Sveučilište u Zagrebu
Prirodoslovno Matematički Fakultet
Matematički odsjek

Robert Manger

STRUKTURE PODATAKA I ALGORITMI

Nastavni materijal

Četvrto izdanje
Zagreb, listopad 2013.

Sadržaj

Predgovor

Ovaj nastavni materijal sadržava predavanja iz predmeta "Strukture podataka i algoritmi". Riječ je o predmetu koji se predaje studentima matematike na Matematičkom odsjeku Prirodoslovno matematičkog fakulteta Sveučilišta u Zagrebu.

Cilj predmeta, pa tako i ovog teksta, je stjecanje znanja o apstraktnim tipovima podataka te o strukturama podataka koje služe za njihovu implementaciju. Daljnji cilj je stjecanje znanja o osnovnim tehnikama za oblikovanje i analizu algoritama.

Svrha predmeta je osposobljavanje studenata da bolje programiraju. Naime, znanja o strukturama podataka i algoritmima omogućuju nam korištenje provjerenih rješenja za standardne programerske probleme. Usvajanjem tih znanja također usvajamo standardnu terminologiju kojom se lakše sporazumijevamo kad govorimo o programerskim rješenjima. Osim toga, gradivo iz ovog predmete predstavlja temelj za cijelo računarstvo te je preduvjet za bavljenje naprednijim računarskim disciplinama.

Sadržaj ovog nastavnog materijala podudara se u većoj ili manjoj mjeri sa sadržajem odgovarajućih udžbenika s engleskog govornog područja [1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22]. U skladu sa svojim naslovom, materijal u podjednakoj mjeri govori o strukturama podataka kao i o algoritmima. Oba pojma isprepliću se kroz cijeli tekst. Ipak, prvi dio teksta stavlja malo jači naglasak na strukture podataka promatrajući ih u širem kontekstu apstraktnih tipova podataka. Drugi dio teksta više se koncentrira na algoritme te se također bavi općenitim metodama za oblikovanje algoritama.

Nastavni materijal sastoji se od šest poglavlja, svako od njih podijeljeno je na odjeljke i pododjeljke, a uskoro će biti nadopunjeno i zadacima za vježbu. Poglavlje 1 sadržava uvod: tu se definiraju osnovni pojmovi kao što su tip podataka, struktura podataka, algoritam, apstraktni tip podataka, implementacija apstraktnog tipa. Također se opisuju načini grafičkog ili tekstualnog dokumentiranja struktura podataka ili algoritama te načini njihova analiziranja.

Poglavlje 2 obrađuje "linearne" apstraktne tipove kao što su lista, stog ili red te pripadne implementacije zasnovane na poljima ili vezanim listama. Također se daju brojni primjeri primjene tih apstraktnih tipova odnosno struktura koje ih implementiraju. Poglavlje 3 na sličan način obrađuje "hijerarhijske" apstraktne tipove kao što su uređena stabla ili binarna stabla.

Poglavlje 4 posvećeno je apstraktnim tipovima koji odgovaraju matematičkom pojmu skupa, uključujući rječnik, prioritetni red, preslikavanje i binarnu relaciju. Opet se daju primjeri primjene. Uvode se važne strukture podataka kao što su binarno stablo traženja, *hash* tablica ili hrpa.

Poglavlje 5 bavi se problemom sortiranja podataka i može se shvatiti kao studijski primjer gdje se primjenjuju znanja iz prethodnih poglavlja. Pokazuje se da se jedan te isti problem sortiranja može riješiti raznim algoritmima. Ti algoritmi razlikuju se

po svojim osobinama, a neki od njih na zanimljiv način koriste strukture podataka iz poglavlja 4.

Zadnje poglavlje 6 posvećeno je općenitim metodama (strategijama) koje mogu poslužiti za oblikovanje složenijih algoritama. Dakle, algoritmi se klasificiraju na osnovi načina svojeg rada bez obzira na vrstu problema koje rješavaju. Osim ponovnog spominjanja nekih algoritama iz prethodnih poglavlja, opisuje se i nekoliko dodatnih algoritama.

Gradivo o strukturama podataka i algoritmima može se izložiti na razne načine, što se zorno vidi usporedbom udžbenika [1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22]. U ovom tekstu koristi se pristup koji je naprimjereniji ciljanoj grupi studenata. Zaista, kao jezik za zapisivanje algoritama odabran je C, i to zato što su ga studenti već savladali kao svoj prvi programski jezik te zato što će im on tijekom studija i dalje služiti kao polazište za naprednije jezike poput C++, C# ili Jave.

C je pogodan i zbog toga što se u njemu mogu opisati sve pojedinosti programa na razini koja je bliska arhitekturi računala. Noviji jezici poput Python-a manje su pogodni za našu svrhu jer se služe višim razinama apstrakcije, skrivajući pritom baš one detalje o kojima bi ovdje htjeli raspravljati.

U cijelom tekstu namjerno se izbjegava objektni pristup programiranju. Umjesto pojma klase odnosno objekta koristi se sličan no labaviji pojam apstraktnog tipa podataka koji se lakše uklapa u mogućnosti klasičnog C-a. To je u skladu sa činjenicom da studenti kojima je tekst namijenjen nisu još učili objektno programiranje. Naime, nastavni program Matematičkog odsjeka uvodi objektni pristup relativno kasno, smatrajući da on zahtijeva određenu dozu zrelosti i iskustva u klasičnom programiranju. Ovaj nastavni materijal zapravo funkcionira kao priprema i motivacija za objektno programiranje.

U oblikovanju teksta pojavilo se i pitanje izbora imena za tipove podataka, konstante, varijable i funkcije u C-u. Trebalo je odlučiti hoće li se koristiti hrvatski ili engleski nazivi. S obzirom da se imena uklapaju u programe te funkcioniraju kao proširenje programskog jezika, na kraju je bila odabrana engleska varijanta. To se možda neće svima svidjeti, no zapravo je prirodnije, a korisno je i zbog usvajanja standardne engleske terminologije.

Za praćenje ovog nastavnog materijala neophodno je dobro poznavanje programskog jezika C. Naime, naredbe u C-u pojavljuju se u svim dijelovima teksta bez posebnih objašnjenja. Osobito su važni dijelovi jezika koji se odnose na pokazivače (pointere) i dinamičko dodjeljivanje memorije. Čitatelju koji ne zna programirati u C-u preporučuje se da samostalno prouči odgovarajuću literaturu, primjerice [9, 10, 16].

Poglavlje 1

UVOD

Ovo uvodno poglavlje neophodno je za razumijevanje terminologije i notacije koja se koristi u ostatku nastavnog materijala. Također, ono nam bolje objašnjava ciljeve i svrhu cijelog nastavnog materijala. Sastoji se od tri odjeljka. Prvi od njih definira osnovne pojmove, drugi podrobnije govori o strukturama podataka, a treći daje dodatne napomene o algoritmima.

1.1 Osnovni pojmovi

U ovom odjeljku najprije definiramo pojam strukture podataka odnosno pojam algoritma. Zatim uvodimo još nekoliko srodnih pojmova: tip podataka, apstraktni tip, implementacija. Na kraju raspravljamo o tome kakve veze svi ti pojmovi imaju s našom željom da bolje programiramo.

1.1.1 Struktura podataka, algoritam

Svaki programer složit će se s tvrdnjom da pri razvoju računalnog programa moramo voditi brigu o dvije stvari: o strukturama podataka i o algortimima. Strukture podataka čine "statički" aspekt programa – to je ono sa čime se radi. Algoritmi predstavljaju "dinamički" aspekt – to je ono što se radi.

Računalni program mogli bismo usporediti s kuharskim receptom: kao što recept na svom početku sadrži popis sastojaka (ulje, luk, brašno, meso, ...) tako i program mora započeti s definicijama podataka. Kao što recept mora opisivati postupak pripreme jela (sjeckanje, pirjanje, miješanje, ...), tako i program mora imati izvršne naredbe koje opisuju algoritam. Rezultat primjene kuharskog recepta je jelo dobiveno od polaznih sastojaka primjenom zadanog postupka. Rezultat izvršavanja programa su izlazni podaci dobiveni transformacijom ulaznih podataka primjenom zadanog algoritma.

Kao što se dobro kuhanje odlikuje izborom kvalitetnih prehrambenih sastojaka uz ispravnu primjenu kulinarskih postupaka, tako se i dobro programiranje u podjednakoj mjeri sastoji od razvoja pogodnih struktura podataka kao i razvoja pogodnih algoritama. Programeri to često zaboravljaju, previše se koncentriraju na algoritme, makar primjena dobrih struktura podataka može jednako tako utjecati na kvalitetu cjelokupnog rješenja.

U nastavku slijede preciznije definicije dvaju ključnih pojmova koje susrećemo u programiranju.

Struktura podataka ... skupina varijabli u nekom programu zajedno s vezama između tih varijabli. Stvorena je s namjerom da omogući pohranjivanje određenih podataka te efikasno izvršavanje određenih operacija s tim podacima (upisivanje, promjena, čitanje, traženje po nekom kriteriju, ...).

Algoritam ... konačan niz naredbi, od kojih svaka ima jasno značenje i može se izvršiti u konačnom vremenu. Izvršavanjem tih naredbi zadani ulazni podaci pretvaraju se u izlazne podatke (rezultate). Pojedine naredbe mogu se izvršavati uvjetno, ali u tom slučaju same naredbe moraju opisati uvjet izvršavanja. Također, iste naredbe mogu se izvršiti više puta, pod pretpostavkom da same naredbe ukazuju na ponavljanje. Ipak, zahtijevamo da za bilo koje vrijednosti ulaznih podataka algoritam završava nakon konačnog broja ponavljanja.

Strukture podataka i algoritmi nalaze se u nerazlučivom odnosu: nemoguće je govoriti o jednom a da se ne spomene drugo. U ovom nastavnom materijalu proučavat ćemo baš taj odnos: promatrat ćemo kako odabrana struktura podataka utječe na algoritme za rad s njom, te kako odabrani algoritam sugerira pogodnu strukturu za prikaz svojih podataka. Na taj način upoznat ćemo se s nizom važnih ideja koje čine osnove dobrog programiranja, a također i osnove računarstva u cjelini.

1.1.2 Tip podataka, apstraktni tip, implementacija

Uz "strukture podataka" i "algoritme", ovaj nastavni materijal koristi još nekoliko naizgled sličnih pojmova. Oni su nam potrebni da bi izrazili međusobnu ovisnost između podataka i operacija koje se nad njima trebaju obavljati, odnosno ovisnost između operacija i algoritama koji ih realiziraju. Slijede definicije tih dodatnih pojmova.

Tip podataka ... skup vrijednosti koje neki podatak može poprimiti. Primjerice, podatak tipa `int` u nekom C programu može imati samo vrijednosti iz skupa cijelih brojeva prikazivih u računalu.

Apstraktni tip podataka (a.t.p.) ... zadaje se navođenjem jednog ili više tipova podataka, te jedne ili više operacija (funkcija). Operandi i rezultati navedenih operacija su podaci navedenih tipova. Među tipovima postoji jedan istaknuti po kojem cijeli apstraktni tip podataka dobiva ime.

Implementacija apstraktnog tipa podataka ... konkretna realizacija dotičnog apstraktnog tipa podataka u nekom programu. Sastoji se od definicije za strukturu podataka (kojom se prikazuju podaci iz apstraktnog tipa podataka) te od potprograma (kojima se operacije iz apstraktnog tipa podataka ostvaruju pomoću odabranih algoritama). Za isti apstraktni tip podataka obično se može smisliti više različitih implementacija - one se razlikuju po tome što koriste različite strukture za prikaz podataka te različite algoritme za izvršavanje operacija.

Kao konkretni (no vrlo jednostavni) primjer za apstraktni tip podataka u nastavku definiramo apstraktni tip koji odgovara matematičkom pojmu kompleksnih brojeva i njihovom specifičnom načinu zbrajanja i množenja.

Apstraktni tip podataka Complex

`scalar` ... bilo koji tip za koji su definirane operacije zbrajanja i množenja.

`Complex` ... podaci ovog tipa su uređeni parovi podataka tipa `scalar`.

`CoAdd(z1,z2,&z3)` ... za zadane `z1,z2` tipa `Complex` računa se njihov zbroj `z3`, također tipa `Complex`. Dakle za `z1` oblika `(x1,y1)`, `z2` oblika `(x2,y2)`, dobiva se `z3` oblika `(x3,y3)`, tako da bude `x3=x1+x2`, `y3=y1+y2`.

`CoMult(z1,z2,&z3)` ... za zadane `z1,z2` tipa `Complex` računa se njihov umnožak `z3`, također tipa `Complex`. Dakle za `z1` oblika `(x1,y1)`, `z2` oblika `(x2,y2)`, dobiva se `z3` oblika `(x3,y3)`, takav da je `x3=x1*x2-y1*y2`, `y3=x1*y2+y1*x2`.

Vidimo da se apstraktni tip zadaje kao popis gdje su najprije navedeni potrebni tipovi podataka, a zatim osnovne operacije koje mislimo obavljati nad tim podacima. U konkretnoj implementaciji svaki navedeni tip trebao bi se opisati kao tip u C-u s istim nazivom. Također, svaka operacija trebala bi se realizirati kao funkcija u C-u s istim nazivom, istovrsnim argumentima i istovrsnom povratnom vrijednošću. Primijetimo da je u gornjem popisu za svaku operaciju naveden način njezinog pozivanja, a ne prototip za njezino definiranje. Ako operacija mijenja neku varijablu, tada se ta varijabla u pozivu prenosi preko adrese, što je označeno znakom `&`.

Primijetimo da je za apstraktni tip podataka `Complex` važno prisustvo operacija `CoAdd( )` i `CoMult( )`. Bez tih operacija radilo bi se o običnom tipu, i tada kompleksne brojeve ne bismo mogli razlikovati od uređenih parova skalara. Dodatna dimenzija "apstraktnosti" sastoji se u tome što nismo do kraja odredili što je tip `scalar`. Ako za `scalar` odaberemo `float`, tada imamo posla sa standardnim kompleksnim brojevima. Ako `scalar` poistovijetimo s `int`, tada koristimo posebne kompleksne brojeve čiji realni i imaginarni dijelovi su cijeli. Sa stanovišta struktura podataka i algoritama izbor tipa `scalar` zapravo je nebitan. Važan nam je odnos među varijablama a ne njihov sadržaj.

Struktura podataka pogodna za prikaz kompleksnog broja mogla bi se definirati sljedećim složenim tipom.

```
typedef struct {
   scalar re;
   scalar im;
} Complex;
```

Implementacija apstraktnog tipa podataka `Complex` mogla bi se sastojati od naredbe kojom se tip `scalar` poistovjećuje s `int` ili `float`, prethodne definicije tipa `Complex` te od sljedećih funkcija:

```
void CoAdd (Complex z1, Complex z2, Complex *z3p) {
   *z3p.re = z1.re + z2.re;
   *z3p.im = z1.im + z2.im;
   return;
}
```

```
void CoMult (Complex z1, Complex z2, Complex *z3p){
   *z3p.re = z1.re * z2.re - z1.im * z2.im;
   *z3p.im = z1.re * z2.im + z1.im * z2.re;
   return;
}
```

Opisana implementacija apstraktnog tipa `Complex` nije naravno jedina moguća implementacija. Donekle drukčije rješenje dobili bismo kad bi kompleksni broj umjesto kao `struct` prikazali kao polje skalara duljine 2. Makar je to prilično nebitna razlika, ona bi zahtijevala da se funkcije `CoAdd` i `CoMult` zapišu na drukčiji način.

Daljnje (mnogo živopisnije) primjere apstraktnih tipova podataka i njihovih implementacija susrest ćemo u idućim poglavljima. Naime, svaki od odjeljaka 2.1–4.4 obrađuje jedan apstraktni tip podataka. Opisuju se svojstva tog apstraktnog tipa, nabrajaju njegove primjene, promatraju razne njegove implementacije te uočavaju prednosti i mane tih implementacija. Dakle, čitanjem idućih poglavlja upoznat ćemo mnogo apstraktnih tipova te još više struktura podataka i algoritama.

U složenijim problemima može se pojaviti potreba da radimo s više apstraktnih tipova u isto vrijeme. Pritom bi oni mogli imati slične operacije. Da ne bi došlo do nedoumice kojem apstraktnom tipu pripada koja operacija, uvest ćemo sljedeću konvenciju: imena svih operacija iz istog apstraktnog tipa moraju početi s dvoslovčanim prefiksom koji jednoznačno određuje taj apstraktni tip. Ovu konvenciju već smo primijenili kod apstraktnog tipa `Complex` gdje su imena za obje funkcije imale prefix `Co`.

1.1.3 Programiranje postepenim profinjavanjem

Znanje iz ovog nastavnog materijala važno je zato jer nam omogućuje da bolje programiramo. Naime, apstraktni tipovi, strukture podataka i algoritmi predstavljaju vrlo koristan skup alata koji nam pomažu kod rješavanja složenijih programerskih problema.

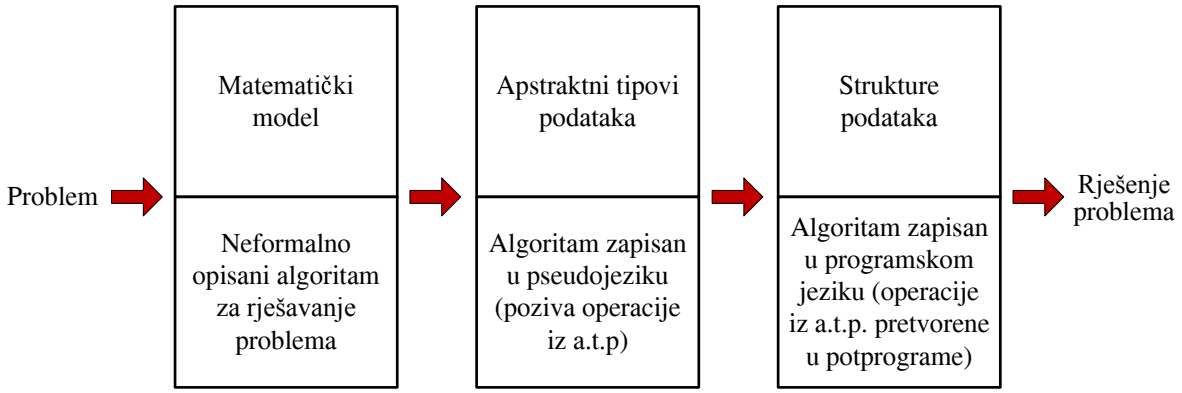
Zaista, kad god se suočimo sa složenijim problemom, tada do rješenja obično ne možemo doći na izravan način. To jest, nije moguće sjesti za računalo i odmah napisati program u C-u. Umjesto toga služimo se metodom postepenog profinjavanja, gdje se do rješenja dolazi postepeno u nizu koraka. Početni koraci definiraju rješenje na neformalan način služeći se nekim općenitim matematičkim modelom. Daljnji koraci razrađuju to rješenje dodajući mu konkretne detalje, sve dok u zadnjem koraku ne dođemo do opisa na razini detaljnosti kakva se traži u programskom jeziku.

Slika 1.1 prikazuje jednu varijantu metode postepenog profinjavanja. Riječ je o varijanti koja se oslanja na apstraktne tipove podataka, a u njoj se postupak rješavanja problema dijeli na tri koraka.

- U prvom koraku sastavlja se matematički model problema, a rješenje se opisuje neformalnim algoritmom koji djeluje u okvirima tog modela.

- U drugom koraku uočavaju se osnovne operacije nad matematičkim objektima, pa se ti objekti zajedno s s operacijama pretvaraju u apstraktne tipove. Algoritam se zapisuje u pseudojeziku kao niz poziva operacija iz apstraktnih tipova.

- U trećem koraku odabiremo implementacijju za svaki od korištenih apstraktnih tipova. Na taj način dolazimo do struktura podataka s jedne strane te do funkcija koje implementiraju operacije iz apstraktnih tipova s druge strane. Algoritam se zapisuje u programskom jeziku kao "glavni" program koji poziva spomenute funkcije kao potprograme.



Slika 1.1: Postupak rješavanja problema metodom postepenog profinjavanja.

Iz slike 1.1 jasno se vidi uloga uloga apstraktnih tipova, struktura podataka i algoritama u postupku rješavanja problema. Također, vidi se da se programiranje u podjednakoj mjeri sastoji od razvijanja struktura podataka kao i od razvijanja algoritama.

Da bismo ilustrirali našu varijantu metode postepenog profinjavanja poslužit ćemo se sljedećim primjerom problema. Zamislimo da na fakultetu u istom danu treba održati veći broj ispita. Neka svaki od njih traje jedan sat. Dvorana i nastavnika ima dovoljno. No poteškoća je u tome što su neki ispiti u koliziji: dakle za njih su prijavljeni dijelom isti studenti, pa se oni ne mogu održati istovremeno. Treba pronaći raspored održavanja ispita takav da ispiti u koliziji ne budu u isto vrijeme, a da ukupno vrijeme održavanja bude što kraće.

U nastavku pratimo rješavanje problema rasporeda ispita metodom postepenog profinjavanja.

- U prvom koraku sam problem opisujemo matematičkim modelom obojenog grafa. Riječ je o neusmjerenom grafu gdje svaki vrh odgovara jednom ispitu. Dva vrha su susjedna (povezani su bridom) ako i samo ako su dotični ispiti u koliziji. Vrhovi su obojeni bojama koje označavaju termine (sate) održavanja ispita. Na taj način iz boja vrhova čita se raspored. Da bi taj raspored bio dopustiv, susjedni vrhovi (dakle oni povezani bridom) moraju biti u različitim bojama. Da bi ukupno vrijeme održavanja ispita bilo što kraće, broj upotrebljenih boja mora biti što manji.

- Kao algoritam za rješavanje problema možemo koristiti sljedeći algoritam bojenja grafa koji nastoji potrošiti što manje boja (makar ne mora nužno postići najmanji mogući broj). Algoritam radi u iteracijama. U jednoj iteraciji bira se jedan neobojani vrh i gledaju se boje njegovih susjeda; zatim se izabrani vrh oboji s

jednom od već upotrebljenih boja koju ne koristi ni jedan od susjeda; ako nema takve boje tada se izabrani vrh oboji novom bojom, čime smo broj upotrebljenih boja povećali za jedan. Iteracije se ponavljaju dok god ima neobojenih vrhova.

- U drugom koraku naš matematički model, dakle obojeni graf, pretvaramo u apstraktni tip. Taj apstraktni tip treba sadržavati tip podataka za prikaz samog grafa te osnovne operacije za rukovanje grafom. Očito će nam trebati operacije: ubacivanja novog vrha, spajanja dvaju postojećih vrhova bridom, pridruživanja zadane boje zadanom vrhu, pronalaženja susjeda zadanog vrha, itd.

- Prije opisani algoritam bojenja grafa sada je potrebno zapisati u terminima osnovnih operacija nad apstraktnim tipom. Primjerice, jedna iteracija gdje se jednom neobojenom vrhu određuje boja koristit će operaciju pronalaženja susjeda vrha, niz operacija čitanja boje vrha te operaciju pridruživanja boje vrhu.

- U trećem koraku razvijamo implementaciju za naš apstraktni tip obojenog grafa. Na primjer, graf se može prikazati strukturom koja se sastoji od vektora boja i matrice susjedstva. Pritom i-ta komponenta vektora sadrži boju i-tog vrha, a (i, j)-ti element matrice sadrži 1 odnosno 0 ovisno o tome jesu li i-ti i j-ti vrh susjedi ili nisu.

