po potrebi. Stoga, za dodavanje nove vrijednosti v pratimo sljedeće korake:

- Ako je stablo prazno, pridružimo vrijednost v korijen čvoru i ostavimo lijeva i desna podstabla prazna.

```
insert(v,bst)
{
  if(isEmpty(bst))
    return MakeTree(v, EmptyTree, EmptyTree)
```

- Ako stablo nije prazno, ubacimo čvor s vrijednosti v na sljedeći način:
  - Ako je vrijednost v manja nego vrijednost korijena, onda ubacimo čvor s vrijednosti v u lijevo podstablo

```
elseif(v < root(bst) )
   return MakeTree(root(bst), insert(v,left(bst)),right(bst))
```

  - Ako je vrijednost v veća nego vrijednost korijena, onda ubacimo čvor s vrijednosti v u desno podstablo

```
elseif(v > root(bst) )
   return MakeTree(root(bst),left(bst), insert(v, right(bst)))
  }
```

  - Inače, javi poruku da vrijednosti moraju biti različite od korijena

Napomena 4.13 *Ukoliko zahtijevamo da stablo pretrage dopušta jednake vrijednosti, potrebno je modificirati algoritam tako da provjeravamo da li je vrijedost manja ili jednaka, odnosno veća ili jednaka.*

4.2.6 Pretraga binarnog stabla pretrage

Pretraga binarnog stabla pretrage ne razlikuje se mnogo od dodavanja novog čvora s novom vrijednošću u binarno stablo pretrage. Jednostavno uspoređimo vrijednost s korijenom, pa zatim s lijevim i desnim podstablom sve dok ne dođemo do odgovarajuće pozicije.

Algoritmi se mogu izraziti na razne načine. Opisat ćemo jedan način algoritma pretrage:

Za pretragu vrijednosti v u binarnom stablu pretrage T radimo sljedeće.

1. Ako je stablo prazno, onda se v ne nalazi u stablu T, stoga stanemo i vratimo laž.

2. Inače, ako je vrijednost v jednaka korijenu, onda se vrijednost pojavljuje u stablu, stoga stanemo i vratimo istina.

3. Ako je vrijednost v manja od korijena, onda je po definiciji binarnog stabla pretrage dovoljno usporediti s vrijednostima u lijevom podstablu.

4. Inače, ako je vrijednost v veća od vrijednosti korijena, onda uspoređujemo s vrijednostima u desnom podstablu. Pretraga u podstablima se radi isto kao i s korijenom i njegovim podstablima.

Primijetimo da ovaj opis algoritma obuhvaća oba koraka koja su potrebna za zaustavljanje i razlog zaustavljanja. Ovaj način opisivanja je vrlo čest kada nešto ne želimo implementirati i pokrenuti u računalnom programu. No ako želimo sugerirati kako bi algoritam trebao izgledati, možemo ga napisati u programskom jeziku Java:

```
isIn(value v, tree t) {
   if ( isEmpty(t) )
      return false
   elseif ( v == root(t) )
      return true
   elseif ( v < root(t) )
      return isIn(v, left(t))
   else
      return isIn(v, right(t))
}
```

4.2.7 Vremenska složenost ubacivanja čvorova i sortiranja binarnog stabla pretrage

Promotrimo vremensku složenost algoritama. Ubacivanje i pretraga elementa u binarnom stablu pretrage zahtijevat će broj usporedbi kao i u određivanju visine binarnog stabla plus jedna usporedba. U najgorem slučaju složenost će biti jednaka broju čvorova u binarnom stablu pretrage.

Da bi znali odgovor na pitanje vremenske složenosti prvo moramo odrediti prosječnu visinu binarnog stabla pretrage. Prosječnu visinu binarnog stabla pretrage možemo izračunati uzimajući u obzir sva binarna stabla pretrage s n čvorova i izračunati svaku visinu, što samo po sebi nije jednostavan zadatak. Problem je što postoje različiti način izgradnje stabla za isto binarno stablo pretrage.

Maksimalna visina binarnog stabla s n čvorova je $n-1$, što je slučaj kada svi unutarnji čvorovi imaju točno jedan list, formiranjem nečega što u laičkom smislu liči na lanac. S druge strane, pretpostavimo da imamo n čvorova i da želimo izgraditi binarno stablo s najmanjom visinom. To možemo napraviti dodavanjem čvorova redom počevši od vrha. Možemo dodavati na

jednu razinu, neovisno o popunjenosti drugih čvorova na istoj razini. Bitno je samo da prelazimo na popunjavanje druge razine tek nakon što popunimo prethodnu razinu. Stablo dobiveno ovakvom izgradnjom nazivamo savršeno balansirano stablo. Savršeno balansirano stablo ima najmanju visinu za dani broj čvorova. što je stablo više balansirano to postoji više načina za izgradnju. Ako uzmemo u obzir sve mogućnosti poretka ubacivanja za n čvorova, u binarno stablo pretrage, sve mogućnosti su jednako vjerojatne. Tada je prosječna visina za binarno stablo pretrage jednaka $O(log_2 n)$, iz čega slijedi da je broj potrebnih operacija uspoređivanja jednak $O(log_2 n)$, koji je jednak složenosti algoritma binarne pretrage u sortiranom nizu.

Dodavanje novog čvora također ovisi o visini binarnog stabla pretrage, stoga je i za ovaj proces potrebno $O(log_2 n)$ koraka. Ako usporedimo složenost dodavanja novog čvora u binarno stablo pretrage i dodavanje novog elementa na određeno mjesto u sortiranom nizu, čija je složenost O(n), operacija dodavanja novog elementa u binarno stablo pretrage je manje složena nego dodavanje novog elementa u niz.

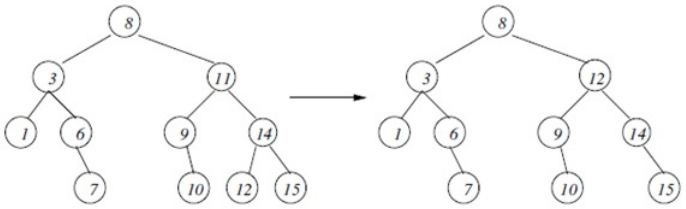
4.2.8 Brisanje čvorova iz binarnog stabla pretrage

Ako želimo izbrisati neki čvor ili vrijednost iz stabla, bilo bi neefikasno kad bi ostatak stabla radili od početka. Za n elemenata bilo bi potrebno n koraka složenosti $O(log_2 n)$, stoga je ukupna složenost $O(nlog_2 n)$. Ukoliko bi usporedili sa složenosti izbacivanja elementa iz niza, koja iznosi O(n), složenost bi trebala biti manja, a ne veća kao u ovom slučaju.

Stoga ćemo kreirati pseudokod koji ažurira binarno stablo pretrage efikasnije nego prethodno opisano.

Koraci su sljedeći:

- Ako je čvor list, samo ga ukloni,



- Ako je samo jedan čvor od podstabla neprazan, onda taj čvor pomaknemo na mjesto uklonjenog čvora,
- Ako čvor ima dva neprazna čvora u podstablu, nađi najljeviji čvor koji se pojavljuje u desnom podstablu (to je najmanja vrijednost u desnom podstablu). Ovaj čvor koristimo za zamjenu vrijednosti koja će biti izbrisana.

Zadnja stavka ispravno radi zato što je najljeviji čvor u desnom podstablu veći po vrijednosti od svih čvorova u lijevom podstablu i manji po vrijednosti od svih čvorova u desnom podstablu i nema čvora lijevo od njega.

Primjer 4.14 *Pokažimo na primjeru. Ako izbrišemo vrijednost 11 kako izgleda novo stablo?*

U praksi, algoritamski opis ćemo prevesti u detaljniji algoritam specificiran za primitivne operacije stabla:

Ako je stablo prazno, znači da se vrijednost ne nalazi u stablu i javimo grešku shodno tome.

Proces "delete", tj. brisanje, koristi dva podalgoritma za pronalazak i uklanjanje najmanje vrijednosti danog podstabla. Kako će podstabla uvijek biti neprazna, možemo ih napisati bez uvjeta provjeravanja. No, odgovornost da su podstabla neprazna je na programeru koji postavlja uvjete. Ako ne naglasi da su stabla neprazna, uvjet provjere se treba dodati.