- Sastavni dio implementacije apstraktnog tipa su funkcije koje realiziraju operacije iz tog apstraktnog tipa. Primjerice, pridruživanje boje vrhu realizira se funkcijom koja mijenja odgovarajuću komponentu vektora boja, a pronalaženje susjeda vrha realizira se funkcijom koja čita odgovarajući redak matrice susjedstava i pronalazi jedinice u njemu.

- Konačni program sastoji se od strukture za prikaz obojenog grafa, funkcija koje obavljaju osnovne operacije nad grafom te glavnog programa koji poziva funkcije i povezuje ih u algoritam bojenja grafa.

Programiranje postepenim profinjavanjem uz korištenje apstraktnih tipova donosi brojne prednosti u odnosu na "ad-hoc" programiranje. Evo nekih od tih prednosti.

- Lakše ćemo doći do rješenja, a ono će biti bolje strukturano, razumljivije i pogodno za daljnje održavanje.

- Iskoristit ćemo već postojeća znanja i iskustva umjesto da "otkrivamo toplu vodu". Primjerice, ako naš problem zahtjeva korištenje grafa, tada nećemo morati izmišljati prikaz grafa u računalu već ćemo ga naći u literaturi.

- Možda ćemo ponovo upotrijebiti već gotove dijelove softvera umjesto da ih sami razvijamo. Na primjer, ako je neki drugi programer već razvio implementaciju grafa u C-u, tada uz njegovo dopuštenje tu implementaciju možemo odmah ugraditi u naš program.

- Lakše ćemo se sporazumijevati s drugima. Ako kolegi kažemo da imamo program koji radi s obojenim grafom, kolega će nas bolje razumijeti nego kad bi mu rješenje opisivali nekim svojim riječima.

1.2 Više o strukturama podataka

U ovom odjeljku govorimo detaljnije o strukturama podataka: od čega se one sastoje, kako se one grade. Objašnjavamo da se struktura podataka sastoji od dijelova koji se udružuju u veće cjeline i međusobno povezuju vezama. Uvodimo posebne nazive za dijelove, načine udruživanja te načine povezivanja. Također, uvodimo pravila kako se strukture prikazuju dijagramima.

1.2.1 Dijelovi od kojih se grade strukture podataka

Rekli smo već da se struktura podataka sastoji od varijabli u nekom programu i veza među tim varijablama. To znači da su dijelovi strukture same te varijable, ili točnije mjesta u memoriji računala gdje se mogu pohraniti podaci. Ipak, varijable mogu biti različitih tipova. Zato razlikujemo sljedeće vrste dijelova.

Klijetka (ćelija) ... varijabla koju promatramo kao nedjeljivu cjelinu. Klijetka je relativan pojam jer se jedna cjelina u jednom trenutku može smatrati nedjeljivom, a kasnije se može gledati unutrašnja građa te iste cjeline. Svaka klijetka ima svoj tip, adresu (ime) i sadržaj (vrijednost). Tip i adresa su nepromjenjivi, a sadržaj se može mijenjati.

Polje (`array` u C-u) ... mehanizam udruživanja manjih dijelova u veće. Polje čini više klijetki istog tipa pohranjenih na uzastopnim adresama. Broj klijetki unaprijed je zadan i nepromjenljiv. Jedna klijetka unutar polja zove se element polja i jednoznačno je određena svojim rednim brojem ili indeksom. Po ugledu na C, uzimamo da su indeksi `0, 1, 2, ... , N-1`, gdje je `N` cjelobrojna konstanta.

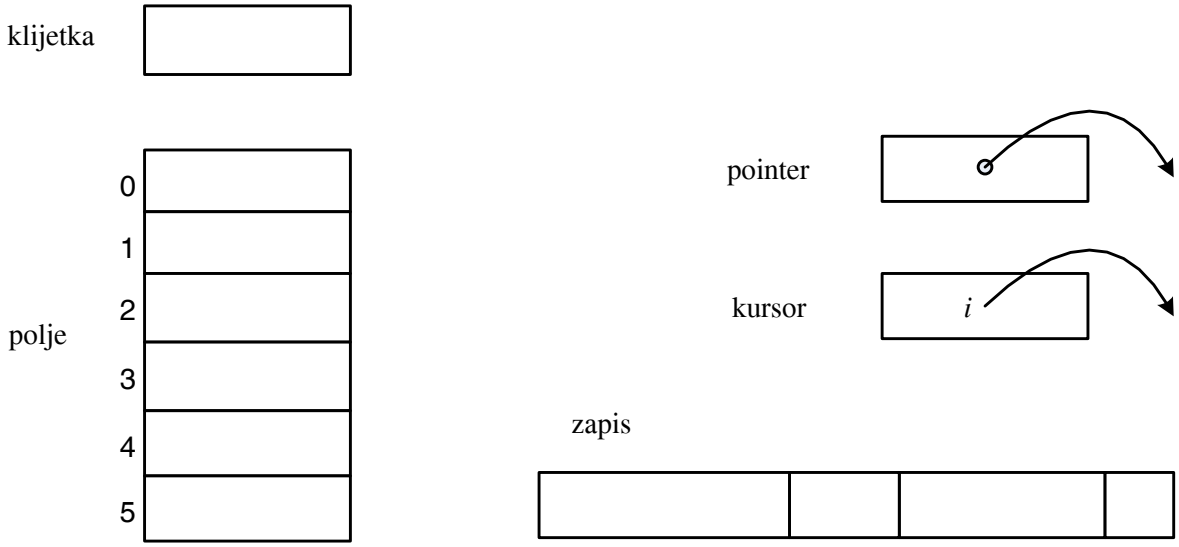
Zapis (slog, `struct` u C-u) ... također mehanizam udruživanja manjih dijelova u veće. Zapis čini više klijetki, koje ne moraju biti istog tipa, no koje su pohranjene na uzastopnim adresama. Broj, redosljed i tip klijetki unaprijed je zadan i nepromjenljiv. Pojedina klijetka unutar zapisa zove se komponenta zapisa i jednoznačno je određena svojim imenom.

Pointer ... služi za uspostavljanje veze između dijelova strukture. Pointer je klijetka koja pokazuje gdje se nalazi neka druga klijetka. Sadržaj pointera je adresa klijetke koju treba pokazati.

Kursor ... također služi za uspostavljanje veze između dijelova strukture. Kursor je klijetka tipa `int` koja pokazuje na element nekog polja. Sadržaj kursora je indeks elementa kojeg treba pokazati.

Strukture podataka precizno se definiraju odgovarajućim naredbama u C-u. No isto tako mogu se prikazati i pomoću dijagrama, što je manje precizno ali je zornije. Svaki dio strukture crta se na određeni način tako da se lakše može prepoznati. Pravila crtanja vidljiva su na slici 1.2.

Dakle u skladu sa slikom 1.2 klijetka se crta kao kućica, njezina adresa ili ime pišu se pored kućice, a sadržaj unutar kućice. Polje se u pravilu crta kao uspravan niz kućica s indeksima elemenata ispisanim uz lijevi rub. Zapis se crta kao vodoravan niz kućica nejednakih veličina s imenima komponenti ispisanim na gornjem ili donjem rubu. I

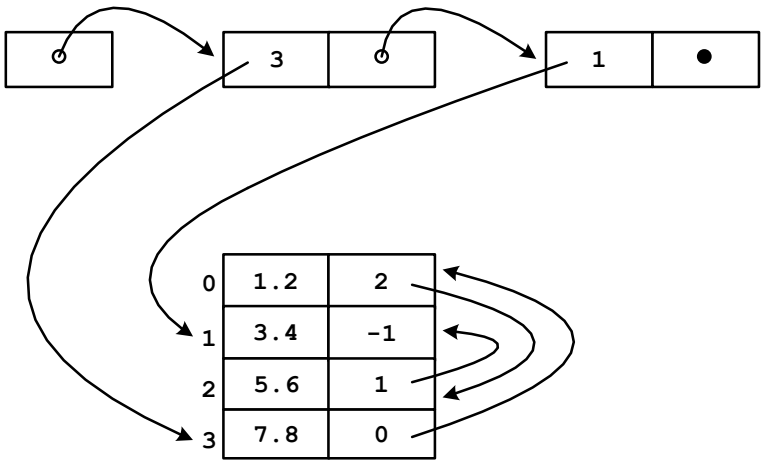


Slika 1.2: Dijelovi strukture podataka, pravila crtanja.

pointer i kursor crtaju se kao kućica iz koje izlazi strelica. Izgled početka strelice malo je drukčiji, naime adresa unutar pointera označena je kružićem, a cijeli broj unutar kursora može biti eksplicitno prikazan.

1.2.2 Povezivanje dijelova strukture u cjelinu

Strukture se grade grupiranjem dijelova u polja ili zapise te povezivanjem dijelova pomoću pointera ili kursora. Polja i zapisi se mogu kombinirati. Na primjer, možemo imati polje zapisa, zapis čije pojedine komponente su polja, polje od polja, zapis čija komponenta je zapis, i slično. Komponente zapisa ili elementi polja mogu biti pointeri ili kursori koji pokazuju na neke druge zapise ili polja ili čak neke druge pointere ili kursore. Sve ove konstrukcije mogu se po volji iterirati.



Slika 1.3: Primjer složenije strukture podataka.

Slika 1.3 sadrži dijagram strukture podataka koja se sastoji od niza zapisa povezanih pointerima te jednog polja zapisa. Zapisi u polju još su međusobno povezani kursorima. Vidimo da se strelica pointera odnosno kursora crta tako da njezin kraj (šiljak) dodiruje klijetku koju treba pokazati. Također vidimo da se nul-pointer (onaj koji nikamo ne pokazuje) označuje kao puni kružić, a nul-kursor kao -1 (nepostojeći indeks).

Struktura na slici 1.3 zanimljivo izgleda no ima jednu manjkavost: nije jasno čemu služi. Molimo čitatelje da razmisle o mogućim interpretacijama te da ih jave autoru nastavnog materijala.

1.3 Više o algoritmima

U ovom odjeljku dajemo neke dodatne napomene o algoritmima. Raspravljamo o raznim načinima kako se algoritmi mogu zapisati te odabiremo način zapisivanja koji ćemo koristiti u ostatku teksta. Dalje govorimo o analiziranju algoritama, posebno o analizi njihove vremenske složenosti, dakle procjeni vremena izvršavanja.

1.3.1 Zapisivanje algoritama

Da bi proučavali algoritme, moramo ih nekako zapisati. Za to nam je potreban jezik za zapisivanje algoritama. Postavlja se pitanje: koji jezik odabrati? Postoje naime barem tri njihove vrste.

Prirodni govorni jezici ... na primjer hrvatski ili engleski. Ne treba ih posebno učiti, razumljivi su svima. No obiluju nejasnim i dvosmislenim konstrukcijama, tako da zapis algoritma u njima može biti neprecizan i višeznačan.

Pseudojezici ... na pola su puta između prirodnih i programskih jezika. Sastoje se od ograničenog skupa strukturiranih rečenica s jasnim značenjem. Zapis algoritma prilično je precizan i jednoznačan, no nije u obliku koji bi se izravno mogao izvoditi na računalu.

Viši programski jezici ... na primjer C, Java, C++, Python. Zahtijevaju učenje, nisu svima razumljivi. Automatski se prevode u programe koji se izvode na računalu. Zapis algoritma sasvim je precizan i jednoznačan, no opterećen je brojnim tehničkim detaljima.

Kao ilustraciju korištenja raznih jezika za zapisivanje algoritama, promatrajmo jednostavan algoritam koji učitava tri broja i ispisuje najmanji od njih. Zapis u prirodnom jeziku mogao bi izgledati ovako:

> *Učitaj sva tri broja. Zatim usporedi prva dva, pa manjeg od njih usporedi s trećim. Ishod druge usporedbe određuje rezultat koji treba ispisati.*

Služimo li se pseudokodom, isti algoritam izgleda ovako:

> učitaj a, b, c;
> ako $a \leq b$ tada $d := a$ inače $d := b$;
> ako $d \leq c$ tada ispiši d inače ispiši c;

Zapis istog algoritma u C-u mogao bi izgledati ovako:

```
#include <stdio.h>
int main( ) {
   int a, b, c, d;
   scanf("%d %d %d", &a,&b,&c);
   if (a <= b)
      d = a;
   else
      d = b;
   if (d <= c)
      printf("%d\n",d);
   else
      printf("%d\n",c);
   return 0;
}
```

Vidimo da je zapis u prirodnom jeziku najjednostavniji no pomalo nejasan. Pseudokod je egzaktan i nedvosmislen. Program u C-u opterećen je tehničkim detaljima pa je manje čitljiv od pseudokoda, no spreman je za izvođenje na računalu. Primijetimo da C-program, zbog sasvim tehničkih razloga, donekle ograničava općenitost algoritma jer predviđa da će brojevi biti cijeli.

U ovom nastavnom materijalu algoritme ćemo zapisivati u C-u. Pouzdajemo se u činjenicu da su studenti već učili taj jezik pa se mogu služiti njime bez ikakve pripreme. Ipak, naši zapisi algoritama obično će biti nedovršeni C-programi. Dozvoljavat ćemo ispuštanje tehničkih detalja ili dijelova koji nisu važni za algoritam te zamjenu ispuštenih dijelova slobodnim tekstom ili komentarima. Na taj način naši zapisi algoritama imat će i sve dobre osobine pseudokoda. Pritom će oni još uvijek biti u obliku koji se vrlo lako može nadopuniti do izvršivog programa.

Primijetimo da u odabranom načinu zapisivanja umjesto samog algoritma zapravo zapisujemo program ili funkciju u C-u koja radi po tom algoritmu. Strogo govoreći, algoritam i program koji radi po njemu nisu iste stvari. Algoritam je plan kako riješiti problem, a program je realizacija tog plana. Za jedan te isti algoritam može se napisati mnogo programa.

Ova razlika između algoritma i programa može se tumačiti kao manjkavost naše metode zapisivanja algoritma. Da bi tu razliku što više smanjili, nastojat ćemo programe pisati na što jednostavniji i očigledniji način, bez suvišnih programerskih trikova i dosjetki. Također, u svrhu smanjivanja razlike služit ćemo se prije spominjanim ispuštanjem tehničkih detalja ili dijelova koji su nevažni za algoritam.

1.3.2 Analiziranje algoritama

Analizirati algoritam znači procijeniti resurse koje algoritam troši pri svojem radu. Najvažniji resursi su: prostor (količina memorije računala koja je zauzeta tijekom rada) i vrijeme (broj vremenskih jedinica koje proteku od početka do kraja rada). Analiza algoritma prvenstveno služi zato da ugrubo predvidi potrebne resurse ovisno o veličini skupa ulaznih podataka. Također, ako imamo više algoritama koji rješavaju isti prob-

lem, analiza tih algoritama trebala bi pokazati koji od njih je bolji u smislu manjeg trošenja resursa.

U ovom nastavnom materijalu analiza algoritama uglavnom se ograničava na **procjenu vremena izvršavanja** tog algoritma. Vrijeme poistovjećujemo s brojem osnovnih operacija koje odgovarajući program treba obaviti (aritmetičke operacije, usporedbe, pridruživanja vrijednosti, ...). Vrijeme izražavamo kao funkciju $T(n)$, gdje je n neka pogodna mjera za veličinu skupa ulaznih podataka. Na primjer:

- Ako promatramo algoritam koji sortira niz brojeva, tada njegovo vrijeme izražavamo kao $T(n)$ gdje je n duljina tog niza brojeva.
- Ako promatramo algoritam za invertiranje kvadratne matrice, tada njegovo vrijeme izražavamo kao $T(n)$ gdje je n red matrice.

U nastavku slijede dva jednostavna ali konkretna primjera analize vremena izvršavanja algoritma i određivanja funkcije $T(n)$. Promatrajmo najprije sljedeći algoritam koji računa skalarni produkt p dva realna vektora a i b duljine n:

```
p = 0;
for (i=0; i<n; i++)
   p = p + a[i]*b[i];
```

U analizi ovog algoritma brojat ćemo operacije množenja, zbrajanja i pridruživanja realnih brojeva. Implicitne operacije s brojačem petlje možemo zanemariti jer su manje zahtjevne, a ionako se pojavljuju uz operacije s realnim brojevima kao neka vrsta njihovog dodatnog tereta. Tada vidimo da se algoritam sastoji od n množenja, n zbrajanja i $n+1$ pridruživanja, pa je njegovo vrijeme izvršavanja $T(n) = 3n + 1$. Promatrajmo dalje algoritam koji računa vektor c duljine n kao umnožak kvadratne matrice A reda n i vektora b duljine n. Matrica i vektori sastoje se od realnih brojeva:

```
for (i=0; i<n; i++) {
   c[i] = 0;
   for (j=0; j<n; j++)
      c[i] = c[i] + a[i][j]*b[j];
}
```

Ovaj algoritam svodi se na n-struko računanje skalarnog produkta, pa je njegovo vrijeme $T(n) = n(3n + 1) = 3n^2 + n$.

Često se dešava da $T(n)$ ne ovisi samo o veličini skupa ulaznih podataka nego također i o vrijednostima tih podataka. Na primjer:

- Algoritam za sortiranje možda brže sortira "skoro sortirani" niz brojeva, a sporije niz koji je "jako izmiješan".
- Algoritam za invertiranje možda brže invertira matricu uglavnom ispunjenu nulama, a sporije matricu koja je dobro popunjena elementima različitim od nule.

Tada $T(n)$ definiramo kao **vrijeme u najgorem slučaju**, dakle kao maksimum za vrijeme izvršavanja po svim skupovima ulaznih podataka veličine n.

Osim vremena u najgorem slučaju moguće je promatrati i **vrijeme u prosječnom slučaju** - njega ćemo označiti s $T_{avg}(n)$. Ono se dobiva kao matematičko očekivanje (srednja vrijednost) za vrijeme izvršavanja. Da bismo mogli govoriti o očekivanju ili o srednjoj vrijednosti, moramo znati distribuciju za razne skupove ulaznih podataka veličine n. Obično se zbog jednostavnosti pretpostavlja uniformna distribucija, dakle smatra se da su svi skupovi ulaznih podataka veličine n jednako vjerojatni.

U ovom nastavnom materijalu uglavnom ćemo se ograničiti na određivanje funkcije $T(n)$, dakle na analizu vremena izvršavanja algoritma u najgorem slučaju. Samo ponekad odredit ćemo i $T_{avg}(n)$, dakle vrijeme izvršavanja u prosječnom slučaju. Naime, analizu u najgorem slučaju obično je lakše provesti. Također, analiza u prosječnom slučaju od upitnog je značaja ako nismo sigurni kako zaista izgleda distribucija skupova ulaznih podataka iste veličine.

Primijetimo da je fukcija $T(n)$ u pravilu vrlo neprecizna. Ona ne može točno u sekundu ili milisekundu odrediti vrijeme izvršavanja algoritma. Osnovni razlog za nepreciznost je taj što smo $T(n)$ dobili brojanjem operacija, makar neke od tih operacija traju dulje a neke kraće. Drugi razlog za nepreciznost je taj što vrijeme izvršavanja pojedine operacije ovisi o računalu koje tu operaciju izvršava, to jest ista operacija na slabijem računalu može trajati 1 mikrosekundu, a na snažnijem 100 puta kraće. Funkcija $T(n)$ zapravo i ne služi za točno određivanje vremena, već da opiše relativne odnose vremena za razne veličine skupa ulaznih podataka. Na primjer:

- Ako algoritam za sortiranje na nekom računalu niz duljine 1000 sortira u 1 sekundi, koliko će mu vremena na istom računalu trebati da sortira niz duljine 2000?
- Ako algoritam za invertiranje matricu reda 100 invertira u vremenu od 1 sekunde, koliko će mu vremena trebati za invertiranje matrice reda 1000 na istom računalu?

Zbog uočene nepreciznosti, funkciju $T(n)$ očito ne treba detaljno zapisivati, dovoljno je utvrditi njezin red veličine. Na primjer, ako prebrojavanjem operacija dobijemo da je $T(n) = 8n^3 + 5n^2 + 3n + 1$, tada je ponašanje te funkcije za velike n-ove određeno članom $8n^3$, a ostali članovi se mogu zanemariti. Kažemo da $T(n)$ raste proporcionalno s n^3 ili da je $T(n)$ reda veličine n^3. Pišemo: $T(n) = \mathcal{O}(n^3)$.

Oznaka "veliko $\mathcal{O}$" koju smo ovdje upotrijebili zapravo je standardna oznaka iz matematičke analize. Njezina točna definicija glasi ovako:

> Neka su $T(n)$ i $f(n)$ dvije realne funkcije definirane na skupu prirodnih brojeva. Kažemo da je $T(n) = \mathcal{O}(f(n))$ ako postoji konstanta $c > 0$ i prirodni broj n_0 tako da za sve prirodne brojeve $n \geq n_0$ vrijedi:
>
> $$|T(n)| \leq c \cdot |f(n)|.$$

Definicija ustvari kaže da $T(n)$ raste istim intenzitetom kao $f(n)$ ili sporije od toga.

Malo prije tvrdili smo da za $T(n) = 8n^3 + 5n^2 + 3n + 1$ vrijedi $T(n) = \mathcal{O}(n^3)$. Ta tvrdnja zaista je u skladu s definicijom oznake "veliko $\mathcal{O}$". Naime zbog $1 \leq n \leq n^2 \leq n^3$ vrijedi sljedeća ocjena:

$$T(n) = 8n^3 + 5n^2 + 3n + 1 \leq 8n^3 + 5n^3 + 3n^3 + 1n^3 = 17n^3.$$

Dakle zaista, definicija "veliko $\mathcal{O}$" je zadovoljena sa $c = 17$ i $n_0 = 1$.

U ovom nastavnom materijalu analizirat ćemo brojne algoritme. Za svakog od njih procjena vremena izvršavanja bit će u obliku $T(n) = \mathcal{O}(f(n))$, gdje je $\mathcal{O}(f(n))$ jedan od sljedećih "referentnih" redova veličine:

$$\mathcal{O}(1) < \mathcal{O}(\log n) < \mathcal{O}(n) < \mathcal{O}(n \log n) < \mathcal{O}(n^2) < \mathcal{O}(n^3) < \ldots < \mathcal{O}(2^n) < \mathcal{O}(n!).$$

Logaritam koji se ovdje koristi može biti dekadski, prirodni, binarni ili bilo koji drugi – naime logaritmi u raznim bazama razlikuju se do na množenje s konstantom, tako da u skladu s definicijom "veliko $\mathcal{O}$" svi algoritmi proizvode isti red veličine.

U gore navedenom nizu referentni redovi veličine nanizani su od "manjih" prema "većima". Točna definicija uređaja među redovima veličine glasi ovako:

> Neka su $f(n)$ i $g(n)$ dvije realne funkcije definirane na skupu prirodnih brojeva. Kažemo da je $\mathcal{O}(f(n)) < \mathcal{O}(g(n))$ ako za bilo koje dvije konstante $c_1 > 0$ i $c_2 > 0$ postoji prirodni broj n_0 tako da za sve prirodne brojeve $n \geq n_0$ vrijedi:
> $$c_1 \cdot |f(n)| \leq c_2 \cdot |g(n)|.$$

Drugim riječima, $g(n)$ prije ili kasnije mora prestići $f(n)$, čak i onda kad $g(n)$ pomnožimo jako malom koeficijentom a $f(n)$ jako velikim koeficijentom.

Očito, ako je $T(n)$ vrijeme izvršavanja nekog algoritma te vrijedi $T(n) = \mathcal{O}(f(n))$ i $\mathcal{O}(f(n)) < \mathcal{O}(g(n))$, tada također vrijedi $T(n) = \mathcal{O}(g(n))$. Primjerice, ako algoritam za sortiranje niza duljine n troši vrijeme $\mathcal{O}(n^2)$, tada u skladu s definicijom "veliko $\mathcal{O}$" to vrijeme je također i $\mathcal{O}(n^3)$. Naravno, u analizi vremena izvršavanja nastojimo uvijek za $T(n)$ odrediti najmanji mogući red veličine, jer takva procjena je najvjerodostojnija.

Korist od određivanja reda veličine za $T(n)$ je u tome što na osnovi reda veličine možemo ugrubo utvrditi povećanje $T(n)$ ovisno o povećanju n. Na primjer:

- Imamo algoritam za sortiranje niza od n brojeva s vremenom izvršavanja $T(n) = \mathcal{O}(n^2)$. Ako se duljina niza udvostruči, vrijeme sortiranja moglo bi se učetverostručiti.

- Imamo algoritam za invertiranje matrice reda n s vremenom izvršavanja $T(n) = \mathcal{O}(n^3)$. Ako se red matrice udvostruči, njezino invertiranje moglo bi trajati 8 puta dulje.

Korist od određivanja reda veličine za $T(n)$ je također i u tome što na osnovi redova veličine možemo uspoređivati algoritme. Ako neki problem možemo riješiti s dva algoritma, prvi od njih ima vrijeme izvršavanja $T_1(n) = \mathcal{O}(f_1(n))$, drugi ima vrijeme $T_2(n) = \mathcal{O}(f_2(n))$, a pritom vrijedi $\mathcal{O}(f_1(n) < \mathcal{O}(f_2(n))$, tada smatramo da je prvi algoritam brži od drugog. Zaista, po definiciji "veliko $\mathcal{O}$" i pod pretpostavkom da su redovi veličine za vremena izvršavanja vjerodostojno utvrđeni, to znači da za dovoljno veliki n prvi algoritam radi brže od drugog. Na primjer:

- Ako imamo dva algoritma za sortiranje niza od n brojeva, prvi od njih ima vrijeme $\mathcal{O}(n \log n)$ a drugi vrijeme $\mathcal{O}(n^2)$, tada prvi algoritam smatramo bržim.

- Ako imamo dva algoritma za invertiranje matrice reda n, prvi od njih ima vrijeme $\mathcal{O}(n^3)$ a drugi vrijeme $\mathcal{O}(n^4)$, tada prvi algoritam smatramo bržim.

Ovi zaključci vrijede pod pretpostavkom da je n dovoljno velik. Za male n-ove odnos brzina algoritama može biti drukčiji.

Na kraju, primijetimo da stvarna upotrebljivost algoritma bitno ovisi o redu veličine za vrijeme njegovog izvršavanja.

- Algoritmi s vremenom $T(n) = \mathcal{O}(1)$ imaju vrijeme ograničeno konstantom. Oni su vrlo upotrebljivi jer im vrijeme ne ovisi o veličini skupa ulaznih podataka.
- Algoritmi s vremenom $T(n) = \mathcal{O}(\log n)$ također su vrlo brzi jer im vrijeme izvršavanja raste znatno sporije nego što raste skup ulaznih podataka.
- Kod algoritama s vremenom $T(n) = \mathcal{O}(n)$ vrijeme izvršavanja raste linearno (propocionalno) s veličinom skupa ulaznih podataka.
- Algoritmi s vremenom $T(n) = \mathcal{O}(n \log n)$, $T(n) = \mathcal{O}(n^2)$ ili $T(n) = \mathcal{O}(n^3)$ smatraju se koliko-toliko upotrebljivima, makar im vrijeme izvršavanja raste neproporcionalno u odnosu na povećanje skupa ulaznih podataka.
- Algoritmi s vremenom $T(n) = \mathcal{O}(n^k)$ za neki prirodni broj k zovu se polinomijalni algoritmi. Barem u teoriji oni se smatraju upotrebljivima, makar za veliki k u stvarnosti mogu biti prespori.
- Algoritmi s vremenom $T(n) = \mathcal{O}(2^n)$ ili $T(n) = \mathcal{O}(n!)$ zovu se eksponencijalni ili nad-eksponencijalni algoritmi. Upotrebljivi su samo za vrlo male n-ove. Za iole veći n zahtijevaju vrijeme koje se mjeri u godinama ili stoljećima.

Poglavlje 2

LISTE

Ovo poglavlje obrađuje liste, dakle "linearne" apstraktne tipove podataka gdje su podaci poredani u niz i gdje među podacima postoji odnos prethodnika i sljedbenika. Poglavlje se sastoji od tri odjeljka. Prvo od njih posvećeno je općenitoj listi, a preostala dva obrađuju posebne vrste liste kao što su stog i red.

2.1 (Općenita) lista

U ovom odjeljku proučavamo općenitu listu, dakle apstraktni tip podataka gdje su podaci poredani u niz i gdje je dozvoljeno ubacivanje i izbacivanje podataka na bilo kojem mjestu u tom nizu. Najprije navodimo svojstva i primjene liste, a zatim obrađujemo dvije bitno različite implementacije liste: pomoću polja odnosno pomoću pointera.

2.1.1 Svojstva i primjene liste

Lista (*list*) je konačni niz (od nula ili više) podataka istog tipa. Podaci koji čine listu nazivaju se njezini **elementi**. U teorijskim razmatranjima listu obično bilježimo ovako:

$$(a_1, a_2, \ldots, a_n).$$

Ovdje je $n \geq 0$ tzv. **duljina** liste. Ako je $n = 0$, kažemo da je lista prazna. Za $n \geq 1$, a_1 je prvi element, a_2 drugi, ..., a_n zadnji element. Moguće je da su neki od elemenata liste jednaki. Naime, identitet elementa određen je njegovom **pozicijom** (rednim brojem) a ne njegovom vrijednošću.

Važno svojstvo liste je da su njezini elementi linearno uređeni s obzirom na svoju poziciju. Kažemo da je a_i ispred a_{i+1}, te da je a_i iza a_{i-1}.

Broj elemenata u listi nije fiksiran: elementi se mogu ubacivati ili izbacivati na bilo kojem mjestu - na taj način lista može rasti ili se smanjivati. Primijetimo da lista nije isto što i polje. Naime, u polju je broj elemenata unaprijed zadan i nepromjenjiv.

U nastavku navodimo nekoliko primjera korištenja liste. Ti primjeri pokazuju da se liste često pojavljuju, kako u običnom životu tako i u računarstvu.

- Riječ u nekom tekstu je lista znakova. Npr. riječ ZNANOST može se interpretirati kao ('Z', 'N', 'A', 'N', 'O', 'S', 'T'). Primijetimo da ovdje dva različita elementa liste imaju istu vrijednost 'N'. No ta dva elementa razlikujemo na osnovi njihove

pozicije. Osim pojedine riječi, cijeli redak u tekstu također možemo shvatiti kao listu znakova, s time da ona mora uključivati i bjeline. Slično, cijeli tekst je ustvari lista redaka. Očito, bilo koji program za rad s tekstom (dakle tekst-editor ili tekst-procesor) mora biti u stanju rukovati s listama znakova i redaka. Štoviše, takav program mora omogućiti ubacivanje novih elemenata u liste, izbacivanje postojećih elemenata, kretanje po listi, njezino čitanje, itd.

- Polinom u matematici bilježimo na sljedeći način:

$$P(x) = a_1 x^{e_1} + a_2 x^{e_2} + \ldots + a_n x^{e_n}.$$

Ovdje su $a_1, a_2, \ldots, a_n$ koeficijenti, a $e_1, e_2, \ldots, e_n$ su eksponenti ($0 \leq e_1 < e_2 < \ldots < e_n$). Da bi zabilježili polinom, dovoljno je zabilježiti listu parova koeficijenata i pripadnih eksponenata, dakle listu oblika:

$$((a_1, e_1), (a_2, e_2), \ldots, (a_n, e_n)).$$

Program za simboličko računanje s polinomima (npr. WRI Mathematica) očito mora biti u stanju raditi s ovakvim listama. Dakle za dvije liste koje predstavljaju zadane polinome on mora biti u stanju proizvesti novu listu koja predstavlja zbroj ili umnožak zadanih polinoma.

- U nekim programskim jezicima lista je osnovna tvorevima od koje se grade sve ostale. Primjer takvog jezika je LISP (*LISt Processing*). Program u LISP-u sastoji se od funkacija, a svaka funkcija shvaća se kao lista čiji prvi element je njezino ime, a ostali elementi su njezini argumenti. Također, osnovni tip podataka u LISP-u je lista, a svi ostali tipovi grade se iz tog osnovnog tipa. Algoritmi u LISP-u obično se implementiraju pomoću rekurzije - pritom se sama lista promatra kao rekurzivno građena tvorevina koja se sastoji od prvog elementa i manje liste koju čine ostali elementi. Očito, interpreter jezika LISP je program koji prikazuje liste i rukuje njima služeći se rekurzijom.

- Baza podataka o stanovnicima nekog grada zapravo je skup zapisa o osobama pohranjenih u vanjskoj memoriji računala. Pretraživanjem baze obično dobivamo listu zapisa koji zadovoljavaju neke uvjete. Npr. ako postavimo upit: ispisati prezimena i imena ljudi koji su rođeni 1990. godine a po zanimanju su matematičari, rezultat vjerojatno neće biti samo jedno ime i prezime nego lista oblika:

(Babić Mato, Marković Pero, Petrović Marija, ...).

Dakle, program za pretraživanje baze podataka mora biti u stanju pohranjivati liste zapisa o osobama, sortirati ih na željeni način, ispisivati ih, i slično.

Iz navedenih primjera vidljivo je da postoje brojni programi koji u svojem radu moraju pohranjivati liste te obavljati razne operacije nad njima. Drugim riječima, postoje brojni programi kojima je potreban apstraktni tip podataka za liste. Da bismo pojam liste pretvorili u apstraktni tip, trebamo uočiti sve tipove podataka koji se pojavljuju te definirati sve potrebne operacije. To se može učiniti na razne načine. Apstraktni tip koji slijedi samo je jedna od mogućih varijanti. Od raznih operacija odabrali

smo samo one elementarne, smatrajući da će se složenije operacije moći realizirati kao kombinacije elementarnih. Slijedi definicija apstraktnog tipa.

Apstraktni tip podataka List

`elementtype` ... bilo koji tip.

`List` ... podatak tipa `List` je konačni niz (ne nužno različitih) podataka tipa `elementtype`.

`position` ... podatak ovog tipa služi za identificiranje elementa u listi, dakle za zadavanje pozicije u listi. Smatramo da su u listi $(a_1, a_2, \ldots, a_n)$ definirane pozicije koje odgovaraju prvom, drugom, ..., n-tom elementu, a također i pozicija na kraju liste (neposredno iza n-tog elementa).

`LiEnd(L)` ... funkcija koja vraća poziciju na kraju liste `L`.

`LiMakeNull(&L)` funkcija pretvara listu `L` u praznu listu, i vraća poziciju `LiEnd(L)`.

`LiInsert(x,p,&L)` ... funkcija ubacuje podatak `x` na poziciju `p` u listu `L`. Pritom se elementi koji su dotad bili na poziciji `p` i iza nje pomiču za jednu poziciju dalje. Dakle, ako je `L` oblika $(a_1, a_2, \ldots, a_n)$, tada `L` postaje $(a_1, a_2, \ldots, a_{\mathtt{p}-1}, \mathtt{x}, a_{\mathtt{p}}, \ldots, a_n)$. Ako je `p == LiEnd(L)` tada `L` postaje $(a_1, a_2, \ldots, a_n, \mathtt{x})$. Ako u `L` ne postoji pozicija `p`, rezultat je nedefiniran.

`LiDelete(p,&L)` ... funkcija izbacuje element na poziciji `p` iz liste `L`. Dakle, ako je lista `L` oblika $(a_1, a_2, \ldots, a_n)$ tada `L` nakon izbacivanja postaje $(a_1, a_2, \ldots, a_{\mathtt{p}-1}, a_{\mathtt{p}+1}, \ldots, a_n)$. Rezultat nije definiran ako `L` nema pozicije `p` ili ako je `p == LiEnd(L)`.

`LiFirst(L)` ... funkcija vraća prvu poziciju u listi `L`. Ako je `L` prazna, vraća `LiEnd(L)`.

`LiNext(p,L), LiPrevious(p,L)` ... funkcije vraćaju poziciju iza odnosno ispred `p` u listi `L`. Ako je `p` zadnja pozicija u `L`, tada je `LiNext(p,L) == LiEnd(L)`. Funkcija `LiNext( )` je nedefinirana za `p == LiEnd(L)`. Funkcija `LiPrevious( )` je nedefinirana za `p == LiFirst(L)`. Obje funkcije su nedefinirane ako `L` nema pozicije `p`.

`LiRetrieve(p,L)` ... funkcija vraća element na poziciji `p` u listi `L`. Rezultat je nedefiniran ako je `p == LiEnd(L)` ili ako `L` nema pozicije `p`.

U ostatku ovog odjeljka raspravljamo o implementacijama liste. Kao prvo, da bismo listu mogli prikazati u memoriji računala, potrebna nam je odgovarajuća struktura podataka. U oblikovanju takve strukture postoje dva pristupa:

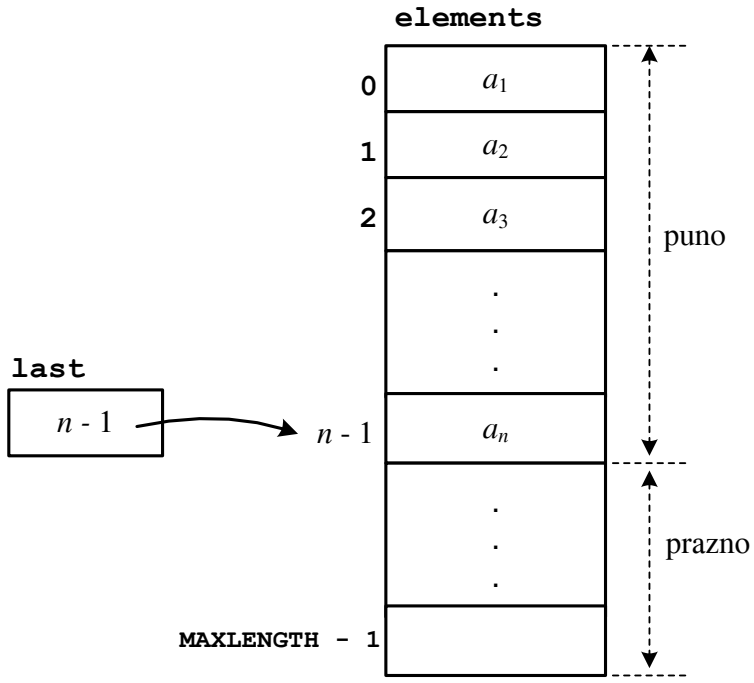
1. Elemente liste pohranjujemo u memoriju računala tako da se njihov fizički redoslijed u memoriji poklapa s logičkim redoslijedom u listi. To znači da elemente zapravo redom spremamo u neku vrstu polja. Logički redoslijed tada ne treba zapisivati jer on implicitno slijedi iz načina na koji su podaci fizički pohranjeni.

2. Elemente liste pohranjujemo na proizvoljna mjesta u memoriju računala. No tada se fizički redoslijed ne poklapa s logičkim, pa se logički redoslijed mora eksplicitno zapisati. U tu svrhu služimo se nekom vrstom pokazivača, dakle pointerima ili kursorima. Praćenjem pokazivača moguće je reproducirati logički redoslijed elemenata u listi bez obzira na njihov stvarni fizički raspored.

Oba pristupa dozvoljavaju razne varijante. U nastavku opisujemo dvije implementacije liste, od kojih prva za prikaz liste koristi polje, a druga se služi pointerima.

2.1.2 Implementacija liste pomoću polja

Implementacije je zasnovana je na strukturi podataka prikazanoj na slici 2.1. Elementi liste spremljeni su u uzastopnim klijetkama polja `elements[ ]` duljine `MAXLENGTH`, gdje je `MAXLENGTH` unaprijed odabrana konstanta. Također imamo kursor `last` koji pokazuje gdje se zadnji element liste nalazi u polju. Kod ove implementacije lagano je pročitati i-ti element. Također je lagano ubacivanje i izbacivanje na kraju liste. S druge strane, ubacivanje i izbacivanje u sredini liste zahtijeva fizičko pomicanje (prepisivanje) dijela podataka. Pritom je duljina liste ograničena veličinom polja, dakle konstantom `MAXLENGTH`.



Slika 2.1: Implementacija liste pomoću polja - korištena struktura podataka.

Implementacija liste pomoću polja može se precizno izraziti u C-u. Slijede definicije tipova te programski kod za neke od funkcija.

```
#define MAXLENGTH ...  /* neka pogodna konstanta */
```

```
typedef struct {
  int last;
  elementtype elements[MAXLENGTH];
} List;

typedef int position;

position LiEnd (List L) {
  return (L.last + 1);
}

position LiMakeNull (List *Lp) {
  Lp->last = -1;
  return 0;
}

void LiInsert (elementtype x, position p, List *Lp) {
  position q;
  if (Lp->last >= MAXLENGTH-1)
    exit(101); /* lista se prepunila */
  else if ( (p > Lp->last+1 ) || ( p < 0 ) )
    exit(102); /* pozicija ne postoji */
  else {
    for (q = Lp->last; q >= p; q--)
      /* pomakni elemente na pozicijama p, p+1,...jedno mjesto dalje */
      Lp->elements[q+1] = Lp->elements[q];
    Lp->last++;
    Lp->elements[p] = x;
  }
}

void LiDelete (position p, LIST *Lp) {
  position q;
  if ( (p > Lp->last ) || (p < 0) )
    exit(102); /* pozicija ne postoji */
  else {
    Lp->last--;
    for (q = p; q <= Lp->last; q++)
      /* pomakni elemente na pozicijama p+1, p+2,...mjesto natrag */
      Lp->elements[q] = Lp->elements[q+1];
  }
}
```

Kao što vidimo, u implementaciji je potrebno unaprijed izabrati konkretnu vrijednost za `MAXLENGTH`. Ako bi tu vrijednost htjeli promijeniti, sve dijelove programa morali bi ponovo prevesti (kompilirati). Također vidimo da nije navedena definicija za `elementtype` - to je zato što je taj tip zapravo proizvoljan. Naravno, konačni program

morao bi negdje sadržavati naredbu oblika `typedef ... elementtype` te bi u slučaju promjene te naredbe opet morali sve ponovo prevesti.

Struktura podataka sa slike 2.1 definirana je kao C-ov `struct` koji se sastoji od jedne cjelobrojne varijable i polja tipa `elementtype` duljine `MAXLENGTH`. Budući da poziciju elementa u listi možemo poistovjetiti s indeksom tog istog elementa u polju, tip `position` poistovjećen je s `int`.

Način rada navedenih funkcija razumljiv je iz njihovog teksta. Svaka funkcija ima tip i argumente u skladu definicijom odgovarajuće operacije iz apstraktnog tipa. Funkcije koje mijenjaju listu umjesto same liste `L` kao argument primaju njezinu adresu `Lp`.

Funkcija `LiInsert( )` svodi se na uvećavanje kursora `last` za 1, prepisivanje elemenata u polju počevši od onog s indeksom `p` za jedno mjesto dalje (tako da se oslobodi klijetka s indeksom `p`) te upisa novog elementa `x` u klijetku s indeksom `p`. Pritom se pazi na redoslijed prepisivanja da ne bi došlo do gubitka podataka.

Slično, funkcija `LiDelete( )` svodi se na smanjivanje kursora `last` za 1 te prepisivanje elemenata u polju s indeksima nakon `p` za jedno mjesto unatrag (tako da se "pregazi" element koji je bio na indeksu `p`). Opet je važan redoslijed prepisivanja.

Funkcija `LiEnd( )` vraća indeks koji slijedi nakon indeksa zapisanog u kursoru `last`. Funkcija `LiMakeNull( )` prazni listu tako da postavi kursor `last` na -1; to je naime vrijednost koja će se prvim ubacivanjem pretvoriti u 0.

Primijetimo da u opisanoj implementaciji postoje izvanredne situacije kad funkcije ne mogu izvršiti svoj zadatak. Zaista, ako lista već popunjava cijelo polje `elements[ ]`, tj. već sadrži `MAXLENGTH` elemenata, tada funkcija `LiInsert( )` nije u stanju ubaciti novi element jer za njega fizički nema mjesta. Slično, funkcija `LiDelete( )` može izbaciti element samo ako on postoji, tj. samo ako je njegova pozicija `p` u rasponu između 0 i `last`. U ovakvim izvanrednim situacijama naše funkcije reagiraju tako da pozovu standardnu funkciju `exit( )` i time naprasno prekinu cijeli program. Pritom se operacijskom sustavu prosljeđuje statusni kod greške iz kojeg je moguće otkriti što se dogodilo. Primjerice, kod 101 znači da se lista prepunila, a kod 102 da se pokušala koristiti nepostojeća pozicija.

Ostali potprogrami koji čine implementaciju a nisu navedeni u prethodnom programskom kodu vrlo su jednostavni. Zaista, `LiFirst(L)` uvijek vraća 0, `LiNext(p,L)` odnosno `LiPrevious(p,L)` vraćaju `p+1` odnosno `p-1`, `LiRetrieve(p,L)` vraća vrijednost `L.elements[p]`. Pritom se opet provjerava jesu li parametri u dozvoljenom rasponu pa se u slučaju izvanrednih situacija poziva `exit( )`.

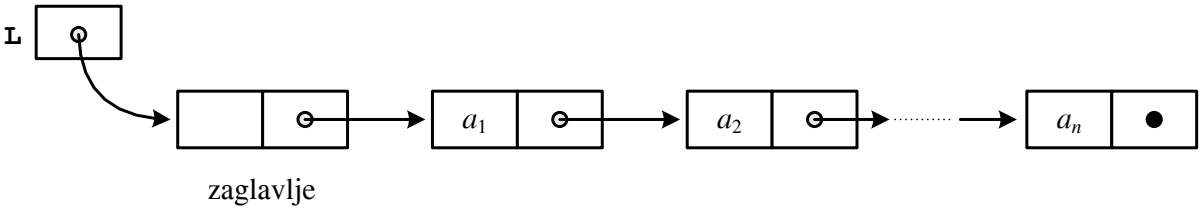
Pregledom programskog koda lako je uočiti da je vrijeme izvršavanja za funkcije `LiInsert( )` i `LiDelete( )` reda veličine $\mathcal{O}(n)$, gdje je n duljine liste. Za ostale funkcije vrijeme je konstantno, dakle $\mathcal{O}(1)$.