Za pronalazak najmanjeg čvora u stablu imamo sljedeći algoritam koji

```
delete(value v, tree t) {
  if ( isEmpty(t) )
     error('Error: given item is not in given tree')
  else
     if ( v < root(t) )    // delete from left sub-tree
        return MakeTree(root(t), delete(v,left(t)), right(t));
     else if ( v > root(t) )     // delete from right sub-tree
        return MakeTree(root(t), left(t), delete(v,right(t)));
     else    // the item v to be deleted is root(t)
        if ( isEmpty(left(t)) )
           return right(t)
        elseif ( isEmpty(right(t)) )
           return left(t)
        else    // difficult case with both subtrees non-empty
           return MakeTree(smallestNode(right(t)), left(t),
                                          removeSmallestNode(right(t))
}
```

```
smallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return root(t)
  else
     return smallestNode(left(t));
}
```

```
removeSmallestNode(tree t) {
  // Precondition: t is a non-empty binary search tree
  if ( isEmpty(left(t) )
     return right(t)
  else
     return MakeTree(root(t), removeSmallestNode(left(t)), right(t))
}
```

koristi činjenicu da je po pretpostavci stablo binarne pretrage. Najmanja vrijednost čvora je najljeviji čvor. Rekurzivno pretražuju lijevo podstablo dok ne dođe do praznog čvora. Prilikom dolaska do praznog podstabla vratit će vrijednost korijena tog podstabla.

Drugi podalgoritam ima sličnu ideju, osim povratne vrijednosti koja je cijelo podstablo. Navedeni algoritmi su ujedno i primjeri rekurzije. Za rekurziju znamo da će završiti zato što svaki rekurzivni poziv uključuje stablo manje od prethodnog sve dok ne nađe prazno stablo.

Jasno je iz navedenih algoritama da brisanje čvora zahtijeva jednak broj koraka za pretragu čvora kao i ubacivanje čvora, tj. zahtjeva prosječnu visinu binarnog stabla pretrage, tj. $O(log_2 n)$, gdje je n ukupan broj čvorova u stablu.

4.2.9 Provjera uvjeta za binarna stabla

Postoji još jedan uvjet koji moramo provjeriti, ako nije naglašeno u preduvjetima da je stablo uistinu binarno stablo pretrage, a to je provjeriti da li je neko binarno stablo stvarno binarno stablo pretrage?

Znamo da je prazno stablo trivijalno binarno stablo pretrage i da sve vrijednosti čvorova u lijevom podstablu moraju biti manje od vrijednosti korijena podstabla. Također sve vrijednosti u desnom podstablu trebaju biti veće od vrijednosti korijena.

Stoga, algoritam provjera glasi: No, ovaj algoritam nije nužno najefi-

```
isbst(tree t) {
   if ( isEmpty(t) )
      return true
   else
      return ( allsmaller(left(t),root(t)) and isbst(left(t))
                and allbigger(right(t),root(t)) and isbst(right(t)) )
}


 allsmaller(tree t, value v) {
    if ( isEmpty(t) )
       return true
    else
       return ( (root(t) < v) and allsmaller(left(t),v)
                                          and allsmaller(right(t),v) )
 }

 allbigger(tree t, value v) {
    if ( isEmpty(t) )
       return true
    else
       return ( (root(t) > v) and allbigger(left(t),v)
                                          and allbigger(right(t),v) )
 }
```

kasniji. Trenutno je najefikasniji, no možda se tijekom vremena pronađe još efikasniji način provjere.

Poglavlje 5

Sortiranje i pretraga

U svakodnevnom životu sortiramo dosta stvari, bilo odjeću, karte, itd. Imamo dosta strategija i načina kako sortirati određene stvari, ovisno o vrsti i količini. Sortiranje je ujedno i jedan od najpoznatijih računalnih zadataka. Možda sortiramo podatke pjesama u bazi podataka kako bi lakše pronalazili ili sortiramo gradove prema poštanskim brojevima.

Zbog velike važnosti sortiranja proučavaju se algoritmi i različiti načini sortiranja. Neki konstruirani algoritmi su očiti i jasni, dok su drugi neshvatljivi na prvi pogled. No iako su dosta istraženi, svejedno postoji dosta problema povezanih sa sortiranjem koji nisu riješeni.

Glavni zadatak ovog poglavlja je izricanje problema prilikom dizajniranja i analize algoritama. Naprimjer, postoji kolekcija algoritama koji na različite načine koriste tehniku „podijeli pa vladaj", kao što su Mergesort, Quicksort i Radix sort. Mergesort dijeli liste na dva dijela, QuickSort dijeli liste na velike vrijednosti i male vrijednosti, dok Radixsort dijeli liste promatrajući jedan ključ u određenom periodu.

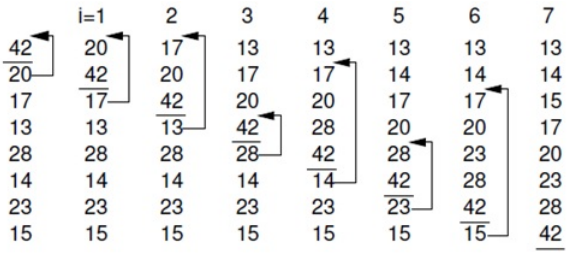
5.1 Notacija i terminologija

Kao ulazne vrijednosti u algoritmima sortiranja promatrat ćemo podatke koji su pohranjeni u nizove. Koristit ćemo metodu usporedi() koji vraća vrijednosti manje od 0, jednake 0, ili veće od 0, ovisno o relaciji između podataka unutar niza. Navedena metoda izvlači ključ iz niza podataka. Također ćemo koristiti metodu zamijeni(), koja mijenja pozicije dvaju podataka unutar niza. Neka imamo skup podataka $r_1, \ldots, r_n$ s vrijednostima ključeva $k_1, \ldots, k_n$. Problem sortiranja je način promjene pozicija unutar niza, tako da za podatke r_{s_1} ,r_{s_2},. . . ,r_{s_n}, odgovarajući ključevi zadovoljavaju svojstvo $k_{s_1} \leq k_{s_2} \leq \ldots \leq k_{s_n}$. Drugim riječima, zadatak problema sortiranja je da poreda podatke tako da su vrijednosti ključeva u nepadajućem poretku.

Pretpostavili smo da problem sortiranja dopušta dva ili više ulaznih podataka s istom vrijednosti. Kada su dopuštena dva ili više podataka istih vrijednosti, poredak ovisi o pojavljivanju unutar niza.

Za algoritam sortiranja kažemo da je stabilan ako ne mijenja poredak podataka (u odnosu na ostatak niza) s identičnim ključevima.

Prilikom usporedbe algoritama prirodno nam se čini isprogramirati i usporediti vrijeme potrebno za izvođenje. No, kao što smo već spomenuli takva usporedba ne mora nužno davati točnu usporedbu. Prilikom analiziranja algoritama sortiranja, promatrat ćemo broj usporedbi između ključeva. Ovakav način mjerenja je usko povezan s vremenom potrebnim za izvršavanje algoritma te samim tim algoritam s manjim brojem usporedbi će imati veću mogućnost programskog implementiranja. No, u pojedinim slučajevima, kada je broj ulaznih podataka velik, fizičko premještanje može zahtijevati određeni vremenski period. U tim slučajevima trebamo uzeti u obzir i metodu premještanja, tj. zamijeni().



5.2 Tri algoritma za sortiranje podataka

Ovo poglavlje predstavlja tri jednostavna algoritma za sortiranje. Lagana su za implementaciju i vidjet ćemo da su jako spora prilikom implementiranja velikog broja podataka. No, također ćemo spomenuti kada i za koje probleme imaju najbolju složenost.

5.2.1 Insertion sort

Ako želimo sortirati račune od zadnje dvije godine, prirodan način za sortiranje bilo bi da uzmemo dva računa, usporedimo datum i poredamo ih shodno datumu. Zatim uzmemo treći račun i usporedimo s prethodna dva. I tako za svaki idući. Ovaj intuitivni način sortiranja je upravo bio inspiracija za insertion sort.

Insertion sort prolazi kroz listu podataka. Svaki podatak ubacuje na točnu poziciju o odnosu na ostale podatke. Na idućem primjeru ćemo objasniti što je zapravo ključ određenog podatka.