Implementacija liste pomoću polja ima svojih prednosti. Kao što smo već naveli, ona omogućuje izravan pristup i-tom elementu bez čitanja prethodnih elemenata. Također, ona troši malo memorije ako je broj elemenata predvidiv i malo se mijenja. No s druge strane, mana je da ubacivanje i izbacivanje elemenata troše linearno vrijeme. Nadalje, uočili smo još jednu veliku manu, a to je mogućnost prepunjenja polja. U skladu s time, implementacija liste pomoću polja smatra se pogodnom pod sljedećim uvjetima:

- Moguće je zadati (ne preveliku) gornju ogradu za duljinu liste;
- Nema mnogo ubacivanja/izbacivanja u sredinu liste.

2.1.3 Implementacija liste pomoću pointera

Implementacija pomoću pointera listu prikazuje nizom klijetki u skladu sa slikom 2.2. Svaka klijetka sadrži jedan element liste i pointer na istu takvu klijetku koja sadrži idući element liste. Struktura sa slike 2.2 obično se zove *vezana lista*. Postoji i polazna klijetka (zaglavlje) koja označava početak liste i ne sadrži nikakav element. Samu listu poistovjećujemo s pointerom na zaglavlje.



Slika 2.2: Implementacija liste pomoću pointera - korištena struktura podataka.

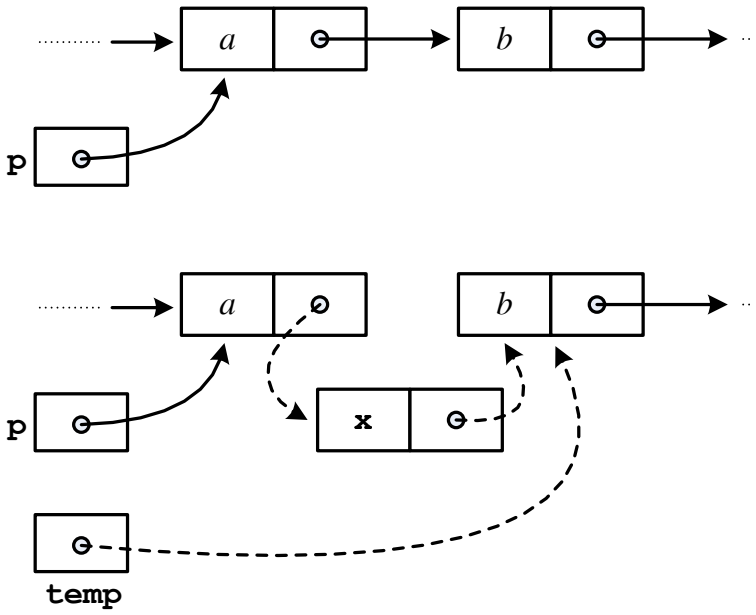
Kod ove implementacije lagano se ubacuju i izbacuju elementi na bilo kojem dijelu liste. S druge strane, nešto je teže pročitati i-ti element: potrebno je redom čitati prvi, drugi, ..., i-ti element. Također, teže je odrediti kraj liste ili prethodni element.

Pojam pozicije nešto je složeniji nego prije. Pozicija elementa a_i definira se kao pointer na klijetku koja sadrži pointer na a_i. Znači, pozicija od a_1 je pointer na zaglavlje, pozicija od a_2 je pointer na klijetku u kojoj je zapisan a_1, itd. Pozicija `LiEnd(L)` je pointer na zadnju klijetku u vezanoj listi. Ova pomalo neobična definicija uvodi se zato da bi se efikasnije obavljale operacije `LiInsert( )` i `LiDelete( )`.

Slijedi zapis ove implementacija u C-u. Način rada funkcije `LiInsert( )` odnosno `LiDelete( )` podrobnije je prikazan slikom 2.3 odnosno 2.4. Na tim slikama pune strelice označavaju polaznu situaciju (prije početka rada funkcije), a crtkane strelice označavaju promjene (nakon što je funkcija obavila svoj zadatak).

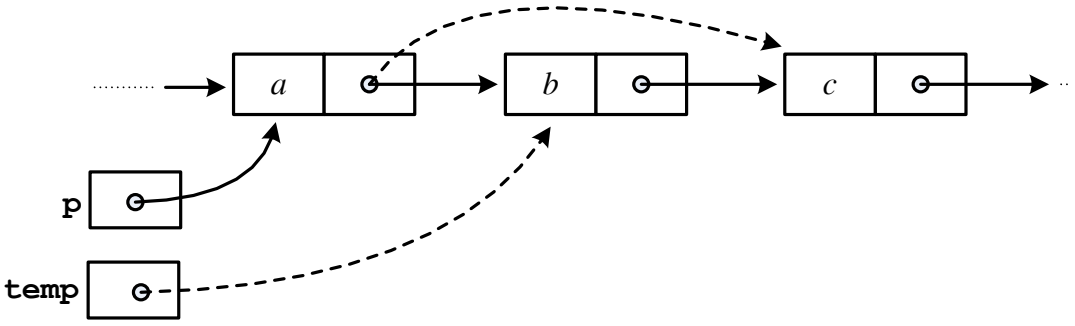
```
typedef struct celltag {
   elementtype element;
   struct celltag *next;
} celltype;

typedef celltype *List;

typedef celltype *position;

position LiEnd (List L) {
  /* vraća pointer na zadnju klijetku u L */
  position q;
  q = L;
  while (q->next != NULL)
    q = q->next;
  return q;
}
```

```
position LiMakeNull (List *Lp) {
  *Lp = (celltype*) malloc(sizeof(celltype));
  (*Lp)->next = NULL;
  return(*Lp);
}

void LiInsert (elementtype x, position p) {
  position temp;
  temp = p->next;
  p->next = (celltype*) malloc(sizeof(celltype));
  p->next->element = x;
  p->next->next = temp;
}

void LiDelete (position p) {
  position temp;
  temp = p->next;
  p->next = p->next->next;
  free(temp);
}
```



Slika 2.3: Implementacija liste pomoću pointera - funkcija `LiInsert( )`.

Kao što vidimo, implementacija opet započinje definicijom tipova. Najprije je potrebno definirati tip `celltype` za klijetke od kojih se gradi naša vezana lista. Tipovi `List` i `position` mogu se zatim poistovjetiti s pointerima na `celltype`. Definicija tipa `elementtype` opet se ispušta jer se može odabrati po volji.



Slika 2.4: Implementacija liste pomoću pointera - funkcija `LiDelete( )`.

Funkcija `LiInsert( )` najprije fizički stvara novu klijetku pozivom standardne funkcije za dinamičku alokaciju memorije `malloc( )`. Zatim u tu klijetku upisuje novi element `x` te je uključuje u listu na poziciju `p`. Ili bolje rečeno, nova klijetka uključuje se u vezanu listu neposredno ispred klijetke koja je do tada bila na poziciji `p` - to je na slici 2.3 ona sa sadržajem b. Vidimo da se promjena zapravo zbiva u klijetki sa sadržajem a, dakle jedno mjesto ispred b. To opravdava našu "neobičnu" definiciju pozicije gdje je `p` zapravo adresa od a (koja nam treba) a ne adresa od b (koja ionako piše uz a).

Funkcija `LiDelete( )` najprije promjenom pointera "premošćuje" klijetku koja se nalazi na poziciji `p` - to je na slici 2.4 opet ona sa sadržajem b. Zatim se klijetka s b fizički uklanja pozivom standardne funkcije za "recikliranje" dinamički alocirane memorije `free( )`. Promjena pointera opet se zbiva u klijetki sa sadržajem a, dakle onoj koja je jedno mjesto ispred b i čiju adresu znamo zahvaljujući načinu definiranja pozicije.

Funkcija `LiMakeNull( )` pozivom `malloc( )` stvara novu klijetku i u nju upisuje pointer `NULL`, a zatim njezinu adresu upisuje u onu varijablu `L` čija adresa je bila proslijeđena kao `Lp`. Na taj način `L` postaje lista koja se sastoji samo od zaglavlja, dakle prazna lista. Argument `Lp` zapravo je pointer na pointer, dakle podatak tipa `**celltype`.

Primijetimo da funkcije `LiInsert( )` i `LiDelete( )` sadrže manje argumenata nego što je predviđeno u definiciji apstraktnog tipa. To je zato što u slučaju ove implementacije pozicija `p` već jednoznačno određuje o kojoj listi je riječ. Ako želimo biti sasvim "kompatibilni" s definicijom apstraktnog tipa, tada možemo objema funkcijama dodati nedostajući argument `List *Lp`, no funkcije će ga ignorirati.

Potprogrami `LiFirst( )`, `LiNext( )` i `LiRetrieve( )` koji nisu bili navedeni u prethodnom programskom kodu vrlo su jednostavni: svode se na po jedno pridruživanje. Funkcija `LiPrevious( )` ne može se lijepo implementirati: jedini način je da prođemo cijelom listom od početka do zadane pozicije. Slično vrijedi i za `LiEnd( )`. Vrijeme izvršavanja za `LiEnd( )` i `LiPrevious( )` očito iznosi $\mathcal{O}(n)$, gdje je n duljina liste. Sve ostale operacije imaju vrijeme $\mathcal{O}(1)$.

U odnosu na implementaciju pomoću polja, mana implementacije liste pomoću pointera je da ona troši više memorije po elementu; naime uz svaki podatak pohranjuje se i jedan pointer. Daljnje mane su: spor pristup do i-tog, odnosno zadnjeg, odnosno prethodnog elementa. Glavna prednost je da kod ubacivanja ili brisanja elementa nema

potrebe za prepisivanjem drugih podataka. Također važna prednost je da se uglavnom zaobilazi problem prepunjenja koji je postojao kod implementacije pomoću polja. Zaista, novi podaci se više ne upisuju u unaprijed rezervirani i ograničeni memorijski prostor, već u dijelove memorije koji se dinamički alociraju po potrebi.

Doduše, zaliha memorije u računalu nije beskonačna, tako da ni dinamička alokacija nije sasvim imuna na prepunjenje. Ipak, riječ je o vrlo fleksibilnom mehanizmu gdje se izvanredne situacije dešavaju vrlo rijetko. Fleksibilnost dinamičke alokacije postiže se time što ona funkcionira na razini cijelog programa, a ne na razini pojedinih struktura. To primjerice znači da se neočekivani rast jedne liste može kompenzirati istovremenim smanjivanjem nekih drugih listi unutar istog programa.

Uzevši u obzir sve navedene prednosti i mane, zaključujemo da je implementacija liste pomoću pointera pogodna pod sljedećim uvjetima:

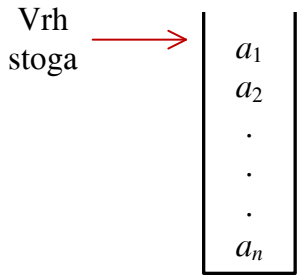
- Ima mnogo ubacivanja/izbacivanja u listu.
- Duljina liste jako varira.
- Nije nam potrebna operacija `LiPrevious( )`.

2.2 Stog

U ovom odjeljku proučavamo jednu posebnu vrstu liste koja se naziva stog. Posebnost je u načinu ubacivanja i izbacivanja elemenata. Stog je važan zato jer se pojavljuje u važnim primjenama te zato jer se zbog svoje posebnosti može efikasnije implementirati nego općenita lista. U odjeljku najprije navodimo svojstva i primjene stoga, a zatim se bavimo njegovim implementacijama.

2.2.1 Svojstva i primjene stoga

Stog je posebna vrsta liste koju zamišljamo u skladu sa slikom 2.5. Dakle, riječ je o listi gdje se sva ubacivanja i izbacivanja obavljaju na jednom kraju koji se zove **vrh**. Zahvaljujući takvom načinu rada, element koji je zadnji bio ubačen prvi je na redu za izbacivanje. Ili općenitije, redoslijed izbacivanja suprotan je od redoslijeda ubacivanja. Stog se također naziva i *stack*, LIFO lista (*last-in-first-out*) te *pushdown* lista.



Slika 2.5: Ograničene mogućnosti rada sa stogom.

U nastavku slijedi nekoliko primjera korištenja stoga. Neki od njih su iz običnog života, a neki su usko vezani uz računarstvo.

- U restoranu se čisti tanjuri slažu u "toranj". Djelatnik koji pere tanjure, bojeći se da ne sruši toranj, novooprane tanjure stavlja na vrh. Iz istih razloga, konobar tanjure koje će odnijeti gostima skida s vrha. Dakle, toranj tanjura ponaša se kao stog. Zadnje oprani tanjur prvi je na redu da ide gostu na stol. Posljedica je da će gosti dobivati vlažne i tople tanjure.

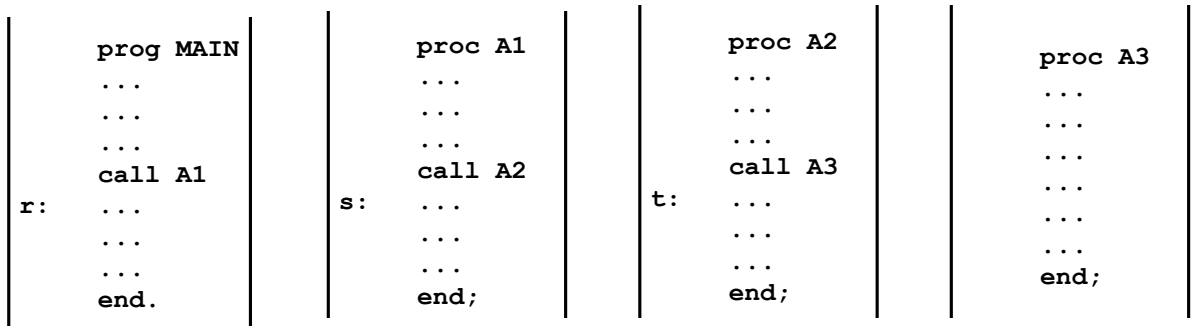
- Kod većine pištolja postoji potisni spremnik za metke. Novi metak koji se odozgo umeće u spremnik potiskuje prije umetnute metke dublje u unutrašnjost. Pištolj uvijek ispaljuje metak s vrha spremnika, dakle onaj koji je zadnji bio utisnut. Nakon ispaljivanja, ostali metci u spremniku pomiču se za jedno mjesto prema vrhu. Kod rafalne paljbe metci izlijeću u redoslijedu suprotnom od redoslijeda kojim su bili utisnuti. Očito, pištolj se ponaša kao stog.

- U većini programskih jezika, program se sastoji od više dijelova (procedura). Glavni program poziva potprograme, a potprogrami pozivaju svoje potprograme. Da bi se takav način rada ispravno odvijao, potrebno je pamtiti adrese povratka u nadređenu proceduru. Mnogi jezici (kao npr. C, Java, Pascal) u tu svrhu koriste stog. Kod ulaska u novu proceduru na stog se stavlja adresa povratka u nadređenu proceduru. Kad procedura završi rad, uzima se adresa s vrha stoga te se na nju prenosi kontrola. Jedna moguća konfiguracija prikazana je na slici 2.6. Zbog jednostavnosti pretpostavljeno je da potprogrami nemaju argumenata ni povratnih vrijednosti. Sadržaj stoga u trenutku ulaska u zadnji potprogram vidi se na slici 2.7. Adresa `q` na slici 2.7 je adresa povratka u operacijski sustav.

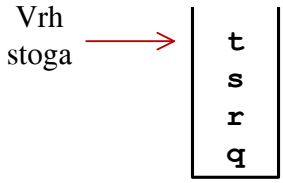
- Primijetimo da se u prethodnom primjeru na stog mogu stavljati i adrese iz istog potprograma. To u principu omogućuje rekurziju. No da bi različiti pozivi istog rekurzivnog programa mogli koristiti istu kopiju strojnog koda, nužno je da se na stog uz adresu povratka stavlja i cjelokupni "kontekst" svakog poziva, dakle vrijednosti svih argumenata i lokalnih varijabli te povratna vrijednost. Znači, rekurzija se relativno jednostavno može realizirati na načunalu, a za tu realizaciju potreban je stog. Pritom se rekurzivni postupak pretvara u neku vrstu iteracije (petlje). Dakle vrijedi krilatica: "rekurzija = stog + iteracija".

- Većina programskih jezika omogućuje rad s aritmetičkim izrazima. Interpreter takvog jezika (npr. BASIC-interpreter) mora biti u stanju izračunati vrijednost zadanog izraza. Interpreteru je jednostavnije ako za zapisivanje izraza umjesto konvencionalnog načina koristimo tzv. postfix notaciju, gdje operatori slijede iza operanada te nema potrebe za zagradama. Primjerice, umjesto konvencionalnog $(A/(B \uparrow C))*D+E$ pišemo $ABC \uparrow /D*E+$. Računanje se tada provodi čitanjem postfix izraza s lijeva na desno. Kad susretne operand, interpreter ga stavlja na stog. Kad susretne operator, interpreter skida sa stoga onoliko operanada koliko taj operator traži, obavlja operaciju i rezultat vraća na stog. Na kraju postupka, izračunata vrijedost izraza nalazi se na vrhu stoga.

Iz navedenih primera vidljivo je da je stog izuzetno važan posebni slučaj liste. Za računarstvo su osobito zanimljiva posljednja dva primjera vezana uz realizaciju rekurzije odnosno računanje aritmetičkih izraza. Zbog svoje primjenjivosti i čestog korištenja, stog se također promatra i kao posebni apstraktni tip. Evo definicije.



Slika 2.6: Pozivi potprograma unutar programa, adrese povratka.



Slika 2.7: Sadržaj stoga prilikom poziva potprograma sa prethodne slike.

Apstraktni tip podataka Stack

`elementtype` ... bilo koji tip.

`Stack` ... podatak tipa `Stack` je konačan niz podataka tipa `elementtype`.

`StMakeNull(&S)` ... funkcija pretvara stog `S` u prazan stog.

`StEmpty(S)` ... funkcija vraća "istinu" ako je `S` prazan stog. Inače vraća "laž".

`StPush(x,&S)` ... funkcija ubacuje element `x` na vrh stoga `S`. U terminima operacija s listama, to je ekvivalentno s `LiInsert(x,LiFirst(S),&S)`.

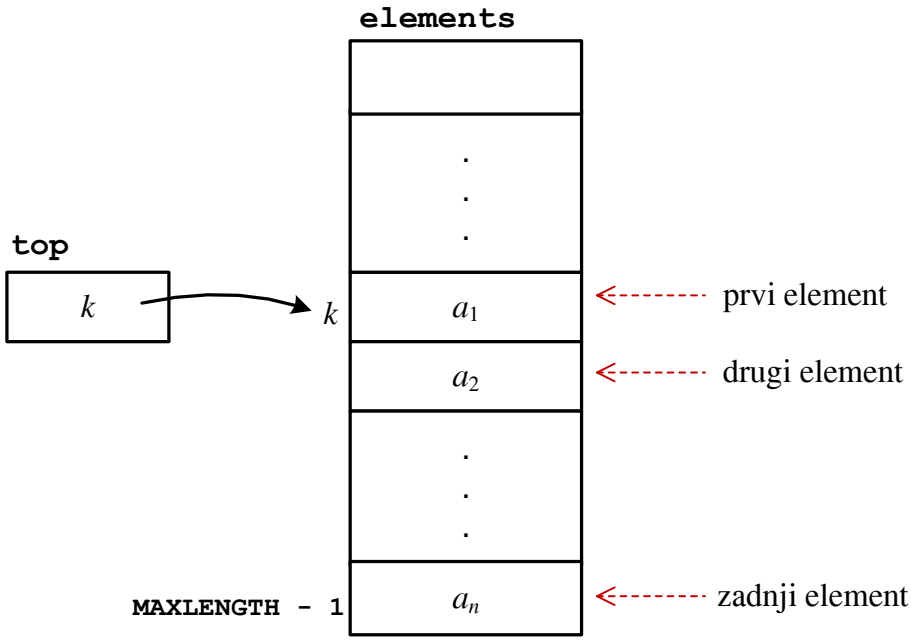
`StPop(&S)` ... funkcija izbacuje element s vrha stoga `S`. To je ekvivalentno s `LiDelete(LiFirst(S),&S)`.

`StTop(S)` ... funkcija vraća element koji je na vrhu stoga `S` (stog ostaje nepromijenjen). To je ekvivalentno s `LiRetrieve(LiFirst(S),S)`.

Svaka implementacija liste može se upotrijebiti i kao implementacija stoga. Štoviše, budući da se sa stogom obavljaju jednostavnije operacije nego s općenitom listom, promatrane implementacije mogu se pojednostaviti i učiniti efikasnijima. U ostatku ovog odjeljka proučit ćemo dvije implementacije stoga koje su dobivene preradom implementacija liste iz odjeljka 2.1. Prva od njih koristi polje, a druga pointere.

2.2.2 Implementacija stoga pomoću polja

Ova se implementacija zasniva na strukturi podataka sa slike 2.8. Riječ je o onoj istoj strukturi koju smo već opisali za općenitu listu, no s jednom malom modifikacijom. Elementi stoga spremljeni su u uzastopnim klijetkama polja `elements[ ]` duljine `MAXLENGTH`, gdje je `MAXLENGTH` unaprijed odabrana konstanta. Također imamo kursor `top` koji pokazuje gdje se u polju nalazi vrh stoga. Da prilikom ubacivanja/izbacivanja ne bi morali prepisivati ostale elemente, niz podataka umjesto u "gornji" smještamo u "donji" dio polja. Stog raste "prema gore", dakle prema manjim indeksima polja. Duljina stoga ograničena je veličinom polja, dakle konstantom `MAXLENGTH`.



Slika 2.8: Implementacija stoga pomoću polja - korištena struktura podataka.

Slijedi zapis implementacije u C-u. Najprije navodimo definicije tipova, a zatim programski kod za potrebnih pet funkcija.

```
#define MAXLENGTH ...  /* pogodna konstanta */

typedef struct {
   int top;
   elementtype elements[MAXLENGTH];
} Stack;

void StMakeNull (Stack *Sp) {
  Sp->top = MAXLENGTH;
}
```

```
int StEmpty (Stack S) {
  if (S.top >= MAXLENGTH)
    return 1;
  else
    return 0;
}

void StPush (elementtype x, Stack *Sp) {
  if (Sp->top == 0)
    exit(201) /* stog se prepunio */
  else {
    Sp->top--;
    Sp->elements[Sp->top] = x;
  }
}

void StPop (Stack *Sp) {
  if (StEmpty(*Sp))
    exit(202); /* stog je prazan */
  else
    Sp->top++;
}

elementtype StTop (Stack S) {
  if (StEmpty(S))
    exit(202); /* stog je prazan */
  else
    return (S.elements[S.top]);
}
```

Kao što vidimo, definicije tipova su slične kao kod implementacije liste pomoću polja. Najprije se izabire konkretna vrijednost za konstantu `MAXLENGTH`. Struktura podataka sa slike 2.8 definira se kao C-ov `struct` koji se sastoji od kursora `top` i polja `elements[ ]` duljine `MAXLENGTH` s elementima tipa `elementtype`.

Funkcija `StPush( )` smanjuje kursor `top` za 1 (pomiče vrh stoga prema gore) te nakon toga u polje `elements[ ]` na mjesto gdje sada pokazuje `top` upisuje novi element. Taj element je dakle ubačen na vrh stoga. Slično, funkcija `StPop( )` povećava kursor `top` za 1 (pomiče vrh stoga prema dolje). Time se element koji je do tada bio na vrhu stoga ne može više dohvatiti, a to je isto kao da smo ga izbacili. Funkcija `StTop( )` jednostavno pročita iz polja `elements[ ]` vrijednost s indeksom `top`.

Lako se uvjeriti da u praznom stogu kursor `top` ima vrijednost `MAXLENGTH`, tj. on pokazuje ispod zadnjeg elementa u polju `elements[ ]`. U skladu s time, funkcija `StMakeNull( )` pridružuje kursoru `top` vrijednost `MAXLENGTH`, a funkcija `StEmpty( )` provjerava je li `top` veći ili jednak `MAXLENGTH`.

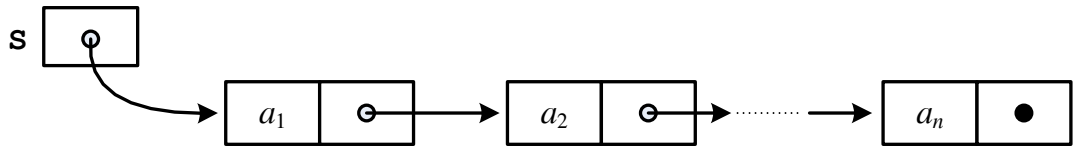
Slično kao u implementaciji liste pomoću polja, i u ovoj implemantaciji postoje izvanredne situacije kad funkcije ne mogu izvršiti svoj zadatak pa zato prekidaju program. Ovisno o vrsti izvanredne situacije, tada se operacijskom sustavu dojavljuje jedan od

statusnih kodova 201 ili 202. Pritom 201 znači da se stog prepunio (u polju nema mjesta za ubacivanje novog elementa), dok 202 znači da je stog prazan a pokušali smo iz njega izbaciti ili pročitati element.

Implementacija stoga pomoću polja vrlo je efikasna zato jer svaku od operacija sa stogom izvršava u vremenu $\mathcal{O}(1)$. Ipak, implementacija ima jednu značajnu manu, a to je mogućnost prepunjenja stoga ako je konstanta `MAXLENGTH` premala.

2.2.3 Implementacija stoga pomoću pointera

Ova se implementacija zasniva na vezanoj listi prikazanoj na slici 2.9. Riječ je o sličnoj strukturi kakvu smo već u odjeljku 2.1 koristili za općenite liste, no s malim modifikacijama. Dakle, stog se prikazuje nizom klijetki, svaka klijetka sadrži jedan element stoga i pointer na istu takvu klijetku koja sadrži idući element. Budući da kod stoga ne postoji pojam "pozicije", nije nam više potrebna polazna klijetka (zaglavlje), već je dovoljan pointer na prvu klijetku. Time se struktura pojednostavnjuje. Vrh stoga je na početku vezane liste. Sam stog poistovjećuje se s pointerom na početak vezane liste.



Slika 2.9: Implementacija stoga pomoću pointera - korištena struktura podataka.

Slijedi zapis implementacije stoga pomoću pointera u C-u. Opet najprije navodimo potrebne definicije tipova, a zatim svih pet funkcija.

```
typedef struct celltag {
   elementtype element;
   struct celltag *next;
} celltype;

typedef celltype *Stack;

void StMakeNull (Stack *Sp) {
  *Sp = NULL;
}

int StEmpty (Stack S) {
  if (S == NULL)
    return 1;
  else
    return 0;
}
```

```
void StPush (elementtype x, Stack *Sp) {
  celltype *temp;
  temp = *Sp;
  *Sp = (celltype*) malloc(sizeof(celltype));
  (*Sp)->element = x;
  (*Sp)->next = temp;
}

void StPop (Stack *Sp) {
  celltype *temp;
  if (StEmpty(*Sp))
    exit(202); /* stog je prazan */
  else {
    temp = *Sp;
    *Sp = (*Sp)->next;
    free(temp);
  }
}

elementtype StTop (Stack S) {
  if (StEmpty(S))
    exit(202); /* stog je prazan */
  else
    return (S->element);
}
```

Kao što vidimo, programski kod dosta liči na onaj za implementaciju liste pomoću pointera iz pododjeljka 2.1.3. Definicije tipova gotovo su iste. Opet se definira tip `celltype` za klijetke od kojih se gradi vezana lista. Zatim se tip `Stack` poistovjećuje s pointerom na `celltype`.

Funkcija `StPush( )` odnosno `StPop( )` liči na `LiInsert( )` odnosno `LiDelete( )` iz pododjeljka 2.1.3. Dakle `StPush( )` fizički stvara novu klijetku i uključuje je u vezanu listu. `StPop( )` isključuje klijetku iz vezane liste te je fizički razgrađuje. Mala razlika je jedino u tome što funkcije predviđene za listu rade na sredini vezane liste, dakle između postojećih klijetki, a funkcije za stog rade na početku vezane liste.

Ostale tri funkcije vrlo su jednostavne. Budući da je u ovoj implementaciji prazan stog prikazan pointerom `NULL`, poziv oblika `StMakeNull(&S)` pridružuje `S = NULL`, a `StEmpty(S)` svodi se na provjeru je li `S` jednako `NULL`. Poziv `StTop(S)` vraća element koji piše u klijetki koju pokazuje `S`.

Za funkcije `StPop(S)` i `StTop(S)` prazan stog je izvanredna situacija. U slučaju praznog stoga te dvije funkcije prekidaju program i vraćaju operacijskom sustavu statusni kod 202.

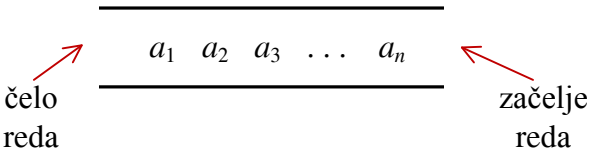
Implementacija stoga pomoću pointera ponovo je vrlo efikasna jer se sve njezine funkcije izvršavaju u vremenu $\mathcal{O}(1)$. U odnosu na implementaciju pomoću polja ima jednu malu manu, a to je da troši više memorije po elementu stoga. S druge strane, prednost joj je da uglavnom zaobilazi problem prepunjenja, budući da se oslanja na dinamičku alokaciju memorije a ne na unaprijed rezervirani memorijski prostor.

2.3 Red

U ovom odjeljku proučavamo red. Riječ je o još jednoj posebnoj vrsti liste koja se odlikuje posebnim načinom ubacivanja i izbacivanja elemenata. Slično kao stog, i red je važan zato što se pojavljuje u važnim primjenama te zato što se može efikasno implementirati. Odjeljak započinjemo navođenjem svojstava i primjena reda, a u nastavku opisujemo dvije njegove implementacije.

2.3.1 Svojstva i primjene reda

Red je posebna vrsta liste koju zamišljamo u skladu sa slikom 2.10. Elementi se ubacuju na jednom kraju te liste (začelje), a izbacuju na suprotnom kraju (čelo). Dakle, element koji je prvi bio ubačen prvi je na redu za izbacivanje. Ili općenitije, redoslijed izbacivanja poklapa se s redoslijedom ubacivanja. Red se također naziva i *queue* ili FIFO lista (*first-in-first-out*).



Slika 2.10: Ograničene mogućnosti rada s redom.

U nastavku, evo nekoliko primjera za red. Neki od njih su opet iz običnog života, a ostali su vezani uz računarstvo.

- Ljudi koji stoje pred blagajnom za kupnju kino-ulaznica oblikuju red. Novopridošla osoba priključuje se na začelje reda. Osoba na čelu reda upravo kupuje ulaznicu. Pod pretpostavkom da nema preguravanja, ljudi će kupiti ulaznice točno onim redom kako su dolazili pred blagajnu.

- Revolver je vrsta pištoja gdje je spremnik za metke građen u obliku okretnog bubnja. Da bismo napunili pištolj, okrećemo bubanj i redom umećemo metke u njegove rupe. Metci se ispaljuju onim redoslijedom kako su bili umetnuti. Znači, revolver se ponaša kao red za metke.

- Lokalna mreža ima samo jedan štampač, koji odjednom može štampati samo jednu datoteku. Da bi korisnik mogao poslati datoteku na štampanje čak i onda kad je štampač zauzet, u računalu koje upravlja štampačem formira se red. Štampač štampa datoteku sa čela tog reda. Korisnik šalje datoteku na začelje reda. Datoteke se štampaju onim redoslijedom kako su bile poslane. Situacija je ilustrirana slikom 2.11.

- Primjeri slični prethodnom pojavljuju se u računalima ili mrežama uvijek onda kad postoji jedan resurs, a više potencijalnih korisnika. Primjerice, kod starih *mainframe* računala korisnici su svoje podatke pohranjivali na magnetske trake. Računalo je obično imalo samo jednu jedinicu za rad s magnetskom trakom. Budući da je jedinica odjednom mogla raditi samo s jednom trakom, zahtjevi

korisnika stavljali su se u red. Tehničar koji je posluživao računalo dobivao je preko svoje konzole upute od računala da skine jednu traku s jedinice te stavi drugu traku. Redoslijed obrade poklapao se s redoslijedom primanja zahtjeva.

- Većina današnjih računalnih program je interaktivna, dakle oni za vrijeme svojeg rada komuniciraju s korisnikom. No velike obrade podataka koje traju satima obično se odvijaju na neinteraktivan način, tako da program čita podatke iz unaprijed pripremljene ulazne datoteke te ispisuje rezultate u izlaznu datoteku. Ovakve dugotrajne poslove računalo izvodi u "pozadini", dakle bez sudjelovanja korisnika - to se zove paketna (*batch*) obrada. Da paketna obrada ne bi zauzela previše resursa i usporila interaktivni rad, postavlja se ograničenje na broj poslova u pozadini koji istovremeno mogu biti aktivni. Primjerice, taj broj mogao bi biti dva. Koordinacija izvršavanja većeg broja poslova postiže se uz pomoć reda. Korisnici svoje nove poslove stavljaju na začelje. Operacijski sustav najprije skida dva posla sa čela te ih pokreće. Čim jedan od tih poslova završi rad, operacijski sustav skida sa čela i pokreće sljedeći posao, itd.

Slika 2.11: Niz datoteka u redu za štampanje.

Iz navedenih primjera vidljivo je da je red, slično kao stog, važan i koristan posebni slučaj liste koji zaslužuje da ga se promatra kao kao posebni apstraktni tip podataka. Slijedi definicija.

Apstraktni tip podataka Queue

`elementtype` ... bilo koji tip.

`Queue` ... podatak tipa `Queue` je konačni niz podataka tipa `elementtype`.

`QuMakeNull(&Q)` ... funkcija pretvara red `Q` u prazan red.

`QuEmpty(Q)` ... funkcija vraća "istinu" ako je `Q` prazan red, inače "laž".

`QuEnqueue(x,&Q)` ... funkcija ubacuje element `x` na začelje reda `Q`. U terminima a.t.p.-a `List`, to je ekvivalentno s `LiInsert(x,LiEnd(Q),&Q)`.

`QuDequeue(&Q)` ... funkcija izbacuje element sa čela reda `Q`. To je ekvivalentno s `LiDelete(LiFirst(Q),&Q)`.

`QuFront(Q)` ... funkcija vraća element na čelu reda `Q` (red ostaje nepromjenjen). To je ekvivalentno s `LiRetrieve(LiFirst(Q),Q)`.

Implementacije reda opet se mogu dobiti iz implementacija liste, uz odgovarajuća pojednostavnjenja ili modifikacije. U ostatku ovog odjeljka proučit ćemo dvije implementacije reda koje su dobivene preradom i dotjerivanjem implementacija liste iz odjeljka 2.1. Prva od njih koristi polje, a druga pointere.

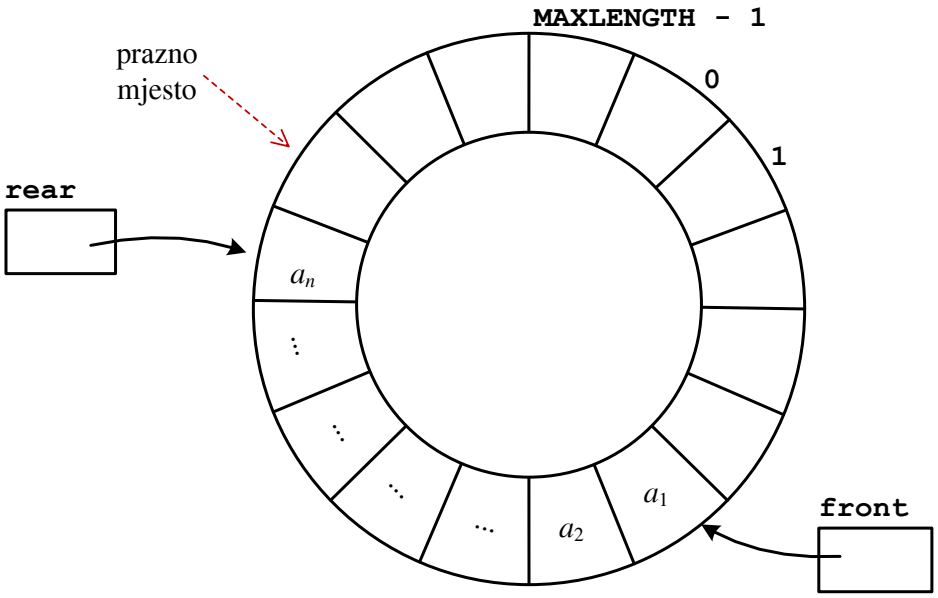
2.3.2 Implementacija reda pomoću cirkularnog polja

Mogli bismo doslovno preuzeti implementaciju liste pomoću polja iz pododjeljka 2.1.2, s time da početak polja bude čelo reda. Operacija `QuEnqueue( )` tada se obavlja u konstantnom vremenu, budući da ne zahtijeva pomicanje drugih elemenata polja. No, `QuDequeue( )` nije jednako efikasna, jer zahtjeva da se cijeli ostatak polja prepiše za jedno mjesto "gore".

Tome možemo doskočiti tako da dozvolimo da se izbacivanjem elemenata čelo pomiče prema dolje. Uvodimo još jedan kursor koji pamti trenutni položaj čela. Tada više nema prepisivanja, no pojavila se nova nepogodnost: ubacivanjem i izbacivanjem red "putuje" prema "donjem kraju" polja. Dosegnut ćemo donji kraj, makar u početnom dijelu polja ima mjesta.

Još bolje rješenje je cirkularno polje prikazano slikom 2.12. Dakle imamo polje `elements[ ]` s elementima tipa `elementtype` duljine `MAXLENGTH`, gdje je `MAXLENGTH` unaprijed odabrana konstanta. Cirkularnost znači da zamišljamo da nakon zadnjeg indeksa ponovo slijedi početni indeks - to je zorno prikazano na slici. Red zauzima niz uzastopnih klijetki polja. Postoji kursor `front` koji pokazuje gdje je čelo te kursor `rear` koji pokazuje začelje. Uslijed ubacivanja i izbacivanja elemenata red "putuje" poljem u smjeru kazaljke na satu.

Ako detaljnije analiziramo strukturu sa slike 2.12, primijetit ćemo da kod nje postoji jedan vid dvoznačnosti u prikazu. Naime, prazni red (onaj koji nema ni jedan element) i posve puni red (onaj koji ima `MAXLENGTH` elemenata) prikazani su na sasvim isti način. U oba slučaja `rear` pokazuje jedno mjesto ispred mjesta koje pokazuje `front`. Da bismo ipak razlikovali slučaj praznog i posve punog reda, dogovorit ćemo se da u polju između čela i začelja mora biti bar jedno prazno mjesto. Znači, zahtijevat ćemo da red nikad ne smije imati više od `MAXLENGTH - 1` elemenata. Tada situacija kad `rear` pokazuje jedno mjesto ispred `front` znači da je red prazan.



Slika 2.12: Implementacija reda pomoću cirkularnog polja - struktura podataka.

Slijedi zapis implementacije reda pomoću cirkularnog polja u jeziku C. Nakon potrebnih definicija tipova, navedena je jedna pomoćna funkcija te pet funkcija koje odgovaraju operacijama iz apstraktnog tipa Queue.

```
#define MAXLENGTH ...  /* dovoljno velika konstanta */

typedef struct {
   elementtype elements[MAXLENGTH];
   int front, rear;
} Queue;

int addone (int i) {
  return ((i+1) % MAXLENGTH);
}

void QuMakeNull (Queue *Qp) {
  Qp->front = 0;
  Qp->rear = MAXLENGTH-1;
}

int QuEmpty (Queue Q) {
  if (addone(Q.rear) == Q.front) return 1;
  else return 0;
}

void QuEnqueue (elementtype x, Queue *Qp) {
  if (addone(addone(Qp->rear)) == (Qp->front))
    exit(301); /* red se prepunio */
  else {
    Qp->rear = addone(Qp->rear);
    Qp->elements[Qp->rear] = x;
  }
}

void QuDequeue (Queue *Qp) {
  if (QuEmpty(*Qp))
    exit(302) /* red je prazan */
  else
    Qp->front = addone(Qp->front);
}

elementtype QuFront (Queue Q) {
  if (QuEmpty(Q))
    exit(302) /* red je prazan */
  else
    return (Q.elements[Q.front]);
}
```

Definicije tipova slične su kao kod implementacije liste pomoću polja. Najprije se zadaje vrijednost za konstantu `MAXLENGTH`. Struktura sa slike 2.12 definira se kao C-ov `struct` koji se sastoji od kursora `front` i `rear` te polja `elements[ ]` duljine `MAXLENGTH`.

Primijetimo da je polje u unutar naše strukture zapravo sasvim obično (necirkularno) polje. Ipak, cirkularnost se postiže tako da s indeksima računamo isključivo u modulo aritmetici. Naime, bilo koja izračunata vrijednost za indeks zamjenjuje se s ostatkom kod dijeljenja te vrijednosti s `MAXLENGTH`.

Modulo aritmetika implementirana je pomoćnom funkcijom `addone( )`. Ona zadanu cjelobrojnu vrijednost povećava za 1 u smislu modulo `MAXLENGTH`. Dakle, ako krenemo od 0, uzastopni pozivi `addone( )` proizvode niz vrijednosti 1, 2, ..., `MAXLENGTH - 1`, a zatim opet 0, 1, 2, ..., itd. Funkcija `addone( )` poziva se u ostalim funkcijama. Njome se kursori `front` i `rear`, koji prema slici 2.12 pokazuju u polje `elements[ ]`, pomiču za jedno mjesto dalje u smjeru kazaljke na satu.

Rekli smo da se u našoj implementaciji prazni red prepoznaje po tome što kod njega `rear` pokazuje jedno mjesto ispred mjesta koje pokazuje `front`. U skladu s time, funkcija `QuMakeNull( )` inicijalizira `rear` i `front` na vrijednosti `MAXLENGTH - 1` i 0. Slično, `QuEmpty( )` provjerava da li bi pomicanjem `rear` za jedno mjesto dalje došli na isto mjesto koje već pokazuje `front`.

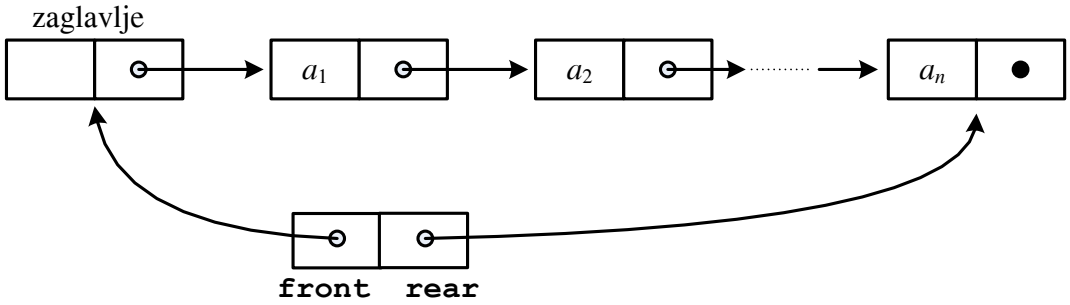
Funkcija `QuEnqeue( )` pomiče `rear` za jedno mjesto dalje te na to novo mjesto koje sad postaje novo začelje reda upisuje novi element. `QuDequeue( )` pomiče `front` za jedno mjesto dalje, pa time element koji je bio na čelu isključuje iz reda. `QuFront( )` čita element koji pokazuje `front` jer taj element se nalazi na čelu reda.

Slično kao u nekim prethodnim implementacijama, i u ovoj implementaciji postoje izvanredne situacije kad funkcije ne mogu obaviti svoj zadatak pa prekidaju program. Prva takva situacija je prepunjenje reda prilikom ubacivanja novog elementa, a druga je pokušaj čitanja ili izbacivanja elementa iz praznog reda. Statusni kod koji se u prvom odnosno drugom slučaju vraća operacijskom sustavu je 301 odnosno 302. U skladu s našim dogovorom o izbjegavanju dvoznačnosti prikaza, prepunjenje reda nastupa već onda kad je u polju u trenutku ubacivanja elementa ostalo samo jedno prazno mjesto. Provjera postoji li samo jedno prazno mjesto svodi se na provjeru bi li pomicanjem `rear` za *dva* mjesta došli na isto mjesto koje već pokazuje `front`.

Implementacija reda pomoću cirkularnog polja vrlo je efikasna u smislu vremena izvršavanja operacija, naime sve operacije izvršavaju se u vremenu $\mathcal{O}(1)$. Kao i kod svih drugih implementacija zasnovanih na polju, glavna mana joj je mogućnost prepunjenja reda.

2.3.3 Implementacija reda pomoću pointera

Implementacija se zasniva na strukturi podataka prikazanoj na slici 2.13. Riječ je o varijanti vezane liste iz odjeljka 2.1. Dakle, red se prikazuje nizom klijetki. Svaka klijetka sadrži jedan element reda i pointer na istu takvu klijetku koja sadrži idući element. Opet postoji zaglavlje (prazna klijetka) na početku vezane liste koja olakšava prikaz praznog reda. Čelo reda nalazi se na početku vezane liste, a začelje na suprotnom kraju. Sam red poistovjećuje se s parom pointera `front` i `rear`. Pritom `front` pokazuje početak vezane liste, dakle čelo reda, a `rear` kraj vezane liste, dakle začelje.



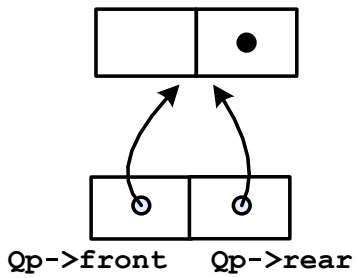
Slika 2.13: Implementacija reda pomoću pointera - korištena struktura podataka.

Slijedi zapis implementacije reda pomoću pointera u programskom jeziku C. Detalji u radu funkcija `QuMakeNull( )`, `QuEnqueue( )` i `QuDequeue( )` pojašnjeni su slikama 2.14, 2.15 i 2.16. Na tim slikama pune strelice opet označavaju polaznu situaciju, a crtkane strelice učinak funkcije.