Primjer 5.1 *Neka imamo niz [8 3 5 1 4 2]*

1. *vrijednost ključa kojeg prvog promatramo je 3. Znači promatramo vrijednost podatka na drugoj poziciji.*

2. *Ključ podatka na drugoj poziciji uspoređujemo s ključem na prvoj poziciji, tj. s 8. Kako je 8 veći od 3, to zamjenjujemo pozicije podataka s*

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=1; i<A.length; i++) // Insert i'th record
    for (int j=i; (j>0) && (A[j].compareTo(A[j-1])<0); j--)
      DSutil.swap(A, j, j-1);
}
```

vrijednosti ključa 8 i 3. Trenutni poredak je sada [3 8 5 1 4 2]

3. *Promatrajmo sada vrijednost ključa na trećoj poziciji, tj. 5. Vrijednost ključa na trećem mjestu je manja od vrijednosti ključa na drugoj poziciji, tj.* $8 > 5$*, stoga se zamjenjuju. Sada niz izgleda ovako [3 5 8 1 4 2]*

4. *Nastavljenjem prethodno opisanih koraka dobivamo sortiran niz [1 2 3 4 5 8].*

Ako želimo implementirati insert sort u Java programskom jeziku, za n podataka izgledalo bi ovako: Slika 5.2.1, primjer algoritma sortiranja U slučaju kada algoritam sortiranja razmatra i-ti podatak, s ključem X, pomiče ga za poziciju prije u nizu sve dok je X manji od vrijednosti ključa ispred, sve dok ne dođe do podatka sa ključem koji je manji ili jednak od vrijednosti X. Tijelo algoritma se sastoji od dvije for petlje. Vanjska petlja se izvodi $n - 1$ put, dok je unutarnja teža za analiziranje jer ovisi o vrijednostima ključeva između prvog i (i-1)-og mjesta

U najgorem slučaju mora provjeriti svaki ključ do nulte pozicije. Ovaj slučaj se događa kada je niz prvobitno obrnuto sortiran, tj. od najveće prema najmanjoj vrijednosti ključa. Stoga bi broj usporedbi bio: $\sum_{i=2}^{n} i \approx \frac{n^2}{2} = \theta(n^2)$

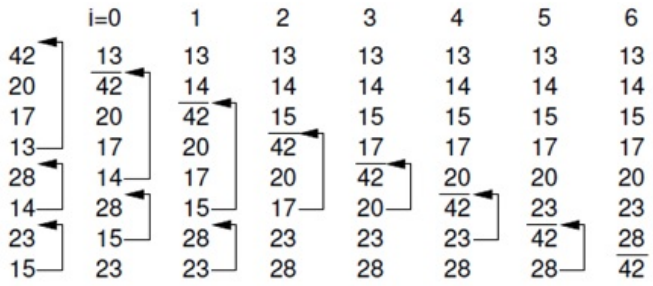
Ako promotrimo najbolji slučaj, kada je niz prvobitno sortiran od najmanje do najveće vrijednosti ključa, svaka for petlja bi izašla odmah i nijedna vrijednost ne bi bila pomaknuta na drugo mjesto. Ukupan broj usporedbi

bi bio $n-1$, što je broj ponavljanja vanjske for petlje. Stoga je složenost u najboljem slučaju $\Theta(n)$.

Iako je najbolji slučaj znatno manje složen od najgoreg slučaja, najgori slučaj je ipak mjerodavniji prilikom razmatranja vremenske potrošnje prilikom izvođenja programske implementacije ovog algoritma (u slučajevima kada mali broj podataka nije sortiran).

No, kolika je složenost u prosječnom slučaju? Ako želimo obraditi i-ti podatak, broj ponavljanja unutarnje for petlje ovisi koliko je niz nesortiran. Točnije, unutarnja for petlja se izvodi jedan put za svaki ključ (pozicije manje od i) koji je veći od ključa podatka i. Ako pogledamo prethodnu sliku 5.2.1, vrijednost 15 je obrađena 5 puta, jer je 5 vrijednosti s većim ključem od 15. Inverzijom nazivamo svako ponavljanje u kojem je broj vrijednosti koji se pojavljuju prije dane vrijednosti unutar niza, većih od dane vrijednosti. Jednostavnije rečeno, ako želimo naći broj inverzija, potrebno je pogledati svaku poziciju u permutaciji i izbrojati broj manjih brojeva od onih koji se nalaze s desne strane. Broj inverzija će odlučiti koliko će biti ukupno usporedbi i zamjena u nizu. Mi trebamo odrediti prosječan broj inverzija za vrijednosti na i-toj poziciji. Stoga, prosječan slučaj iznosi polovinu složenosti u najgorem slučaju, tj. $\frac{n^2}{4}$ složenost je $\Theta(n^2)$.

Stoga prosječni slučaj nije bolji od složenosti najgoreg slučaja u asimptotskoj složenosti. Prebrojavanjem usporedbi i zamjena dobivamo sličan rezultat. Svako ponavljanje (osim zadnjeg) unutarnje petlje, ima usporedbu i zamjenu. Stoga broj zamjena u čitavom algoritmu sortiranja je $n-1$, što je složenosti 0 u najboljem slučaju i $\Theta(n^2)$ u prosječnom i najgorem slučaju.



5.2.2 Bubble Sort

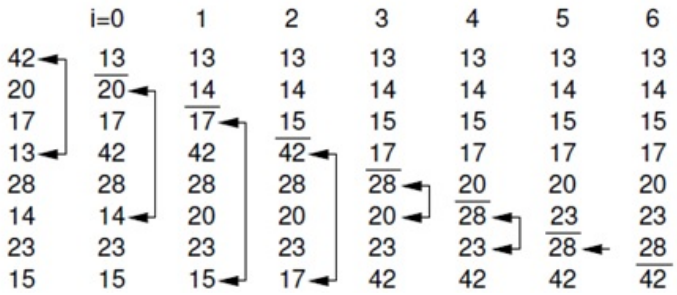
Sljedeći algoritam koji ćemo razmatrati naziva se Bubble Sort. Bubble Sort se često uči pri upoznavanju s računalnom znanosti. Navedeni algoritam je jako spor i nije lagan za razumijevanje kao Insertion Sort, nije lagan za interpretiranje u svakodnevnom životu i ima jako lošu složenost u najboljem slučaju. No, Bubble Sort može služiti kao inspiracija za kreiranje boljih algoritama sortiranja.

Bubble Sort se sastoji od jednostavne dvostruke for petlje. Prva iteracija unutarnje for petlje prolazi kroz niz podataka od početka do kraja, uspoređujući ključeve. Ako je donje pozicioniran ključ veće vrijednosti od više pozicioniranog susjeda, onda se dvije vrijednosti zamjenjuju. Jednom kada se naiđe na najmanju vrijednosti ovaj proces će pomaknuti na početak niza. Iduće ponavljanje kroz niz ponavlja proces. No, kako znamo da je najmanja vrijednost pomaknuta na početak, nema potrebe usporediti dvije vrijednosti na početku.

Pokažimo na primjeru kako bi izgledalo Bubble sortiranje. Zatim ćemo pogledati implementaciju opisanih koraka u Java programskom jeziku.

Određivanje broja usporedbi je lagano. Neovisno o poredanosti vrijednosti u nizu, broj usporedbi i-tog elementa koji se napravi kroz ponavljanje for petlje je uvijek i, iz čega dolazimo do složenosti koja iznosi: $\sum_{i=1}^{n} i \approx$

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=0; i<A.length-1; i++) // Bubble up i'th record
    for (int j=A.length-1; j>i; j--)
      if ((A[j].compareTo(A[j-1]) < 0))
        DSutil.swap(A, j, j-1);
}
```



$\frac{n^2}{2} = \theta(n^2)$

Vrijeme izvođenja Bubble Sort sortiranja je ugrubo jednako u najboljem, prosječnom i najgorom slučaju. Broj potrebnih zamjena ovisi koliko često je vrijednost manja od sljedeće u nizu. Možemo pretpostaviti da će se navedeni slučaj dogoditi barem kroz polovinu usporedbi u prosječnom slučaju. Slijedi da je $\Theta(n^2)$ očekivani broj zamjena. Stvarni broj zamjena kroz izvođenje Bubble Sort algoritma bit će jednak broju zamjena u Insertion Sortu.

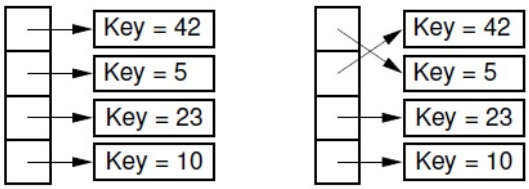
5.2.3 Selection Sort

Pokušajmo ponovo razmotriti problem sortiranja računa od prethodne godine. Drugi intuitivni pristup je da pogledamo kroz cijelu hrpu računa, dok ne pronađemo siječanj. Zatim ponovo prođemo kroz hrpu računa dok ne pronađemo veljaču itd. Navedena inspiracija upravo je princip rada Selection Sorta.

Princip rada Selection Sorta je sljedeći:

I-ti prolazak kroz Selection Sort odabire najmanji i-ti ključ u nizu, pos-

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=0; i<A.length-1; i++) { // Select i'th record
    int lowindex = i;                // Remember its index
    for (int j=A.length-1; j>i; j--) // Find the least value
      if (A[j].compareTo(A[lowindex]) < 0)
        lowindex = j;                // Put it in place
    DSutil.swap(A, i, lowindex);
  }
}
```



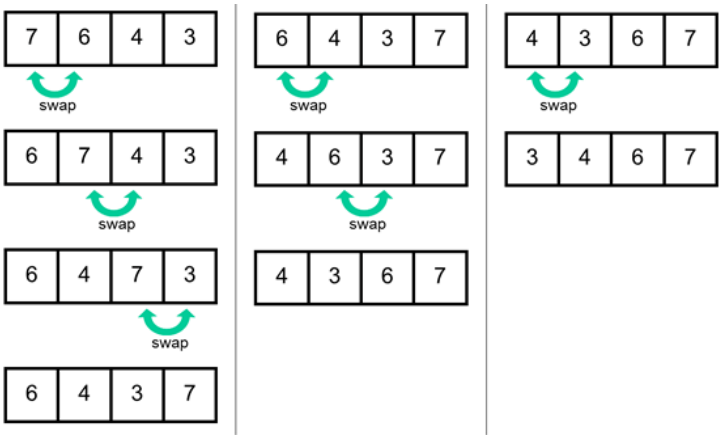
tavljajući ga na i-tu poziciju. Drugim riječima, Selection Sort prvo pronađe najmanji ključ u nesortiranoj listi, zatim drugi najmanji itd. Ovaj algoritam ima jako malo zamjena. No, za pronalazak sljedećeg najmanjeg ključa zahtjeva prolazak kroz cijeli niz nesortiranih podataka, ali samo jedna zamjena je eventualno potrebna. Stoga, ukupan broj zamjena je najviše $n - 1$.

Pokažimo implementaciju ovog algoritma u Java programskom jeziku: Selection Sort je sličan Bubble Sortu, razlika je što Bubble Sort zamjenjuje susjedne vrijednosti ovisno o relaciji između njihovih ključeva, dok Selection Sort zapamti poziciju najmanjeg u nizu i zamijeni kad na dođe do kraja niza. Stoga je broj usporedbi još uvijek $\Theta(n^2)$, ali je broj zamjena znatno manji nego u Bubble Sortu.

Kada koristiti Bubble sort?

Kad imamo podatke tipa riječi, Bubble Sort ima jako velik broj zamjena, dok Selection Sort samo jednu. Ovakve situacije, kao i mnoge druge koriste Selection Sort, jer je znatno bolji. Pogledajmo primjer zamjene ključeva podataka.

Postoji i drugi pristup u kojem je broj zamjena podataka mali, kojeg može



koristiti bilo koji algoritam sortiranja čak i prilikom velikog broja podataka.

Razlika je u tome što možemo umjesto vrijednosti spremati pokazivač na te vrijednosti. U ovoj implementaciji broj zamjena je potreban samo prilikom izmjena pokazivača, vrijednosti se same po sebi ne moraju mijenjati. Ova tehnika prikazana je na slici 5.2.3. Naravno, potrebna je dodatna memorija za spremanje pokazivača, ali ishod je brži nego kod primjene zamjena vrijednosti.

Primjer 5.2 *Uspoređujemo ključeve prva dva elementa, kako je* $6 < 7$*, to mijenjaju pozicije. Zatim promatramo ključeve 7 i 4, kako je 4 manje,to mijenjanju mjesta. Zatim se vrijednost ključa 7 uspoređuje sa 3, kako je broj 3 manji, to je mijenjaju za pozicije. Nakon što smo došli do kraja niza usporedbom prvog elementa, sada promatramo drugi element i uspoređujemo ga sa ostalima s desne strane. I tako za svaki element do kraja niza.*

5.3 Složenosti algoritama zamjene

Sumirajmo naša saznanja o tri algoritma u obliku tablice koja prikazuje broj potrebnih usporedbi i zamjena u najboljem, najgorem i prosječnom slučaju. Vrijeme potrebno za izvođenje za sva tri u prosječnom i najgorem slučaju

	Insertion sort	Bubble sort	Selection sort
Broj usporedbi			
Najbolji slučaj	Θ (n)	Θ (n^2)	Θ (n^2)
Prosječni slučaj	Θ (n^2)	Θ (n^2)	Θ (n^2)
Najgori slučaj	Θ (n^2)	Θ (n^2)	Θ (n^2)
Broj zamjena			
Najbolji slučaj	0	0	Θ (n)
Prosječni slučaj	Θ (n^2)	Θ (n^2)	Θ (n)
Najgori slučaj	Θ (n^2)	Θ (n^2)	Θ (n)

iznosi $\Theta(n^2)$. Prije prelaska na idući algoritam sortiranja, razmotrimo što prethodna tri algoritma čini sporima. Glavni problem leži u tome što se uspoređuju susjedne vrijednosti.

Algoritmi koji koriste zamjenu susjednih podataka često se nazivaju algoritmi zamjene. Složenost bilo kojeg algoritma zamjene može biti u najboljem slučaju jednaka ukupnom broju koraka potrebnih za podatke u nizu koje se moraju pomaknuti na ispravnu poziciju.

Koji je prosječan broj inverzija?

Pretpostavimo da niz L sadrži n podataka. Definiramo L_r niz u obrnutom poretku od L. L ima $\frac{n(n-1)}{2}$ različitih parova vrijednosti (vrijednosti ključeva elemenata koji se nalaze na nekoliko susjednih pozicija), svaki može biti potencijalna inverzija. Nekoliko ih mora biti ili inverzija u L ili inverzija u L_r. Stoga je ukupni broj inverzija u L i L_r ukupno jednak $\frac{n(n-1)}{2}$, po nizu. Stoga znamo da bilo koji algoritam sortiranja koji ograničava usporedbe na susjedne vrijednosti ima složenost barem $\frac{n(n-1)}{2} = \Omega(n^2)$ u prosječnom slučaju.

5.4 Shellsort

Sljedeći algoritam sortiranja koji ćemo razmatrati je Shellsort, imenovan po svom izumitelju D. L. Shell. Poznat je i po svom drugom nazivu "diminishing decrement sort", koji u prijevodu znači opadajući prirast.

5.4. Shellsort

Za razliku od Insertion i Selection Sorta, implementacije u svakodnevnom životu nema. Strategija Shellsorta je da napravi djelomično sortirani niz, kako bi Insertion Sort završio posao.

Prilikom ispravne implementacije, Shellsort će dati bolju složenost od $\Theta(n^2)$ u najgorem slučaju. Shellsort koristi proces koji formira bazu za puno drugih algoritama sortiranja.

Koraci su sljedeći:

- Podijeli listu u podliste,
- sortiraj podliste Insertion Sortom,
- zatim ponovno kombiniraj podliste.

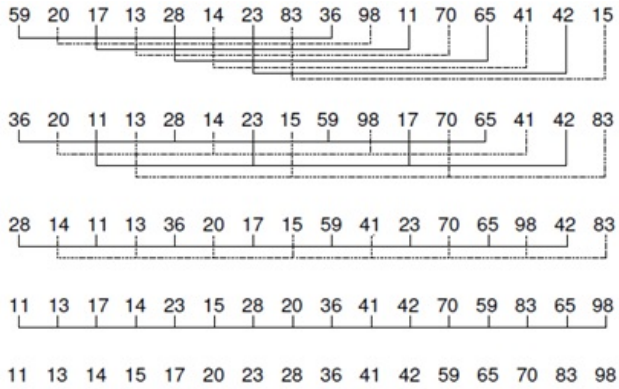
Prilikom iteracija, Shellsort razmatra podliste početne liste, tako da je svaki element u podlisti fiksiran svojom pozicijom. Naprimjer, pretpostavimo da trebamo sortirati n vrijednosti, gdje je n potencija broja 2. Jedina moguća implementacija Shellsorta je da početno "podijelimo" listu u n/2 podlista sa dva elementa. Pod podijelimo, misli se da gledamo jednu polovicu liste, pa zatim drugu.

Sljedeći prolazak Shellsorta, gleda manje i veće liste. Drugi prolaz bi imao $\frac{n}{2}$ lista s 4 elementa gdje bi pozicije bile udaljene za $\frac{n}{4}$. Treći prolazak bi imao dvije liste, jedna bi se sastojala od neparnih pozicija, druga od parnih. Ako je $n = 2^k$, gdje je k prirodan broj, tada bi broj prolazaka bio k.

Shellsort će raditi ispravno bez obzira na broj povećavanja, s tim da u zadnjem koraku broj povećavanja je 1.

Pogledajmo implementaciju u programskom jeziku Java:

Analiza Shellsorta je složena. Stoga bez dokaza navodimo da je složenost Shellsorta jednaka $\Theta(n^{1.5})$.