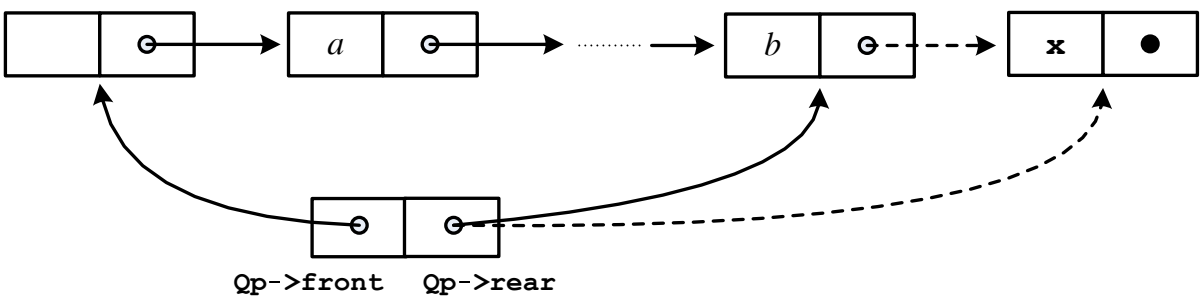
```
typedef struct celltag {
   elementtype element;
   struct celltag *next;
} celltype;

typedef struct {
   celltype *front, *rear;
} Queue;

void QuMakeNull (Queue *Qp) {
  Qp->front = (celltype*)malloc(sizeof(celltype));
  Qp->front->next = NULL;
  Qp->rear = Qp->front;
}

int QuEmpty (Queue Q) {
  if (Q.front == Q.rear)
    return 1;
  else
    return 0;
}

elementtype QuFront (Queue Q) {
  if (QuEmpty(Q))
    exit(302) /* red je prazan */
  else
    return (Q.front->next->element);
}
```

```
void QuEnqueue (elementtype x, Queue *Qp) {
  Qp->rear->next = (celltype*)malloc(sizeof(celltype));
  Qp->rear = Qp->rear->next;
  Qp->rear->element = x;
  Qp->rear->next = NULL;
}

void QuDequeue (Queue *Qp) {
  celltype *temp;
  if (QuEmpty(*Qp))
    exit(302) /* red je prazan */
  else {
    temp = Qp->front;
    Qp->front = Qp->front->next;
    free(temp);
  }
}
```

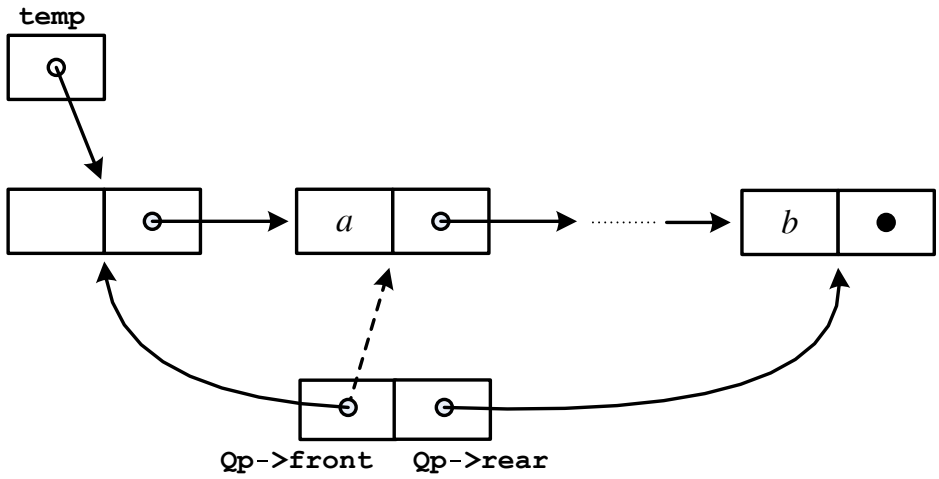
Kao što vidimo, implementacija započinje definicijom tipa `celltype` za klijetke od kojih se gradi naša vezana lista. Tip `Queue` zatim se poistovjećuje sa `struct`-om koji sadrži pointere `front` i `rear` - oba pointera pokazuju na `celltype`.



Slika 2.14: Implementacija reda pomoću pointera - operacija `QuMakeNull( )`.



Slika 2.15: Implementacija reda pomoću pointera - operacija `QuEnqueue( )`.



Slika 2.16: Implementacija reda pomoću pointera - operacija `QuDequeue( )`.

Budući da je u našoj implementaciji prazan red prikazan vezanom listom koja se sastoji samo od zaglavlja, funkcija `QuMakeNull( )` fizički stvara to zaglavlje i postavlja pointere `front` i `rear` da oba pokazuju na njega. Fukcija `QuEmpty( )` prepoznaje prazan red po tome što oba pointera `front` i `rear` pokazuju na istu adresu, tj. na zaglavlje.

Funkcija `QuEnqueue( )` fizički stvara novu klijetku te je preko pointera `rear` uključuje na kraj vezane liste. `QuDequeue( )` mijenjanjem pointera `front` isključuje prvu klijetku iz vezane liste te je fizički razgrađuje - time iduća klijetka koja je do tada sadržavala čelni element reda preuzima ulogu zaglavlja. `QuFront( )` nalazi i čita čelni element reda tako da preko pointera `front` dođe do zaglavlja, a zatim iz zaglavlja pročita adresu iduće klijetke u vezanoj listi. Zadnje dvije funcije u slučaju praznog reda prekidaju program i vraćaju statusni kod 302.

Implementacija reda pomoću pointera podjednako je efikasna kao i implementacija pomoću polja jer isto izvršava sve operacije nad redom u vremenu $\mathcal{O}(1)$. Makar troši malo više memorije po elementu, njezina prednost je da uglavnom zaobilazi problem prepunjenja reda.

Poglavlje 3

STABLA

Ovo poglavlje bavi se skupinom sličnih apstraktnih tipova podataka koje jednom rječju nazivamo stablima. Za razliku od lista, zasnovanih na linearnom uređaju podataka, stabla se zasnivaju na hijerarhijskom uređaju. Dakle, među podacima postoji odnos "nadređeni-podređeni" ili "roditelj-dijete". Poglavlje se sastoji od dva odjeljka od kojih svaki obrađuje jednu vrstu stabala. Najprije govorimo o uređenom stablu, a zatim o binarnom stablu.

3.1 Uređeno stablo

U ovom odjeljku proučavamo jednu prilično općenitu vrstu stabala koja se zove *uređeno* stablo. Riječ je o apstraktnom tipu podataka gdje podaci čine hijerarhiju. Pritom jednom nadređenom podatku može biti pridruženo nula, jedan ili više podređenih podataka. Za te podređene podatke zadan je redoslijed, to jest nije svejedno koji od njih je prvi, koji drugi, itd. U odjeljku najprije navodimo svojstva i primjene uređenog stabla, zatim opisujemo algoritme njegovog obilaska, a na kraju raspravljamo o dvije implementacije.

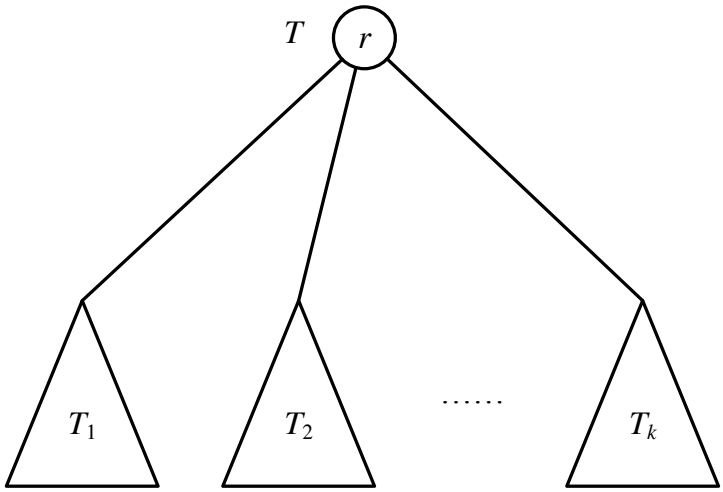
3.1.1 Svojstva i primjene uređenog stabla

Uređeno stablo je tvorevina koja se osim u računarstvu također proučava u diskretnoj matematici. Stroga matematička definicija glasi ovako. **Uređeno stablo** (*ordered tree*) T je neprazni konačni skup podataka istog tipa koje zovemo **čvorovi**. Pritom:

- Postoji jedan istaknuti čvor r koji se zove **korijen** od T.
- Ostali čvorovi grade konačni niz $(T_1, T_2, \ldots, T_k)$ od 0, 1 ili više disjunktnih (manjih) uređenih stabala.

Ova definicija ilustrirana je slikom 3.1. Ako nije drukčije navedeno, pod pojmom "stablo" u nastavku ćemo podrazumijevati uređeno tablo.

Važno je uočiti da je definicija uređenog stabla rekurzivna, jer se osim na T primjenjuje i na manja stabla $T_1, T_2, \ldots, T_k$. Ta manja stabla zovu se **podstabla** korijena r. Korijeni $r_1, r_2, \ldots, r_k$ od $T_1, T_2, \ldots, T_k$ su **djeca** od r, a r je njihov **roditelj**. Naravno, svako od podstabala $T_1, T_2, \ldots, T_k$ može imati svoja još manja pod-podstbla, ona opet svoja pod-pod-podstabla, itd. Dosljednom primjenom rekurzije na kraju ćemo za



Slika 3.1: Uređeno stablo i njegova podstabla.

svaki čvor u stablu ustanoviti čije je on dijete te čiji je roditelj. Odnos roditelj-dijete smatramo hijerarhijskim odnosom među čvorovima, dakle roditelj je nadređen djeci, a djeca su podređena roditelju. Primijetimo da je korijen jedini čvor u stablu koji nema roditelja, a svaki od preostalih čvorova ima točno jednog roditelja.

Uređenost stabla se očituje u tome što među podstablima postoji linearan uređaj, tj. zna se koje je prvo, koje drugo, ..., itd. Taj linearni uređaj prenosi se i na djecu istog roditelja, dakle i za njih je određeno koje je prvo dijete, koje je drugo, ..., itd.

Uređeno stablo obično prikazujemo dijagramom. Na takvom dijagramu postoje kružići i spojnice (bridovi): kružić predstavlja čvor, a spojnica vezu između roditelja i djeteta. Dijagram se crta tako da najprije nacrtamo korijen, zatim njegovu djecu, zatim djecu od djece, ..., itd, sve dok ne nacrtamo sve čvorove. Roditelj se uvijek crta iznad svoje djece, a djeca istog roditelja poredana su slijeva na desno u skladu sa svojim zadanim redoslijedom. Nekoliko dijagrama za stabla vidimo na slikama 3.2 - 3.9.

Uređena stabla često se pojavljuju u raznim područjima života, svugdje gdje je potrebno opisati nekakav hijerarhijski odnos, klasifikaciju ili sistematizaciju. Slijedi nekoliko primjera.

- Uredno napisana knjiga može se shvatiti kao stablo. Poglavlja čine podstabla, odjeljci su pod-podstabla, pododjeljci pod-pod-podstabla, itd. Budući da poglavlja i ostali dijelovi imaju zadani redoslijed, riječ je zaista o *uređenom* stablu. Slika 3.2 s lijeve strane prikazuje sadržaj neke knjige koji se sastoji od tri poglavlja, s time da poglavlje 1 ima dva, a poglavlje 2 tri odjeljka, od kojih jedan ima dva pododjeljka. Dijelovi knjige numerirani su na uobičajeni način. Isti sadržaj je s desne strane slike 3.2 predočen uređenim stablom. Ovaj nastavni materijal također je organiziran u poglavlja, odjeljke i pododjeljke te bi se mogao nacrtati kao stablo sa stotinjak čvorova (pokušajte ga nacrtati za vježbu).

- Neka država sastoji se od regija, regije od županija, županije od općina. Struktura te države može se predočiti stablom, gdje je sama država korijen, regije su djeca od korijena, županije djeca od djece, a općine djeca od djece od djece. Uređenost

stabla u ovom slučaju nije nužna, no može se postići ako dijelove na istoj razini nabrajamo abecednim redom.

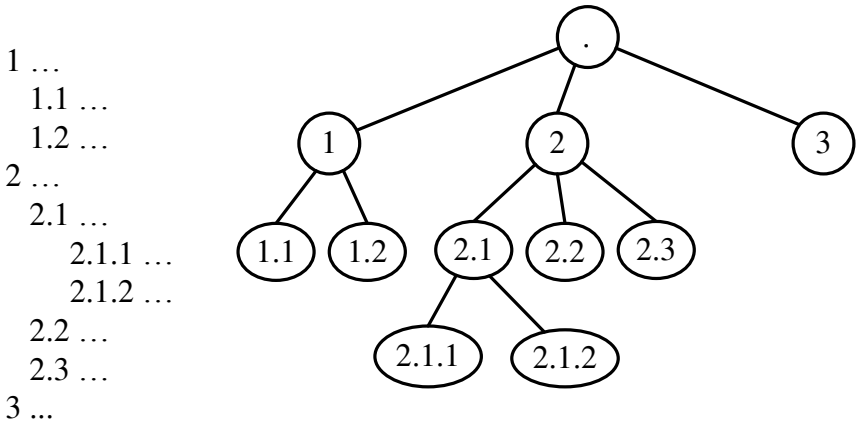
- Sveučilište u Zagrebu može se predočiti stablom. Iz tog prikaza vidjelo bi se da se Sveučilište sastoji od fakulteta i akademija, ti fakulteti ili akademije imaju svoje odsjeke, a odsjeci zavode ili katedre. Da bi stablo bilo uređeno, možemo izmisliti neki redoslijed za sastavnice na istoj razini. Primjerice, mogli bi ih poredati po datumima osnivanja.

- Poznate plemićke porodice imale su običaj crtati svoja porodična stabla. Čvorovi u tom stablu predstavljali su pojedine članove porodice, a veze roditelj-dijete njihove stvarne porodične odnose. U korijenu stabla nalazio se najstariji poznati predak, a na dnu hijerarhije bili su trenutno najmlađi članovi. Da bi svako dijete imalo samog jednog roditelja, prikazivali su se npr. samo muški preci. Uređenost stabla mogla se postići tako da su se djeca istog roditelja redala po starosti.

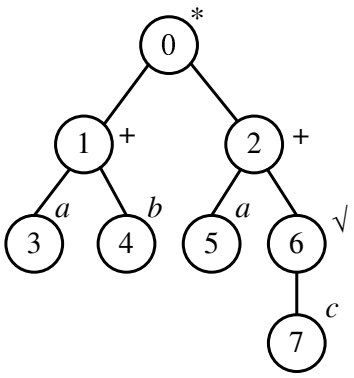
- Građa aritmetičkog izraza može se prikazati stablom. To je na slici 3.3 napravljeno za izraz $(a + b) * (a + \sqrt{c})$. Čvorovi bez djece predstavljaju operande, a ostali čvorovi računske operacije. Na slici 3.3 pored svakog čvora nalazi se oznaka koja kaže o kojem operandu ili operaciji je riječ. Primijetimo da se isti operand ili operacija može pojaviti na više mjesta. Uređenost stabla važna je ako su operacije nekomutativne. Izvrednjavanje aritmetičkog izraza ostvaruje se na sljedeći način: svaki roditelj računa međurezultat koji se dobije primjenom njegove operacije na vrijednosti pridružene njegovoj djeci. Primjerice, čvor 1 na slici 3.3 računa $a + b$, čvor 6 daje $\sqrt{c}$, a čvor 2 izvrednjava $a+\sqrt{c}$. Računanje se mora provoditi "odozdo prema gore". Na kraju će se vrijednost cijelog izraza pojaviti u korijenu.

- Bilo koji C program ima hijerarhijsku građu: on se sastoji od globalnih varijabli i funkcija, a funkcije imaju svoje lokalne varijable i naredbe. Pojedina naredba može biti složena, tj. unutar svojih vitičastih zagrada ona može sadržavati podnaredbe, a te podnaredbe mogu imati svoje pod-podnaredbe, itd. Građa C programa može se predočiti stablom gdje je korijen sam program, dijete korijena je npr. funkcija, dijete od djeteta je npr. naredba unutar funkcije, itd. Takvo stablo generira se prilikom prevođenja (kompiliranja) programa i naziva se *sintaksno* stablo. Riječ je naravno o uređenom stablu - naime program ne bi radio isto ako bi mu promijenili redoslijed naredbi.

Stablo sa slike 3.3 naziva se **označeno** stablo. Kod njega je svakom čvoru pridružen dodatni podatak koji zovemo **oznaka**. Pritom razlikujemo čvor (tj. njegovo ime) od oznake. Ime čvora služi za identifikaciju, naime u istom stablu ne mogu postojati dva čvora s istim imenom. S druge strane, oznaka čvora daje neku dodatnu informaciju, s time da više čvorova smije imati istu oznaku. Uočimo da su pojmovi: stablo, oznaka, čvor (u kontekstu stabala) redom analogni pojmovima: lista, element, pozicija (u kontekstu listi). Zaista, dok smo kod liste elemente razlikovali po njihovim pozicijama, sad kod stabla čvorove razlikujemo po njihovim imenima. Također, dok je kod liste korisna informacija bila pohranjena u vrijednostima elemenata, sad kod stabla ona se obično nalazi u oznakama čvorova. Kod crtanja dijagrama stabla držat ćemo se pravila da se ime čvora upisuje u odgovarajući kružić, a oznaka se dopisuje pored kružića.



Slika 3.2: Sadržaj knjige prikazan kao stablo.



Slika 3.3: Aritmetički izraz $(a + b) * (a + \sqrt{c})$ prikazan kao stablo.

U nastavku slijedi još nekoliko definicija kojima proširujemo terminologiju vezanu uz stabla. Niz čvorova $i_1, i_2, \ldots, i_m$ takvih da je i_p roditelj od i_{p+1} $(p = 1, \ldots, m-1)$ zove se **put** od i_1 do i_m. **Duljina** tog puta je $m - 1$. Za svaki čvor različit od korijena postoji jedinstveni put od korijena stabla do tog čvora. Ako postoji put od čvora i do čvora j tada je i **predak** od j, a j je **potomak** od i. Znači, korijen je predak za sve ostale čvorove u stablu, a oni su njegovi potomci. **Razina** ili **nivo** s je skup čvorova stabla sa svojstvom da jedinstveni put od korijena do tog čvora ima duljinu s. Razinu 0 čini sam korijen (po dogovoru). Razinu 1 čine djeca korijena, razinu 2 njihova djeca, itd. **Visina** stabla je maksimalna neprazna razina. **List** je čvor bez djece. **Unutrašnji čvor** je čvor koji nije list. Djeca istog čvora zovu se **braća**.

Iz prethodnih primjera vidjeli smo da se uređeno stablo pojavljuje u brojnim situacijama. Ako te situacije želimo informatizirati, tada se javlja potreba pohranjivanja stabala u računalu i automatskog rukovanja s njima. To znači da matematički pojam stabla moramo najprije pretvoriti u apstraktni tip podataka te ga zatim implementirati. U definiciji apstraktnog tipa potrebno je odabrati elementarne operacije koje bi se trebale obavljati nad stablom. Također, nužno je obuhvatiti oznake čvorova budući

da su one neophodne u većini primjena. U nastavku slijedi jedna od mogućih varijanti definicije apstraktnog tipa za označeno stablo.

Apstraktni tip podataka Tree

`node` ... bilo koji tip (čvorovi, tj. njihova imena). U skupu `node` uočavamo jedan poseban element `LAMBDA` (koji služi kao ime nepostojećeg čvora).

`labeltype` ... bilo koji tip (oznake čvorova).

`Tree` ... podatak tipa `Tree` je uređeno stablo čiji čvorovi su podaci tipa `node` (međusobno različiti i različiti od `LAMBDA`). Svakom čvoru je kao oznaka pridružen podatak tipa `labeltype`.

`TrMakeRoot(l,&T)` ... funkcija pretvara stablo `T` u stablo koje se sastoji samo od korijena s oznakom `l`. Vraća čvor koji služi kao korijen (tj. njegovo ime).

`TrInsertChild(l,i,&T)` ... funkcija u stablo `T` ubacuje novi čvor s oznakom `l`, tako da on bude prvo po redu dijete čvora `i` - vidi sliku 3.4. Funkcija vraća novi čvor (njegovo ime). Nije definirana ako `i` ne pripada `T`.

`TrInsertSibling(l,i,&T)` ... funkcija u stablo `T` ubacuje novi čvor s oznakom `l`, tako da on bude idući po redu brat čvora `i` - vidi sliku 3.5. Funkcija vraća novi čvor (njegovo ime). Nije definirana ako je `i` korijen ili ako `i` ne pripada `T`.

`TrDelete(i,&T)` ... funkcija izbacuje list `i` iz stabla `T`. Nije definirana ako je `i` korijen, ili ako `i` ne pripada `T` ili ako `i` ima djece.

`TrRoot(T)` ... funkcija vraća korijen od stabla `T`.

`TrFirstChild(i,T)` ... funkcija vraća prvo po redu dijete čvora `i` u stablu `T`. Ako je `i` list, vraća `LAMBDA`. Nije definirana ako `i` ne pripada `T`.

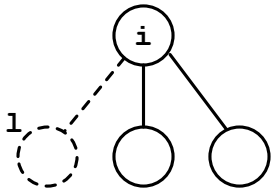
`TrNextSibling(i,T)` ... funkcija vraća idućeg po redu brata čvora `i` u stablu `T`. Ako je `i` zadnji brat, tada vraća `LAMBDA`. Nije definirana ako `i` ne pripada `T`.

`TrParent(i,T)` ... funkcija vraća roditelja čvora `i` u stablu `T`. Ako je `i` korijen, tada vraća `LAMBDA`. Nije definirana ako `i` ne pripada `T`.

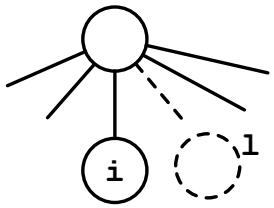
`TrLabel(i,T)` ... funkcija vraća oznaku čvora `i` u stablu `T`. Nije definirana ako `i` ne pripada `T`.

`TrChangeLabel(l,i,&T)` ... funkcija mijenja oznaku čvora `i` u stablu `T`, tako da ta oznaka postane `l`. Nije definirana ako `i` ne pripada `T`.

U zadnjem dijelu ovog odjeljka proučit ćemo dvije implementacije apstraktnog tipa Tree. Obje se zasnivaju na tome da se eksplicitno zapišu čvorovi, njihove oznake te veze među čvorovima. Razlika je u načinu zapisivanja veza: prva implementacija bilježi vezu od čvora prema njegovom roditelju, a druga vezu od čvora do prvog djeteta odnosno idućeg brata.



Slika 3.4: Učinak funkcije `TrInsertChild( )`.



Slika 3.5: Učinak funkcije `TrInsertSibling( )`.

3.1.2 Obilazak stabla

U mnogim primjenama potrebno je obaviti neku obradu nad svim čvorovima stabla, npr. ispisati ili zbrojiti oznake, naći najmanju oznaku i slično. Tada govorimo o obilasku stabla. Kada bi na sličan način htjeli obraditi podatke u listi, napravili bi sekvencijalni prolazak listom od njezinog početka do njezinog kraja. No sa stablom je kompliciranije budući da u njemu podaci nisu organizirani u niz nego u hijerarhiju. Očito, obilazak stabla može se obaviti na razne načine i za to nam je potreban algoritam koji će točno odrediti redoslijed "posjećivanja" čvorova.

Obilazak stabla definiramo kao algoritam kojim posjećujemo čvorove stabla tako da svaki čvor posjetimo točno jednom. Primjetimo da svaki obilazak uspostavlja jedan privremeni linearni uređaj među čvorovima. Najpoznatiji obilasci su: `Preorder( )`, `Inorder( )`, `Postorder( )`. Sva tri algoritma oslanjaju se na rekurzivnu građu stabla, pa su i sami zadani na rekurzivan način. U nastavku slijedi njihova specifikacija.