```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=A.length/2; i>2; i/=2) // For each increment
    for (int j=0; j<i; j++)         // Sort each sublist
      inssort2(A, j, i);
  inssort2(A, 0, 1);     // Could call regular inssort here
}

/** Modified Insertion Sort for varying increments */
static <E extends Comparable<? super E>>
void inssort2(E[] A, int start, int incr) {
  for (int i=start+incr; i<A.length; i+=incr)
    for (int j=i; (j>=incr)&&
                  (A[j].compareTo(A[j-incr])<0); j-=incr)
      DSutil.swap(A, j, j-incr);
}
```

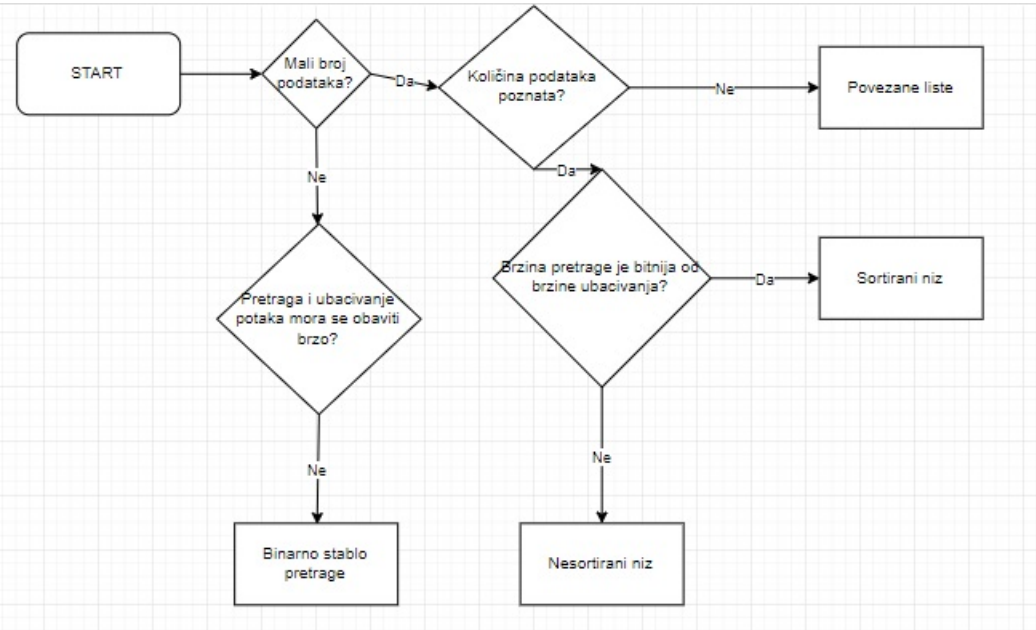
5.5 Kada koristiti koju strukturu podataka?

U ovom poglavlju ćemo obrazložiti koju strukturu podataka ili algoritam odabrati u određenoj situaciji. Unutar ovog poglavlja promatrat ćemo:

- Osnovnu svrhu struktura podataka poput niza, vezanih listi i stabla
- Posebne strukture podataka poput redova i stogova
- Sortiranja

5.5.1 Osnovna svrha struktura podataka

Ako trebamo spremati podatke poput naziva pjesama, listu korisnika ili popis inventara koristit ćemo osnovne strukture podataka poput redova, vezanih listi i stabla. Nazivaju se osnovnim strukturama podataka jer se koriste



uglavnom za spremanje i dobivanje podataka korištenjem ključeva. Koje su od ovih struktura podataka korisne za dani problem? Osnovne strukture podataka se koriste prilikom jednostavnijih algoritama koji uglavnom služe za održavanje baze podataka. Kako bi odlučili koju strukturu podataka koristiti pogledat ćemo sljedeći dijagram.

Dijagram odluke za osnovne strukture podataka

Brzina izvođenja i algoritmi

Brzina izvođenja osnovnih struktura podataka može se opisati ovako: nizovi i povezane liste su spori, stabla su brža. No, dok ne donesemo odluku u skladu sa dijagramom odluke za osnovne strukture podataka prikazanom na slici 5.5.1, ne možemo reći da ćemo koristiti određenu strukturu podataka. Naprimjer binarna stabla su jako spora za sortirane podatke, a balansirana stabla rješavaju problem sortiranih podataka, ali su jako teška za koristiti u programiranju. Najbrže strukture podataka dolaze sa svojim manama.

Naime, svake godine se povećava količina CPU-a i memorijske dostupnosti unutar računala. Ovo vodi nevjerovatnom napretku izvođenja u odnosu na računala proizvedena u prošlim godinama, stoga nam brzina neće biti bitnija od svrhe same strukture unutar algoritma. Dakle, osim ako struktura podataka nije znatno sporija, isplati ju se koristiti unutar programa. Kad govorimo o brzini manipuliranja objektima, Java programski jezik ima prednosti nad ostalim jezicima. Prednost je u tome što Java sprema samo reference na objekte, ne i same objekte. Stoga će se većina algoritama izvoditi brže u Javi nego bilo kojem programskom jeziku. No pri analiziranju algoritama to nije slučaj, obzirom da se objekti spremaju i potrebno je određeno vrijeme za prebacivanje objekata ovisno o veličini i samoj vrsti objekta. Naravno i drugi jezici poput C++ imaju mogućnost spremanja pokazivača na objekte umjesto samih objekata, ali je sinteksa samog jezika znatno kompliciranija za korištenje.

Liste

U dosta problema nizovi su prvi oblik strukture podataka koji trebamo razmotriti, ako radimo sa pohranom i manipuliranjem podataka. Nizovi su korisni kada:

- količina podataka je jako mala
- količina podataka se zna unaprijed

Ako imamo dovoljno memorije možemo zanemariti drugi uvjet. Prilikom ubacivanja elemenata, brzina je jako bitna, tada koristimo nesortiranu listu. Brisanje je uvijek sporo u listama, jer u prosjeku trebamo pomaknuti pola od ukupnog broja elemenata.

Povezane liste

Povezane liste uzimamo u obzir kada je količina podataka nepredvidiva ili ako je potrebno dodavati i izbacivati elemente često. Povezane liste zauzimaju memoriju prilikom dodavanja novog elementa, stoga se može dogoditi da ostanemo bez dostupne memorije. Ubacivanje elemenata je jako brzo u nesortiranoj listi. Pretraga i brisanje su spori (iako je pretraga brža nego u običnoj listi), stoga povezane liste koristimo kad imamo malu količinu podataka za obraditi. Povezane liste su kompleksne prilikom korištenja unutar programa, stoga se manje koriste nego liste, ali više nego stabla.

Binarna stabla

Binarna stabla su prva struktura podataka koju trebamo razmatrati kada se liste i povezane liste pokažu sporima. Stablo omogućuje brzinu $O(logn)$ prilikom ubacivanja, pretrage i brisanja. Također je jako brz pri određivanju najmanjeg i najvećeg elementa. Nebalansirana binarna stabla su jako lagana u primjeni unutar programa, znatno jednostavnija od balansiranih stabala, ali prilikom rada sa sortiranim podacima brzina se može smanjiti do O(n), što nije bolje od povezanih listi.

5.6 Složenije strukture podataka

Složenije strukture podataka poput stogova i redova se često koriste prilikom rada s programima unutar kojih korisnici aplikacije mogu dodavati podatke te se često koriste kao ispomoć za izvođenje određenog algoritma. Stogovi i redovi su apstraktni tipovi podataka koji se implementiraju pomoću osnovnih struktura podataka poput liste, povezane liste, itd. Ove apstraktne strukture podataka su jednostavne za korisnike aplikacije prilikom dodavanja

ili pristupa podacima.

Stogovi

Stogovi se koriste kada trebamo pristupiti samo zadnjem podatku ubačenom u bazu podataka. Stog se često implementira pomoću liste ili povezane liste. Implementacija pomoću liste je efikasna za učestalo ubacivanje podataka na kraj liste, također i za brisanje istog podatka. No, ako je količina podataka velika, liste se ne koriste. Ako stog sadrži veliku količinu podataka i količina se konstantno povećava, povezana lista je bolji izbor za implementaciju stoga. Povezane liste su korisne za brzo ubacivanje i brisanje elementa na početku.

Redovi

Redove koristimo kad želimo pristupiti samo prvom ubačenom elementu. Kao stogovi, redovi se mogu implementirati pomoću lista ili povezanih lista. Obje su efikasne. Implementacija pomoću liste zahtijeva dodatnu izradu programa prilikom rada sa zadnjim elementima unutar reda. Kao stogovi, odabir implementacije pomoću niza ili povezane liste odlučuje se na temelju količine podataka s kojom radimo. Kada znamo unaprijed količinu podataka koristit ćemo implementaciju pomoću niza, inače koristimo povezane liste.

Sortiranje

Kao i s odabirom strukture podataka tako i sa odabirom algoritma za sortiranje, prvo trebamo započeti sa sporijim sortiranjem kao Insertion sort. Počinjemo sa sporijim, jer je velika vjerojatnost da će sa trenutnom računalnom snagom algoritam Insertion sort završiti unutar razumnog vremena. Insertion sort je jako dobar za djelomično sortirane podatke sa složenosti operacije O(n). Ovo je slučaj kad je mali broj podataka dodan u novosortiranu listu.

Ako se pokaže da je Insertion sort prespor za dani problem, idući kandidat je Shellsort. Shellsort je lako implementirati unutar računalnog programa i procjenjuje se da je koristan prilikom rada na problemu pri veličini liste do 5000 podataka. Quicksort koristimo ako su podaci razvrstani na neki način, sa malim brojem nesortiranih podataka, također je sklon greškama prilikom implementacije. Stoga implementaciju algoritma u računalnom programu treba uzeti s oprezom.

Poglavlje 6

Primjeri implementacije algoritma

6.1 Primjer problema rekurzivnog sortiranja koristeći različite strukture podataka

Prva implementacija algoritma rekurzivnog sortiranja koju ćemo promatrati je red. Želimo konstruirati algoritam koji će za dani skup podataka vraćati skup podataka u poretku od najmanjeg do najvećeg elementa. Ideja rješenja problema je čuvati elemente u pozivu metode, kao privremene varijable, sve dok se struktura podataka ne isprazni. Kad se struktura podataka isprazni, vratit ćemo ih nazad u strukturu podataka u odgovarajućem poretku.

6.1.1 Red

Kreirat ćemo metodu **FrontToLast()** koja će za ulazne parametre dobiti red i duljinu reda koju promatra kroz varijablu qsize.

1. Ako je qsize nula, tj. ako je red prazan onda stajemo.

2. Inače premjestimo prvi element na zadnje mjesto, smanjimo vrijednost varijable qsize za jedan i rekurzivno pozovemo metodu **FrontToLast()** sa smanjenom vrijednosti qsize.

Metoda **pushInQueue()** za ulazne parametre dobiva red, privremenu vrijednost i duljinu reda unutar varijable qsize. Glavna funkcionalnost ove metode je sortiranje danog reda obzirom na privremenu vrijednost. Koraci su sljedeći:

1. Ako je red prazan, tj. vrijednost qsize jednaka nuli, onda dodamo privremenu vrijednost

2. Ako je privremena vrijednost manja od prve vrijednosti u redu, onda dodamo privremenu vrijednost u red i pozovemo metodu **FrontToLast()** sa trenutnim redom i veličinom reda qsize.

3. Inače, prvu vrijednost premjestimo na zadnje mjesto i rekurzivno pozovemo metodu **pushInQueue()** s ulaznim parametrima koji su red q, privremena vrijednost temp i qsize -1.

Glavna metoda, tj. metoda samog sortiranja u algoritmu je metoda **sortQueue()**. Metoda izbaci prvi element reda i promatra ga kao privremenu varijablu temp, zatim ju uspoređuje s ostalim vrijednostima rekurzivno pozivajući sebe, nakon toga vrati ju u niz ovisno o poretku pomoću metode **pushInQueue()**.

1. korak: Ako je red prazan, stani

2. korak: Pridruži varijabli temp vrijednost prvog elementa u redu, potom ga izbaci iz reda.

3. korak: Pozivamo metodu **sortQueue()** (tj. samu sebe), sa ulaznim parametrom reda bez početnog elementa, tj. rekurzivno samu sebe s duljinom n-1

4. korak: Poredamo elemente u nizu ovisno o privremenoj varijabli temp, tj. pozovemo metodu **pushInQueue()**

Pogledajmo implementaciju algoritma u programskom jeziku Java:

```
import java.util.*;
class Project1
{
    static void FrontToLast(Queue<Integer> q,int qsize)
    {
        if (qsize <= 0)
            return;
    q.add(q.peek());
    q.remove();
    FrontToLast(q, qsize - 1);
    }
    static void pushInQueue(Queue<Integer> q,int temp, int qsize)
    {
        if (q.isEmpty() || qsize == 0)
        {
            q.add(temp);
                return;
        }
        else if (temp <= q.peek())
        {
```

```
        q.add(temp);
            FrontToLast(q, qsize);
        }
        else
        {
            q.add(q.peek());
            q.remove();
            pushInQueue(q, temp, qsize - 1);
        }
    }
    static void sortQueue(Queue<Integer> q)
    {
        if (q.isEmpty())
            return;
        int temp = q.peek();
        q.remove();
        sortQueue(q);
        pushInQueue(q, temp, q.size());
    }
    public static void main(String[] args)
    {
        Queue<Integer> qu = new LinkedList<>();
        qu.add(10);
        qu.add(7);
        qu.add(16);
        qu.add(9);
        qu.add(20);
```

```
        qu.add(5);
        sortQueue(qu);
        while (!qu.isEmpty())
        {
            System.out.print(qu.peek() + " ");
            qu.remove();
    }
    }
}
```

Za ubacivanje svakog elementa na odgovorajuće mjesto potrebna je složenost O(n). Obzirom da takvih elemenata ima n, ukupna složenost je $O(n^2)$.

6.1.2 Stog

Pogledajmo sada primjenu algoritma rekurzivnog sortiranja na stogovima. Ideja samog algoritma je ista. Prilikom implementacije samog algoritma u Java programskom jeziku definiramo dvije metode: **sortedInsert()** i **sortStack()**. **SortedInsert()** metoda ubacuje dani ključ unutar sortiranog stoga prilikom sortiranja. Koraci su sljedeći:

1. Ako je red prazan ili je vrijednost ključa veća od vrijednosti prvog elementa, dodaj ključ na zadnje mjesto.

2. Inače pridruži privremenoj varijabli prvu vrijednost, te ju izbaci iz stoga, pozovi metodu **sortedInsert()** (tj. samu sebe) sa duljinom stoga $n - 1$ i ključem, te dodaj prvi element na kraj stoga.

Metoda **sortStack()** je glavna rekurzivna metoda za sortiranje stoga. Koraci su sljedeći:

1. Ako je stog prazan, stani

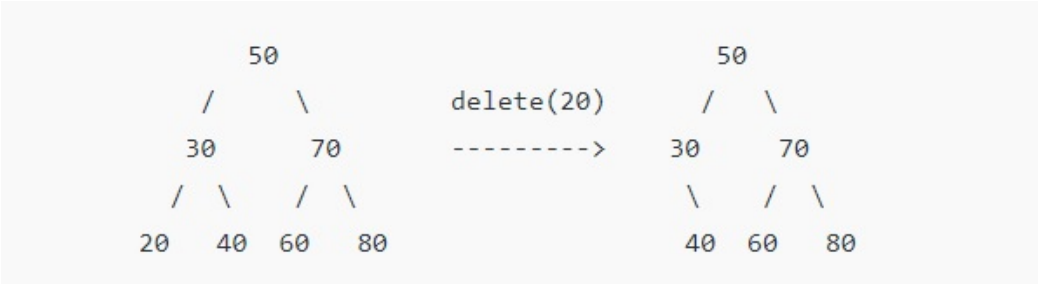
2. Inače, pridruži privremenoj varijabli temp vrijednost prvog elementa stoga, zatim ju izbaci. Pozovemo metodu **sortStack()** sa preostalim vrijednostima u stogu i metodu **sortedInsert()** koja sortira ostatak stoga obzirom na privremeni element temp.

Pogledajmo implementaciju algoritma rekurzivnog sortiranja pomoću stogova implementiranog u programskom jeziku Java:

```
import java.util.Arrays;
import java.util.List;
import java.util.Stack;

class Main
{
    public static void sortedInsert(Stack<Integer> stack, int ključ)
    {
        if (stack.isEmpty() || ključ > stack.peek())
        {
            stack.push(ključ);
            return;
        }
        int temp = stack.pop();
        sortedInsert(stack, ključ);
        stack.push(temp);
    }

    public static void sortStack(Stack<Integer> stack)
```

```
    {
        if (stack.isEmpty()) {
            return;
        }
        int temp = stack.pop();
        sortStack(stack);
        sortedInsert(stack, temp);
    }