Neka je T stablo sastavljeno od korijena r i podstabala $T_1, T_2, \ldots, T_k$ od korijena - vidi sliku 3.1 uz definiciju stabla. Tada:

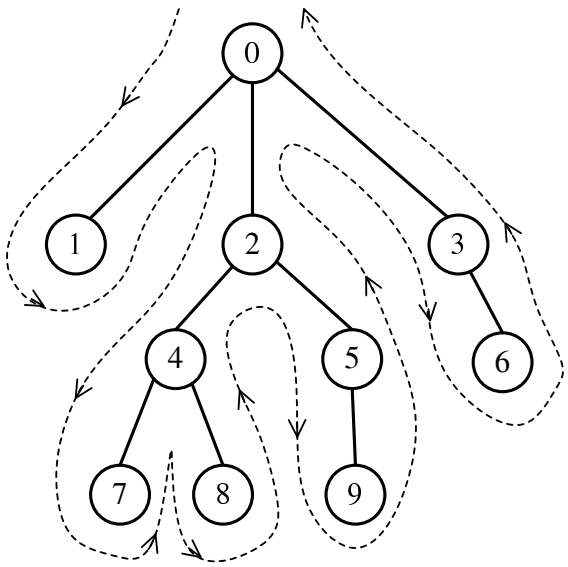
`Preorder( )` ... najprije posjećuje r, zatim obilazi T_1, zatim obilazi T_2, ..., na kraju obilazi T_k.

`Inorder( )` ... najprije obilazi T_1, zatim posjećuje r, zatim obilazi T_2, ..., na kraju obilazi T_k.

`Postorder( )` ... najprije obilazi T_1, zatim obilazi T_2, ..., zatim obilazi T_k, na kraju posjećuje r.

Razlike između triju obilazaka vidljive su iz primjera na slici 3.6. Čvorove stabala na toj slici algoritmi obilaze u sljedećem redosljedu:

- `Preorder( )`: 0,1,2,4,7,8,5,9,3,6
- `Inorder( )`: 1,0,7,4,8,2,9,5,6,3
- `Postorder( )`: 1,7,8,4,9,5,2,6,3,0



Slika 3.6: Obilazak stabla.

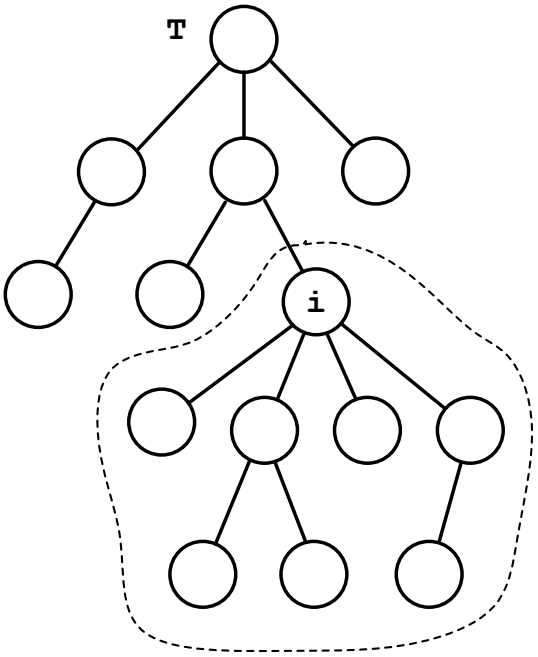
Za svaki od algoritama obilaska, redoslijed posjećivanja čvorova u konkretnom stablu jednoznačno je određen rekurzivnom definicijom tog algoritma. No taj redoslijed dodatno možemo pojasniti na način koji je ilustriran slikom 3.6. Dakle, dijagram stabla obrubimo crtkanom linijom koja kreće s lijeve strane korijena, prianja uz sve spojnice i čvorove te završava s desne strane korijena. Zamišljamo da algoritam "putuje" po crtkanoj liniji. Time se on svakom čvoru približava barem tri puta: najprije s lijeva, zatim (možda nekoliko puta) odozdo, te na kraju s desna. Tada vrijede sljedeća pravila.

- `Preorder( )` posjećuje čvor onda kad mu se približi s lijeva.
- `Inorder( )` posjećuje čvor onda kad mu se prvi put približi odozdo.
- `Postorder( )` posjećuje čvor onda kad mu se približi s desna.

Algoritmi obilaska mogu se lagano zapisati kao potprogrami. Najjednostavniji zapis dobiva se ako izravno slijedimo definiciju obilaska i služimo se rekurzijom. Pozivanjem isključivo funkcija iz apstraktnog tipa Tree možemo postići da naš potprogram bude neovisan o implementaciji apstraktnog tipa. Dakle možemo postići da isti potprogram radi u okviru raznih implementacija, pod uvjetom da su u svakoj od tih implementacija funkcije iz apstraktnog tipa Tree realizirane pod istim imenima i s istim argumentima.

U nastavku slijedi potprogram za `Preorder( )`. Slično bi izgledali i potprogrami za druga dva obilaska. Zbog konkretnosti, operacija posjećivanja čvora pretvorena je u ispis oznake čvora. No mogle bi se, naravno, obavljati i razne druge obrade. Način rada potprograma detaljnije je objašnjen slikom 3.7.

```
void Preorder (node i, Tree T) {
  /* obilazi se pod-stablo od T kojeg čini čvor i s potomcima */
  node c;
  printf ("...", TrLabel(i,T));
  c = TrFirstChild(i,T);
  while (c != LAMBDA) {
    Preorder (c,T);
    c = TrNextSibling(c,T);
  }
}
```



Slika 3.7: Objašnjenje implementacije `Preorder( )` - podstablo od `T` s korijenom `i`.

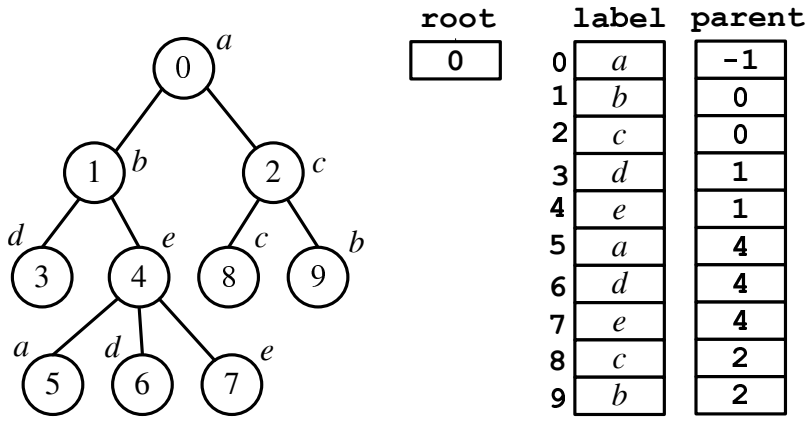
Kao što vidimo, naš potprogram `Preorder( )` prima dva argumenta: stablo `T` i čvor `i` iz tog stabla. Jednim rekurzivnim pozivom obilazi se podstablo u `T` koje se sastoji od čvora `i` i njegovih potomaka. `Preorder( )` najprije posjećuje čvor `i`, a zatim ulazi u petlju. Svaki korak petlje odgovara jednom djetetu od `i` i u njemu se rekurzivnim pozivom obilazi pod-podstablo koje se sastoji od tog djeteta i njemu pripadajućih potomaka. Prvo dijete od `i` pronalazi se pozivom funkcije `TrFirstChild( )` iz implementacije a.t.p. Tree. Svako iduće dijete (dakle idući brat prethodnog djeteta) dobiva se pozivom funkcije `TrNextSibling( )`. Obilazak cijelog stabla `T` postigao bi se pozivom `Preorder( )` gdje je uz samo stablo `T` na mjestu argumenta `i` zadan korijen od `T`.

3.1.3 Implementacija stabla na osnovi veze od čvora do roditelja

Prva implementacija uređenog stabla koju ćemo razmatrati zasniva se na tome da svakom čvoru eksplicitno zapišemo njegovog roditelja. Moguće su razne varijante, s obzirom na razne prikaze skupa čvorova i razne načine bilježenja veza među čvorovima. U nastavku biramo jednu od tih varijanti. Za nju su nam potrebne sljedeće definicije tipova u C-u:

```
#define MAXNODES ...  /* dovoljno velika konstanta */
#define LAMBDA -1

typedef int node;

typedef struct {
   node root;
   labeltype label[MAXNODES];
   node parent[MAXNODES];
} Tree;
```

S obzirom da za bilježenje veza namjeravamo koristiti kursore, imena čvorova moraju biti oblika 0, 1, 2, ... U skladu s time, tip `node` poistovjećen je sa cijelim brojevima, a kao ime nepostojećeg čvora koristi se `LAMBDA = -1`. Stablo se prikazuje dvama poljima `label[ ]` i `parent[ ]` čije duljine su određene unaprijed definiranom konstantom `MAXNODES`. Pritom i-te klijetke tih polja opisuju čvor i - u njima piše oznaka čvora odnosno kursor na roditelja. Prvo polje ima elemente tipa `labeltype`, dakle proizvoljnog tipa za oznake. Drugo polje sastoji se kursora pa su mu elementi cijeli brojevi. Postoji još i kursor `root` koji pokazuje gdje se u poljima nalazi korijen stabla. Tip `Tree` za sama stabla poistovjećuje se sa C-ovim `struct`-om koji objedinjuje sve opisane dijelove, dakle `label[ ]`, `parent[ ]` i `root`. Za ispravno funkcioniranje ovakve strukture neophodno je da konstanta `MAXNODES` bude veća ili jednaka stvarnom broju čvorova u stablu.



Slika 3.8: Implementacija stabla vezom od čvora do roditelja - struktura podataka.

Ideja implementacije ilustrirana je na konkretnom primjeru slikom 3.8. Stablo nacrtano na lijevoj strani slike prikazano je strukturom `T` s desne strane slike. Iz te strukture vidi se npr. da čvor 6 ima oznaku d te da mu je roditelj čvor 4, ili da čvor 4 s oznakom e ima kao roditelja čvor 1, ili da je korijen stabla čvor 0. Budući da je u našem slučaju `MAXNODES` jednak stvarnom broju čvorova, sve klijetke u poljima `T.label[ ]` i `T.parent[ ]` su zauzete. Kad bi `MAXNODES` bio veći, postojale bi slobodne klijetke koje bi po potrebi mogli označiti tako da im upišemo neku nemoguću vrijednost (npr. `T.parent[i]==i`).

Opisana struktura dobro podržava operacije `TrParent( )` i `TrLabel( )`. Ostale operacije zahtijevaju pretraživanje cijelog polja. Daljnja mana je da se ne pamti redoslijed braće - stablo je zapravo neuređeno. Ipak, možemo uvesti umjetno pravilo da su braća poredana po svojim imenima (indeksima). Uz to pravilo, funkcija `TrNextSibling( )` može se ovako zapisati:

```
node TrNextSibling (node i, Tree T) {
  node j, p;
  p = T.parent[i];
  for (j=i+1; j<MAXNODES; j++)
    /* tražimo klijetku nakon i-te u kojoj je upisan isti roditelj */
    if (T.parent[j] == p) return j;
  return LAMBDA; /* ne postoji idući brat */
}
```

Način rada ove funkcije prilično je razumljiv iz njezinog teksta. Dakle, da bi našla idućeg brata čvora `i`, funkcija gleda u polje `T.parent[ ]` na `i`-to mjesto te čita indeks `p` roditelja od `i`. Zatim dalje od istog mjesta sekvencijalno čita polje `T.parent[ ]` sve dok ne naiđe na prvu iduću pojavu iste vrijednosti `p`. Indeks pronađene klijetke određuje traženog brata, pa ga funkcija vraća kao svoju povratnu vrijednost. Ako u polju `T.parent[ ]` ne postoji iduća pojava od `p`, to znači da ne postoji idući brat od `i`, pa tada funkcija vraća ime nepostojećeg čvora `LAMBDA`.

Iz svega rečenog jasno je da opisana implementacija stabla na osnovi veze od čvora do roditelja ima nekih prednosti, ali i brojnih mana. Ustvari, implementacija je dobra samo pod sljedećim uvjetima.

- Nema mnogo ubacivanja ni izbacivanja čvorova.
- Nije potrebna uređenost stabla.
- Pretežno se koriste operacije `TrParent( )` i `TrLabel( )`.

3.1.4 Implementacija stabla na osnovi veze od čvora do djeteta i brata

Druga implementacija uređenog stabla koju ćemo razmatrati zasniva se na tome da svakom čvoru eksplicitno zapišemo njegovo prvo dijete te njegovog idućeg brata. Veza od čvora do djeteta odnosno brata može se realizirati pomoću pointera ili pomoću kursora. Odlučujemo se za varijantu s kursorima. Tada su nam potrebne sljedeće definicije u C-u.

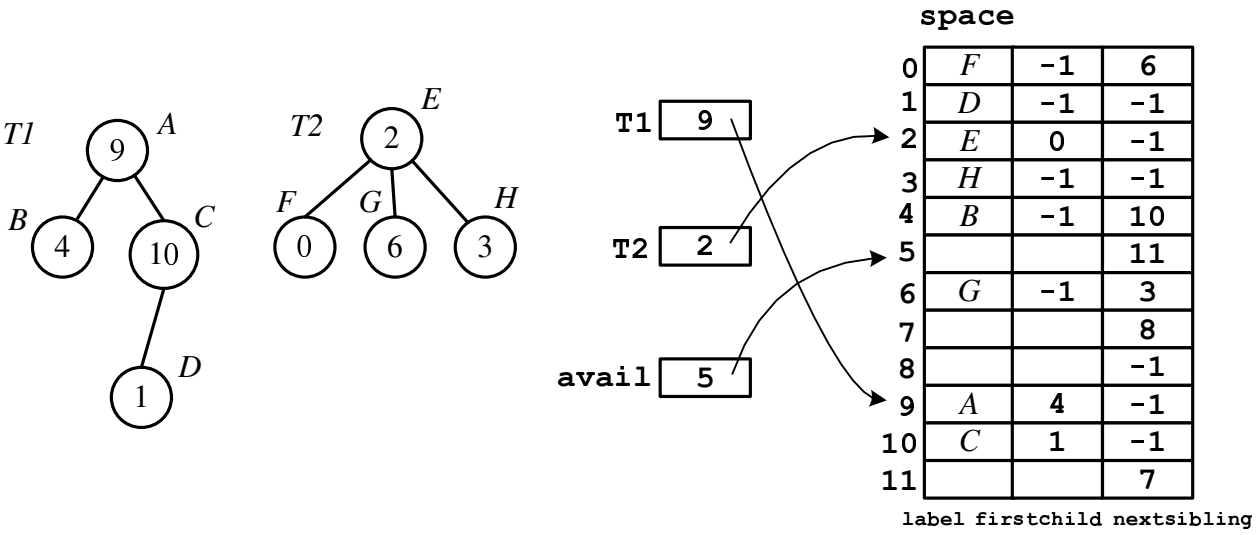
```
#define MAXNODES ...  /* dovoljno velika konstanta */
#define LAMBDA -1

typedef int node;
typedef int Tree;

typedef struct {
   labeltype label;
   node firstchild, nextsibling;
} nodestruct;

nodestruct space[MAXNODES];
int avail;
```

U gornjem programskom kodu najprije je definirano nekoliko konstanti i tipova. S obzirom da će se koristiti kursori, imena čvorova moraju biti oblika 0, 1, 2, ... U skladu s time, tip `node` poistovjećen je sa cijelim brojevima, a za ime nepostojećeg čvora izabrano je `LAMBDA = -1`. Budući da će se samo stablo interpretrirati kao kursor na vlastiti korijen, tip `Tree` je također poistovjećen sa cijelim brojevima. C-ov `struct` tipa `nodestruct` služi za prikaz jednog čvora i sastoji se od oznake tog čvora `label` i dva kursora `firstchild` odnosno `nextsibling` koji pokazuju na njegovo prvo dijete odnosno idućeg brata. Tip `labeltype` za oznake je proizvoljan. Konstanta `MAXNODES` je gornja ograda na ukupan broj čvorova u svim stablima s kojima radimo u isto vrijeme.

U gornjem programskom kodu također imamo dvije deklaracije varijabli. Globalno polje `space[ ]` služi kao "zaliha" klijetki od kojih će se graditi sva stabla. Pritom i-ta klijetka u tom polju opisuje čvor s imenom i. Razna stabla s kojima radimo trošit će klijetke iz istog (jedinstvenog) polja `space[ ]`. To znači da različiti čvorovi ne mogu imati ista imena čak ni onda kad se nalaze u različitim stablima. Klijetke iz istog stabla bit će međusobno povezane kursorima `firstchild` i `nextsibling`. Sve slobodne klijetke u `space[ ]` koje trenutno ne pripadaju ni jednom stablu povezat će se u vezanu listu, i to pomoću kursora smještenih npr. u (za ovu svrhu zloupotrebljenoj) komponenti `nextsibling`. Globalna varijabla `avail` služi kao kursor na početak te vezane liste.

Ideja implementacije ilustrirana je na konkretnom primjeru slikom 3.9. U strukturi na desnoj strani slike odjednom su prikazana dva stabla *T1* i *T2* čije dijagrame vidimo na lijevoj strani slike. Vidimo da je svaki čvor zauzeo odgovarajuću klijetku polja `space[ ]`. Stablo *T1* zauzelo je klijetke 1, 4, 9, 10, a stablo *T2* klijetke 0, 2, 3, 6. Svaka klijetka ima upisanu oznaku odgovarajućeg čvora, te ispravne kursore na prvo dijete odnosno idućeg brata tog čvora. Primjerice, iz klijetke 2 čitamo da čvor 2 ima oznaku E, da mu je prvo dijete 0 te da nema idućeg brata. Slično, klijetka 4 kaže da čvor 4 s oznakom B nema djece te da mu je idući brat 10.

Samo stablo *T1* odnosno *T2* sa slike 3.9 poistovjećuje se s varijablom `T1` odnosno `T2`, dakle s kursorom na korijen tog stabla. Zaista, u `T1` piše da je korijen prvog stabla u klijetki 9, dok u `T2` piše da je korijen drugog stabla u klijetki 2. Budući da dva stabla nisu do kraja popunila polje `space[ ]`, četiri slobodne klijetke 5, 7, 8, 11 povezane su kursorima `nextsibling` u vezanu listu. Globalni kursor `avail` bilježi da je početak te vezane liste u klijetki 5.

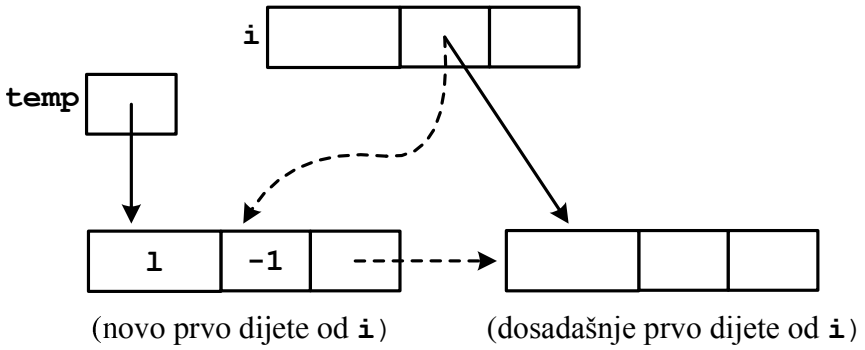


Slika 3.9: Implementacija stabla vezom čvor → dijete/brat - struktura podataka.

Opisana implementacija omogućuje da se sve operacije sa stablom osim `TrParent( )` i `TrDelete` izvršavaju u konstantnom vremenu. Kao primjer, pokazujemo programski kod za `TrInsertChild( )`. Način rada te funkcije dodatno je pojašnjen slikom 3.10. Pune strelice opet označavaju početno stanje, a crtkane strelice promjene.

```
node TrInsertChild (labeltype l, node i) {
  node temp;
  if (avail == LAMBDA)
    exit(401); /* nema slobodne memorije */
  else {
    temp = avail;
    avail = space[avail].nextsibling;
    space[temp].label = l;
    space[temp].firstchild = LAMBDA;
    space[temp].nextsibling = space[i].firstchild;
    space[i].firstchild = temp;
    return temp;
  }
}
```

Da bi čvoru `i` ubacila novo prvo dijete s oznakom `l`, funkcija `TrInsertChild( )` najprije iz vezane liste slobodnih izdvaja prvu po redu slobodnu klijetku i pamti njezin indeks kao `temp`. Zatim u izdvojenu klijetku upisuje oznaku `l` te je uključuje u stablo. Uključivanje se provodi tako da se u klijetku s indeksom `temp` upiše odgovarajući kursor na idućeg brata te da se u `i`-toj klijetki promijeni kursor na prvo dijete. Budući da tek ubačeni čvor još ne može imati svoje djece, njegov kursor na prvo dijete postavlja se na `LAMBDA`. Funkcija na kraju vraća ime novoubačenog čvora, dakle `temp`.



Slika 3.10: Implementacija stabla vezom čvor → dijete/brat - `TrInsertChild( )`.

Pri izvršavanju funkcije `TrInsertChild( )` može se desiti jedna izvanredna situacija, a to je prepunjenje polja `space[ ]`. Tada `TrInsertChild( )` ne može izvršiti svoj zadatak, pa prekida rad programa uz statusni kod 401. Prepunjenje polja prepoznajemo po tome što je vezana lista slobodnih klijetki prazna, tj. `avail == LAMBDA`.

Primijetimo da naša funkcija `TrInsertChild( )` sadrži jedan argument manje nego što je predviđeno u definiciji apstraktnog tipa Tree. To je zato što u ovoj implementaciji ime čvora `i` već jednoznačno određuje o kojem stablu je riječ. Ako bi željeli biti sasvim kompatibilni s definicijom apstraktnog tipa, mogli bi funkciji dodati argument `Tree *Tp`, no taj argument se ne bi koristio.

Temeljem svega rečenog, slijedi da je opisana implementacija stabla pogodna pod sljedećim okolnostima:

- Ima puno naknadnih ubacivanja čvorova.
- Radi se s više stabala koja se spajaju u veća.
- Intenzivno se koriste veze od roditelja prema djeci.

Za implementaciju stabla iz ovog pododjeljka mogli bismo reći da ima dijametralno suprotna svojstva od implementacije iz prethodnog pododjeljka. Naime, dok se u rješenju iz prethodnog odjeljka preferiralo kretanje kroz stablo "prema gore", dakle od djeteta prema roditelju, sad se preferira kretanje u suprotnom smjeru. U slučajevima kad nam je potrebno kretanje u oba smjera moguće je koristiti hibridnu (kombiniranu) implementaciju. Nju bi dobili tako da u klijetku za prikaz čvora, uz kursore na dijete i brata, također ugradimo i kursor na roditelja. Hibridna implementacija omogućila bi brzo manevriranje po stablu, no sadržavala bi veliku redundanciju u prikazu stabla.