    public static void main(String[] args)
    {
        List<Integer> list = Arrays.asList(5, 7, 9, 10, 16, 20);
        Stack<Integer> stack = new Stack<>();
        stack.addAll(list);
        System.out.println("Stog prije sortiranja: " + stack);
        sortStack(stack);
        System.out.println("Stog nakon sortiranja: " + stack);
    }
}
```

Složenost navedenog algoritma u Java implementaciji je $O(n^2)$ i potrebno je $O(n)$ memorije za pozivanje, gdje je n ukupan broj elemenata u stogu. Stoga je zaključak da odabir stogova ili redova nije utjecao znatno na složenost implementacije algoritma rekurzivnog sortiranja.



6.2 Primjer algoritma brisanja elementa

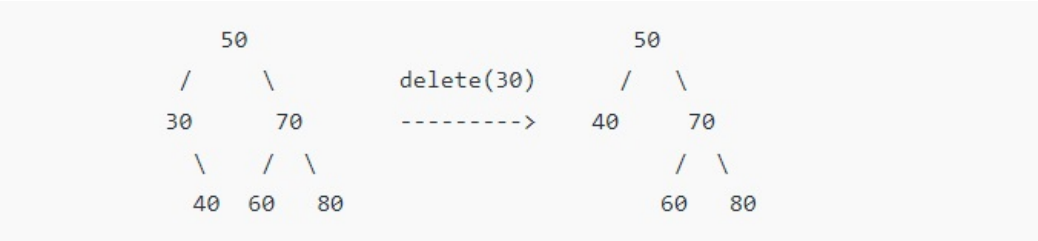
Pogledajmo sada problem brisanja elementa. Uzet ćemo dvije strukture podataka: binarno stablo pretrage i reda. Dakle, problem je jednostavan, izbrisati proizvoljni element iz strukture podataka. Pogledat ćemo implementaciju algoritma brisanja elementa iz strukture u Java programskom jeziku, objasniti korake i na kraju usporediti složenost.

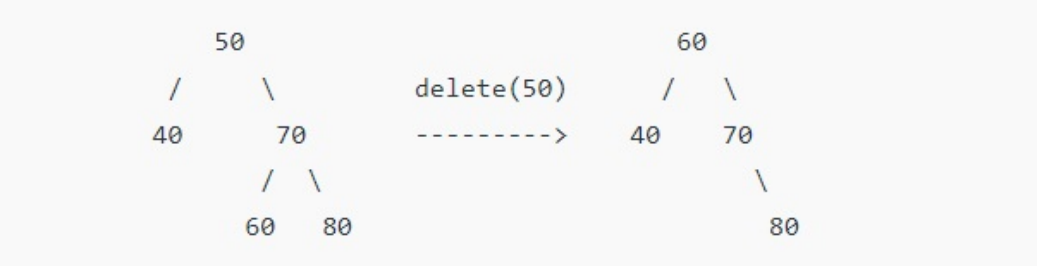
6.2.1 Binarno stablo pretrage

Problem brisanja elementa iz binarnog stabla dijelimo na tri slučaja ovisno o čvoru koji nosi vrijednost elementa:

- Čvor je list
- Čvor ima jedno dijete
- Čvor ima dvoje djece

Ako je čvor list, uklonimo samo taj čvor bez razmještanja ostalih čvorova. Ako čvor ima jedno dijete, vrijednost djeteta zamijenimo sa vrijednosti roditelj čvora i izbrišemo čvor gdje je prvobitno bila vrijednost djeteta. Ako čvor ima dva djeteta, usporedimo vrijednosti čvor djece i odaberemo onog s manjom vrijednosti kao zamjenu vrijednosti roditelj čvora. Dijete čiji je čvor





imao manju vrijednost, izbrišemo. Pogledajmo implementaciju u programskom jeziku Java:

```
class BinarnoStabloPretrage {
    /* Klasa koja sadrži lijevi
    i desni čvor te njihove
    vrijednosti*/
    class Čvor {
        int ključ;
        Čvor lijevi, desni;

        public Čvor(int item)
        {
            ključ = item;
            lijevi = desni = null;
        }
    }
```

```
// Korijen binarnog stabla pretrage
Čvor korijen;

// Konstruktor
BinarnoStabloPretrage() { korijen = null; }

// Metoda koja poziva brisanje čvora
void izbrišiKljuč(int ključ) { korijen = deleteRec(korijen, ključ); }

/*Rekurzivna metoda
brisanja čvora
 */
Čvor deleteRec(Čvor korijen, int ključ)
{
    /* Baza: stablo je prazno */
    if (korijen == null)
        return korijen;

    /* Inače, spuštamo se niz stablo*/
    if (ključ < korijen.ključ)
        korijen.lijevi = deleteRec(korijen.lijevi, ključ);
    else if (ključ > korijen.ključ)
        korijen.desni = deleteRec(korijen.desni, ključ);

    // Ako je vrijednost ključa
    // jednaka vrijednosti korijena,
    //onda će taj čvor biti izbrisan
```

```
        else {
            // Čvor sa samo jednim djetetom
            //ili čvor bez djece
            if (korijen.lijevi == null)
                return korijen.desni;
            else if (korijen.desni == null)
                return korijen.lijevi;

            // Čvor sa dvoje djece: S tim
            //da uzimamo dijete čvor
            //čija je vrijednost manja
            korijen.ključ = minValue(korijen.desni);

            // Brišemo čvor s manjom vrijednosti
            korijen.desni = deleteRec(korijen.desni, korijen.ključ);
        }

        return korijen;
    }

    int minValue(Čvor korijen)
    {
        int minv = korijen.ključ;
        while (korijen.lijevi != null)
        {
            minv = korijen.lijevi.ključ;
            korijen = korijen.lijevi;
```

```
        }
        return minv;
    }

    // Ova metoda poziva ubacivanje
    //čvorova rekurzivno
    void insert(int ključ) { korijen = insertRec(korijen, ključ); }

    // Rekurzivna funkcija za ubacivanje čvorova
    Čvor insertRec(Čvor korijen, int ključ)
    {

        /* Ako je čvor prazan
        dodaj novi čvor */
        if (korijen == null) {
            korijen = new Čvor(ključ);
            return korijen;
        }

        /* Inače, rekurzivno se spusti na dno stabla */
        if (ključ < korijen.ključ)
            korijen.lijevi = insertRec(korijen.lijevi, ključ);
        else if (ključ > korijen.ključ)
            korijen.desni = insertRec(korijen.desni, ključ);

        /* vrati nepromjenjeni pokazivač
        na čvor */
```

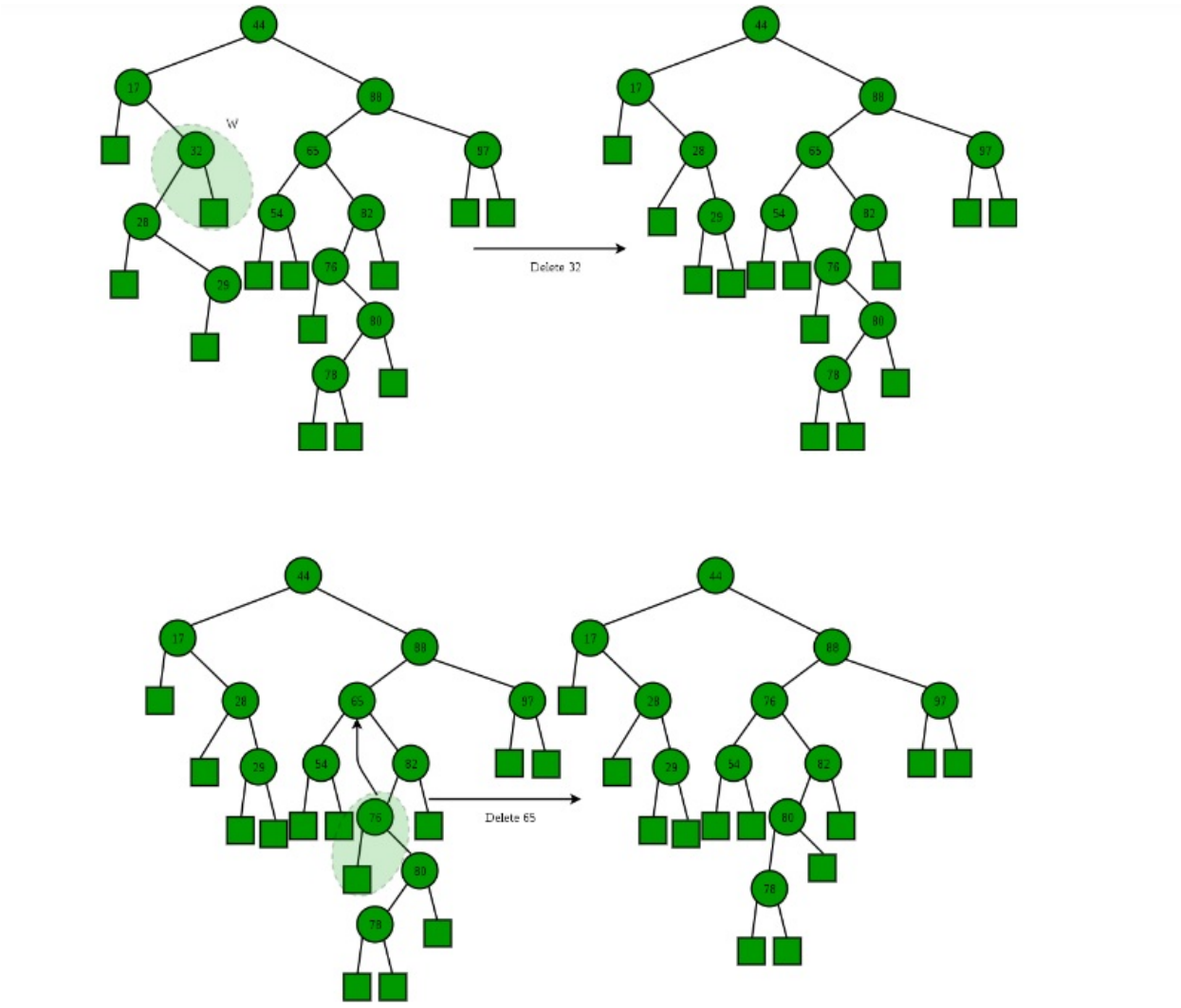
```
        return korijen;
    }


    void inorder() { inorderRec(korijen); }

    // Metoda je poredak
    void inorderRec(Čvor korijen)
    {
        if (korijen != null) {
            inorderRec(korijen.lijevi);
            System.out.print(korijen.ključ + " ");
            inorderRec(korijen.desni);
        }
    }


    public static void main(String[] args)
    {
        BinarySearchTree tree = new BinarySearchTree();

        /*Neka nam stablo pretrage izgleda ovako:
              50
           /     \
          30      70
         /  \    /  \
        20   40  60   80 */
```

```
stablo.insert(50);
stablo.insert(30);
stablo.insert(20);
stablo.insert(40);
stablo.insert(70);
stablo.insert(60);
stablo.insert(80);

System.out.println(
    "Inorder traversal of the given stablo");
stablo.inorder();

System.out.println("\nDelete 20");
stablo.izbrišiKljuč(20);
System.out.println(
    "Inorder traversal of the modified stablo");
stablo.inorder();

System.out.println("\nDelete 30");
stablo.izbrišiKljuč(30);
System.out.println(
    "Inorder traversal of the modified stablo");
stablo.inorder();

System.out.println("\nDelete 50");
stablo.izbrišiKljuč(50);
System.out.println(
```

```
            "Inorder traversal of the modified stablo");
        stablo.inorder();
    }
}
```

Složenost algoritma u najgorem slučaju je $O(h)$, gdje je h visina binarnog stabla pretrage. U najgorem slučaju trebali bi proći put od korijena do lista. Stoga je složenost brisanja elementa iz binarnog stabla pretrage $O(n)$. Program možemo optimizirati na način rekurzivne pretrage nasljednika čvor roditelja odabirom čvora djeteta koji ima manju vrijednost.

6.2. Primjer algoritma brisanja elementa



Slika 6.1: Ilustracija primjera pomoću slike:

6.2.2 Red

Pogledajmo sada implementaciju brisanja elementa iz reda. Element iz reda brišemo pomoću metode dequeue koju smo prethodno naveli i objasnili. Pogledajmo kako bi tu metodu implementirali u programskom jeziku Java:

```
    class node {
    int data;
    node* next;

    node(int val)
    {
        data = val;
        next = NULL;
    }
};
class Queue {
public:
    node* front;
    node* rear;

    Queue()
    {
        front = rear = NULL;
    }

    void enqueue(int val)
    {
        // Ako je red prazan
```

```
        if (rear == NULL) {
            // Kreiraj zadnji element
            rear = new node(val);
            rear->next = NULL;
            rear->data = val;

            // Prvi element će biti
            //zadnji samo ako je jedini
            //element u redu
            front = rear;
        }
        else {
            //Kreiramo privremeni node
            //sa vrijednosti val
            node* temp = new node(val);

            // Dodajemo temp kao zadnji element
            rear->next = temp;

            // Ažuriramo vrijednosti
            rear = temp;
        }
    }

    void dequeue()
    {
        // Usmjerimo pokazivač na
```

```
        // prvi element
        node* temp = front;
        // Ako je red prazan
        if (front == NULL) {
            System.out.println("Underflow");
            return;
        }
        else if (temp->next != NULL) {
            temp = temp->next;
            System.out.println("Element koji je izbrisan je : "
            front->data);
            free(front);
            front = temp;
        }
        //Ako red ima samo jedan element
        else {
            System.out.println("Element koji je izbrisan je: " ,front->data);
            free(front);
            front = NULL;
            rear = NULL;
        }
    }
};
int main()
{
    Queue q;
```

```
    // Ubacivanje elemenata
    q.enqueue(5);
    q.enqueue(7);

    // Brisanje elemenata
    q.dequeue();

    return 0;
}
```

Složenost navedenog algoritma je $O(1)$, tj. dobijemo značajno manju složenost od brisanja elementa iz binarnog stabla pretrage. Stoga ako radimo s podacima koji zahtjevaju dosta promjena predlaže se odabir redova umjesto binarnih stabala pretrage.

Poglavlje 7

Zaključak

Strukture podataka i algoritmi su jedna on najvažnijih tema u računalnoj znanosti. Poznavanje struktura podataka i njihov utjecaj na algoritme je osnova za napredak u programiranju. Odabir strukture podataka u algoritmima je važan radi:

1. Ograničene memorije računala.
2. Lakšeg i bržeg prikupljanja podataka.
3. Kako bi sistem bio organiziran, podaci moraju biti organizirani u specifične strukture podataka radi lakše manipulacije kroz algoritam.
4. Biranje pogrešnog algoritma i strukture podataka čini program sporim i neodrživim.

Izabrati ispravan algoritam je potrebno u:

1. umjetnosti, radi preglednosti, orginalnosti i efikasnosti pronalaska idealnih mjera prilikom crtanja.
2. znanosti, radi principa koji se koriste u dizajnu problema kako bi se određeni problem rješio kroz zadani period.

STRUKTURA PODATAKA	PRISTUP ELEMENTU UNUTAR STRUKTURE PODATAKA	PRETRAGA ELEMENTA UNUTAR STRUKTURE PODATAKA	UBACIVANJE ELEMENTA	BRISANJE ELEMENTA
NIZ	O(1)	O(1)	O(1)	O(1)
STOG	O(1)	O(1)	O(1)	O(1)
RED	O(1)	O(1)	O(1)	O(1)
VEZANA LISTA	O(1)	O(1)	O(1)	O(1)
BINARNO STABLO PRETRAGE	O(log n)	O(log n)	O(log n)	O(log n)

Slika 7.1: Složenost struktura podataka u najboljem slučaju:

STRUKTURA PODATAKA	PRISTUP ELEMENTU UNUTAR STRUKTURE PODATAKA	PRETRAGA ELEMENTA UNUTAR STRUKTURE PODATAKA	UBACIVANJE ELEMENTA	BRISANJE ELEMENTA
NIZ	O(1)	O(n)	O(n)	O(n)
STOG	O(n)	O(n)	O(1)	O(1)
RED	O(n)	O(n)	O(1)	O(1)
VEZANA LISTA	O(n)	O(n)	O(1)	O(1)
BINARNO STABLO PRETRAGE	O(n)	O(n)	O(n)	O(n)

Slika 7.2: Složenost struktura podataka u najgorem slučaju:

Sumirajmo sve do sada obrađeno za sve strukture podataka i osnovne algoritme za pristup, pretragu, ubacivanje elemenata i brisanje elemenata:

Svaki programski jezik ima sučelje sa osnovnim operacijama koje se izvode nad određenom strukturom podataka. Svaka od metoda ima svoju složenost, memoriju koju zauzima za određeni broj ulaznih podataka. Izbor strukture podataka unaprijed definira efikasnost, te mogućnosti koje možemo koristiti

STRUKTURA PODATAKA	PRISTUP ELEMENTU UNUTAR STRUKTURE PODATAKA	PRETRAGA ELEMENTA UNUTAR STRUKTURE PODATAKA	UBACIVANJE ELEMENTA	BRISANJE ELEMENTA
NIZ	O(1)	O(n)	O(n)	O(n)
STOG	O(n)	O(n)	O(1)	O(1)
RED	O(n)	O(n)	O(1)	O(1)
VEZANA LISTA	O(n)	O(n)	O(1)	O(1)
BINARNO STABLO PRETRAGE	O(log n)	O(log n)	O(log n)	O(log n)

Slika 7.3: Složenost struktura podataka u prosječnom slučaju

u algoritmima. Potrebno je poznavati strukturu podataka prije korištenja u algoritmima. Također je potrebno znati specifikaciju algoritma, pravila, operacije koje se koriste prije odlučivanja implementiranja struktura podataka i algoritma zajedno, kako bi se u potpunosti iskoristile mogućnosti koje nude.

Mrežne postavke, software developing i sve ostale grane programiranja usko su povezane s algoritmima i strukturama podataka.

Literatura

[1] Bullinaria John, Data Structures and Algorithms, School of Computer Science, University of Birmingham, UK, version of 27 March 2019

[2] Klaričić Bakula M., Braić S , Uvod u matematiku, skripta PMF-a, Split 2008.

[3] Perić Jurica, Složenost algoritama, skripta PMF-a, Split 2022.

[4] Rodrigues Martins Leonardo , Algorithms and How to Choose the Right Data Structure, https://blog.bitsrc.io/how-the-choice-of-data-structure-impacts-your-code-220d06c4ab96

[5] Shaffer A. Clifford , Data Structures and Algorithm Analysis, Edition 3.2.0.10, dated March 28, 2013.

[6] Singh Chandan , Search Algorithms in Java, https://csinva.io/notes/cs/data$_s$*tructures.html*

[7] Introduction to Data Structures and Algorithms, https://www.geeksforgeeks.org/introduction-to-data-

structures/

[8] Mingzhu Qian and Xiaobao Wang, Queue and Stack Sorting Algorithm Optimization and Performance Analysis, Huanggang Normal University No. 146, Xinggang second Road, Huanggang District, Hubei provincec, China