3.2 Binarno stablo

U ovom odjeljku razmatramo *binarno* stablo. Riječ je o nešto pravilnije građenoj vrsti stabala, koja se često koristi u računarstvu i lakše se prikazuje u računalu. Binarno stablo opet je jedan apstraktni tip gdje su podaci uređeni u hijerarhiju. Pritom svakom nadređenom podatku pripadaju najviše dva podređena, neki od tih podređenih mogu

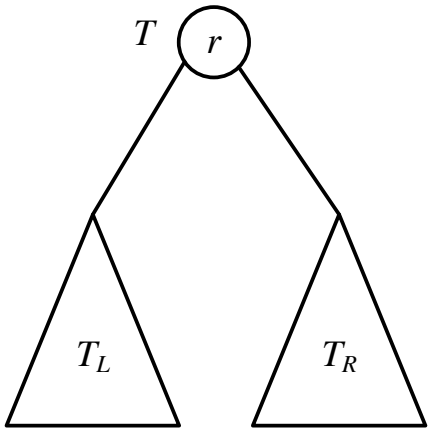
nedostajati, a ako nedostaju tada nije svejedno nedostaje li prvi ili drugi od njih. U odjeljku najprije navodimo svojstava i primjene binarnog stabla, zatim se bavimo njegovom uobičajenom implementacijom pomoću pointera, a na kraju uvodimo posebnu vrstu *potpunog* binarnog stabla te objašnjavamo kako se ono može prikazati poljem.

3.2.1 Svojstva i primjene binarnog stabla

Krećemo s matematičkom definicijom. **Binarno stablo** (*binary tree*) T je konačan skup podataka istog tipa koje zovemo **čvorovi**. Pri tome vrijedi:

- T je prazan skup (prazno binarno stablo), ili
- postoji istaknuti čvor r koji se zove **korijen** od T, a ostali čvorovi grade uređeni par (T_L, T_R) disjunktnih (manjih) binarnih stabala.

Ova definicija ilustrirana je slikom 3.11.



Slika 3.11: Binarno stablo i njegova podstabla.

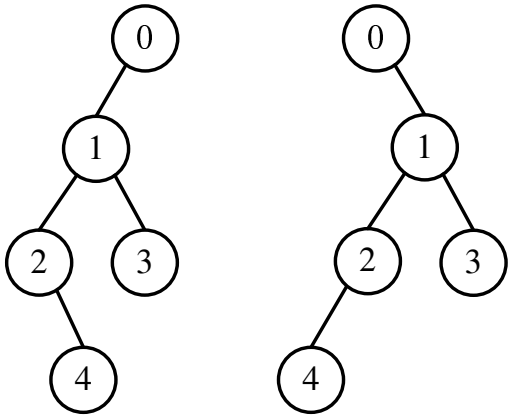
Opet je važno naglasiti da je definicija binarnog stabla rekurzivna. Dakle, ako T sadrži korijen r, tada se definicija primjenjuje i na manja binarna stabla T_L i T_R. Ta dva manja binarna stabla zovu se **lijevo** i **desno podstablo** od T. Korijen od T_L (ako postoji) je **lijevo dijete** od r, korijen od T_R (ako postoji) je **desno dijete** od r, a r je njihov **roditelj**. Svako od podstabala T_L i T_R može imati svoja još manja pod-podstabla, ona opet svoja pod-pod-podstabla, itd. Dosljednom primjenom rekurzije na kraju ćemo za svaki čvor u binarnom stablu ustanoviti čije je on dijete te čiji je roditelj. Očito, korijen je jedini čvor bez roditelja, a svaki od preostalih čvorova ima točno jednog roditelja. Broj djece nekog čvora može biti 0, 1 ili 2.

Za binarna stabla primjenjuje se ista terminologija kao kod uređenih stabala. Dakle, govorimo o putovima, duljini puta, precima, potomcima, braći, listovima, unutrašnjim čvorovima, razinama i visini binarnog stabla. Čvorovi opet mogu imati oznake. Binarno stablo crta se na isti način kao uređeno stablo. Također, primjenjuju se analogni algoritmi obilaska `Preorder( )`, `Inorder( )` i `Postorder( )`.

Primijetimo da binarno stablo nije specijalni slučaj uređenog stabla, kao što se obično misli. Naime:

- Binarno stablo može biti prazno.
- Ako čvor u binarnom stablu ima samo jedno dijete, tada nije svejedno da li je to lijevo ili desno dijete.

Primjerice, slika 3.12 prikazuje dva različita binarna stabla. Ako bi se ista slika odnosila na uređena stabla tada bi to bila dva prikaza jednog te istog uređenog stabla.



Slika 3.12: Dva različita binarna stabla.

U nastavku slijede primjeri korištenja binarnih stabala. Oni su u najvećoj mjeri vezani uz računarstvo.

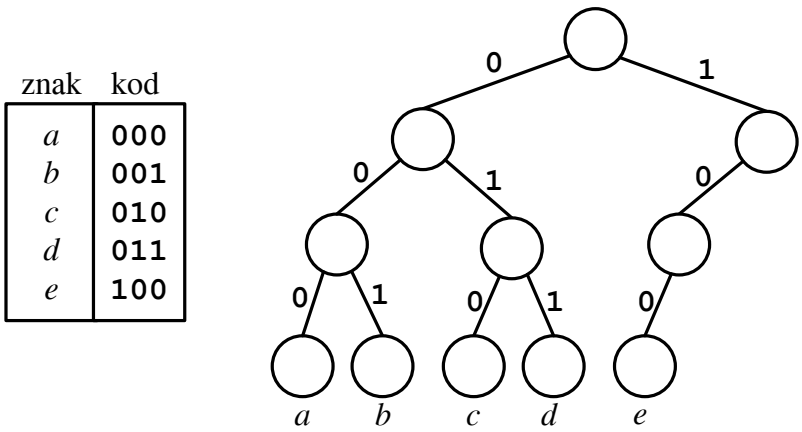
- U odjeljku 3.1 vidjeli smo da se građa aritmetičkog izraza može prikazati uređenim stablom. No ako se aritmetički izraz sastoji isključivo od binarnih operacija, tada se još bolji prikaz postiže binarnim stablom. Kao ilustracija može poslužiti otprilike ista slika kao slika 3.3 - ako čvor 6 izbacimo i zamijenimo ga čvorom 7, dobivamo binarno stablo koje prikazuje izraz $(a+b)*(a+c)$. Razlikovanje lijevog i desnog djeteta je važno ako su binarne operacije nekomutativne.
- Jedino što računalo zaista može pamtiti su bitovi 0 i 1. Svi ostali sadržaji moraju se kodirati nizovima bitova. Da bi pohranjene bitove pretvorili natrag u nama čitljiv oblik, potreban je postupak dekodiranja. Taj postupak zorno se može predočiti binarnim stablom. Primjerice, na slici 3.13 s lijeve strane vidimo tablicu tro-bitnih kodova za pet znakova, a s desne strane je odgovarajuće binarno stablo za dekodiranje. Da bi dekodirali zadani niz od tri bita, krećemo od korijena binarnog stabla, čitamo redom bit po bit iz koda te ovisno o tome što smo pročitali (`0` ili `1`) skrećemo lijevo ili desno. Nakon što smo obradili sva tri bita naći ćemo se u listu čija oznaka sadrži dekodirani znak. Npr. za kod `010` krećemo od korijena, skrećemo lijevo, pa desno, pa opet lijevo i dolazimo u list s oznakom c.
- Postupak donošenja odluke obično se može opisati takozvanim *stablom odlučivanja*. U tom stablu unutrašnji čvorovi sadrže pitanja, a listovi sadrže odluke. Da bi donijeli odluku, krećemo od korijena i manevriramo stablom sve dok ne dođemo do nekog lista. Manevriranje se provodi tako da odgovaramo na pitanja iz čvorova

kroz koje prolazimo te biramo smjer kretanja ovisno o odgovorima. Kad napokon stignemo u list, u njemu piše odluka koju trebamo donijeti. Ako su sva pitanja takva da se na njih odgovara s "da" ili "ne", tada stablo odlučivanja možemo smatrati binarnim. Primjer binarnog stabla odlučivanja vidi se na slici 3.14. Riječ je o donošenju odluke o nazivu za voće koje imamo na stolu.

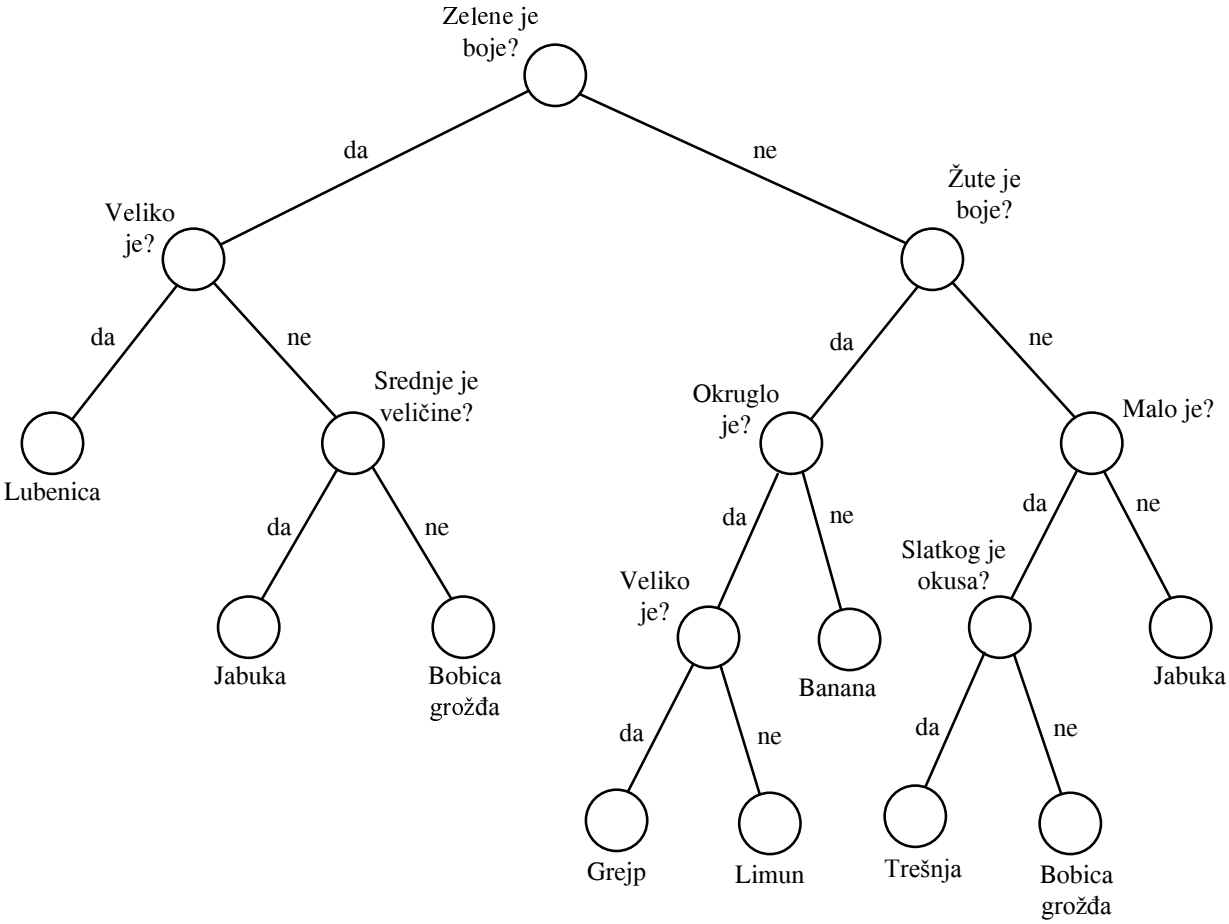
- Bilo koje uređeno stablo može se na osnovi veza od čvora do djeteta odnosno brata pretvoriti u binarno stablo. Zaista, pretvorba se obavlja na sljedeći način. Najprije sve čvorove iz uređenog stabla shvatimo kao čvorove binarnog stabla. Zatim veze od čvora do prvog djeteta iz uređenog stabla interpretiramo kao veze od čvora do lijevog djeteta u binarnom stablu. Slično, veze od čvora do idućeg brata iz uređenog stabla interpretiramo kao veze od čvora do desnog djeteta u binarnom stablu. Postupak pretvorbe ilustriran je na konkretnom primjeru slikom 3.15. Primijetimo da je postupak reverzibilan, tj. iz dobivenog binarnog stabla moguće je na jednoznačan način reproducirati polazno uređeno stablo. Ovu pretvorbu već smo nehotice koristili u implementaciji uređenog stabla iz pododjeljka 3.1.4. - zapravo se radilo o implementaciji ekvivalentnog binarnog stabla. Ovaj primjer ima i jednu dublju poruku. Ta poruka kaže da su uređena stabla suviše nepravilna da bi se mogla efikasno pohranjivati u računalu. Najbolji način njihova pohranjivanja jest da ih pretvorimo u ekvivalentna binarna stabla.

- Još neke važne primjene binarnog stabla bit će objašnjene u poglavlju 4. Naime, vidjet ćemo da se binarno stablo može upotrijebiti za prikaz skupova. Preciznije, radit će se o prikazu posebnih vrsta skupova koje se zovu rječnik odnosno prioritetni red.

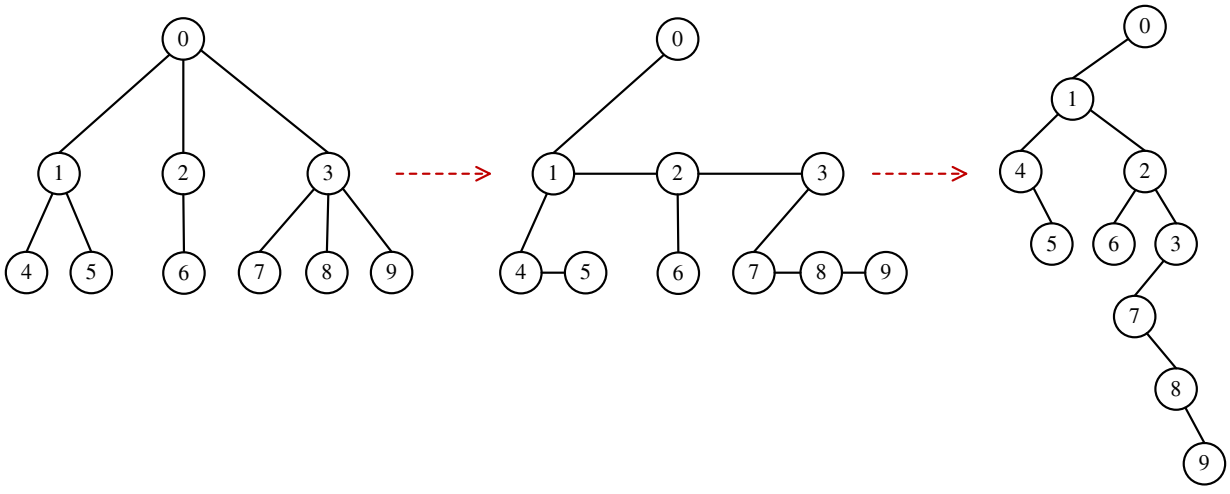
- Također, u poglavlju 5 objasnit ćemo da se binarna stabla pojavljuju kao sastavni dijelovi nekih algoritama za sortiranje. Točnije, riječ je o sortiranju obilaskom binarnog stabla traženja (*tree sort*) odnosno o sortiranju pomoću hrpe (*heap sort*). Oba spomenuta algoritma rade tako da od podataka koje treba sortirati najprije sagrade određenu vrstu binarnog stabla, a zatim iz tog binarnog stabla dobiju sortirani redoslijed.



Slika 3.13: Dekodiranje binarnog koda pomoću binarnog stabla.



Slika 3.14: Binarno stablo odlučivanja - izbor naziva voća.



Slika 3.15: Interpretacija uređenog stabla kao binarnog stabla.

Iz prethodnih primjera vidljivo je da je binarno stablo izuzetno važna tvorevina za računarstvo. To znači da ga trebamo pretvoriti u apstraktni tip podataka, a zatim ga što efikasnije implementirati. Binarno stablo može se na razne načine opisati kao apstraktni tip. Osim operacija koje rade na razini čvorova, lijepo se mogu definirati i operacije na razini podstabala. U nastavku slijedi prilično opširni popis operacija. Neke od njih se mogu izvesti pomoću drugih, no ipak ih navodimo zbog udobnijeg rada.

Apstraktni tip podataka BinaryTree

`node` ... bilo koji tip (čvorovi tj. njihova imena). U skupu `node` uočavamo jedan poseban element `LAMBDA` (koji služi kao ime nepostojećeg čvora).

`labeltype` ... bilo koji tip (oznake čvorova).

`BinaryTree` ... podatak tipa `BinaryTree` je binarno stablo čiji čvorovi su podaci tipa `node` (međusobno različiti i različiti od `LAMBDA`). Svakom čvoru je kao oznaka pridružen podatak tipa `labeltype`.

`BiMakeNull(&T)` ... funkcija pretvara binarno stablo `T` u prazno binarno stablo.

`BiEmpty(T)` ... funkcija vraća "istinu" ako i samo ako je `T` prazno binarno stablo.

`BiCreate(l,TL,TR,&T)` ... funkcija stvara novo binarno stablo `T`, kojem je lijevo podstablo `TL`, a desno pod-stablo `TR` (`TL` i `TR` moraju biti disjunktni). Korijen od `T` dobiva oznaku `l`.

`BiLeftSubtree(T,&TL), BiRightSubtree(T,&TR)` ... funkcija preko parametra `TL` odnosno `TR` vraća lijevo odnosno desno pod-stablo binarnog stabla `T`. Nije definirana ako je `T` prazno.

`BiInsertLeftChild(l,i,&T), BiInsertRightChild(l,i,&T)` ... funkcija u binarno stablo `T` ubacuje novi čvor s oznakom `l`, tako da on bude lijevo odnosno desno dijete čvora `i`. Funkcija vraća novi čvor (tj. njegovo ime). Nije definirana ako `i` ne pripada `T` ili ako `i` već ima dotično dijete.

`BiDelete(i,&T)` ... funkcija izbacuje list `i` iz binarnog stabla `T`. Nije definirana ako `i` ne pripada `T` ili ako `i` ima djece.

`BiRoot(T)` ... funkcija vraća korijen binarnog stabla `T`. Ako je `T` prazno, vraća `LAMBDA`.

`BiLeftChild(i,T), BiRightChild(i,T)` ... funkcija vraća lijevo odnosno desno dijete čvora `i` u binarnom stablu `T`. Ako `i` nema dotično dijete, vraća `LAMBDA`. Nije definirana ako `i` ne pripada `T`.

`BiParent(i,T)` ... funkcija vraća roditelja čvora `i` u binarnom stablu `T`. Ako je `i` korijen, vraća `LAMBDA`. Nije definirana ako `i` ne pripada `T`.

`BiLabel(i,T)` ... funkcija vraća oznaku čvora `i` u binarnom stablu `T`. Nije definirana ako `i` ne pripada `T`.

`BiChangeLabel(l,i,&T)` ... funkcija mijenja oznaku čvora `i` u stablu `T`, tako da ta oznaka postane `l`. Nije definirana ako `i` ne pripada `T`.

Slično kao lista, binarno stablo se može implementirati pomoću pointera ili pomoću polja. Kod prve vrste implementacije eksplicitno se zapisuju veze među čvorovima, dok su kod druge vrste te veze implicitno određene rasporedom podataka u memoriji računala. U preostalom dijelu ovog odjeljka proučit ćemo dvije implementacije, po jednu od svake vrste. Implementacija pomoću pointera je fleksibilnija i obično efikasnija, pa se češće koristi. Prikaz pomoću polja pogodan je samo za vrlo pravilna binarna stabla, primjerice za potpuna binarna stabla.

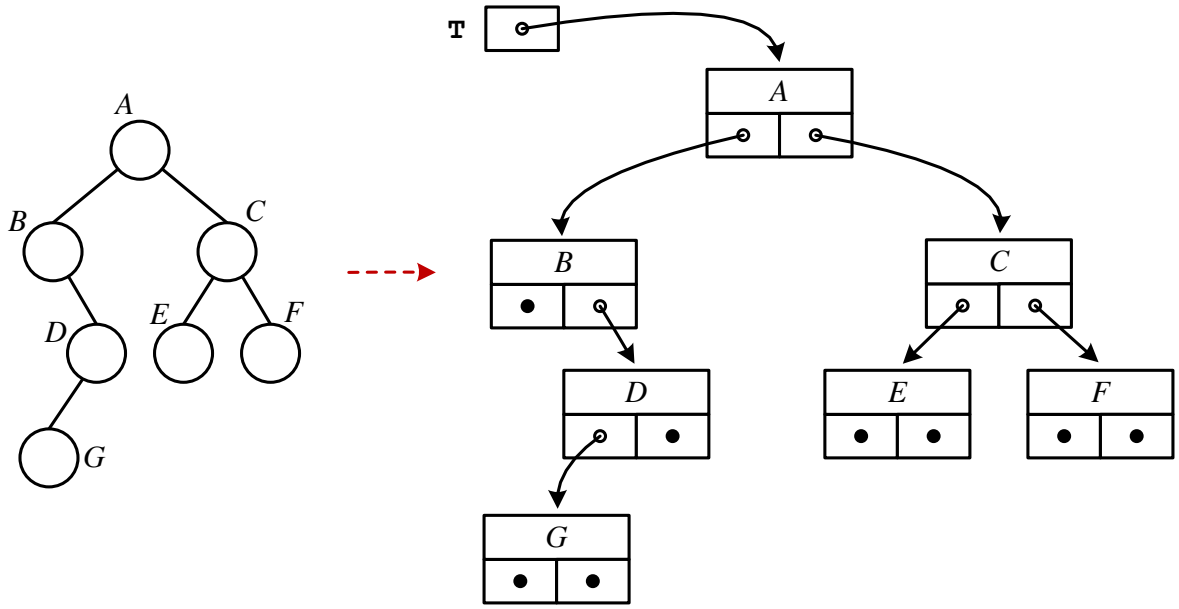
3.2.2 Implementacija binarnog stabla pomoću pointera

Ova implementacija zasniva se na tome da popišemo čvorove binarnog stabla te da svakom čvoru eksplicitno zapišemo njegovo lijevo i desno dijete. Veza od čvora do djeteta mogla bi se zabilježiti pomoću kursora ili pomoću pointera. Varijanta pomoću kursora bila bi gotovo identična implementaciji uređenog stabla opisanoj na kraju odjeljka 3.1. Zato sada, želeći da tekst bude raznolikiji, promatramo varijantu pomoću pointera. Za nju su potrebne sljedeće definicije tipova u C-u:

```
typedef struct celltag {
   labeltype label;
   struct celltag *leftchild;
   struct celltag *rightchild;
} celltype;

typedef celltype *node;
typedef celltype *BinaryTree;
```

Gore definirani C-ov `struct celltype` služi nam za opis jednog čvora binarnog stabla. U klijetki se pojavljuje oznaka čvora, pointer na istu takvu klijetku koja sadrži lijevo dijete tog čvora te pointer na istu takvu klijetku koja odgovara desnom djetetu. Tip `labeltype` za oznake je proizvoljan. Budući da će se klijetke tipa `celltype` stvarati dinamičkom alokacijom memorije, njih možemo identificirati jedino preko njihovih adresa. U skladu s time, tip `node` za imena čvorova poistovjećen je s pointerom na `celltype`. Binarno stablo gradit će se kao vezana struktura klijetki tipa `celltype`. S obzirom da ćemo cijelu strukturu identificirati preko adrese klijetke koja predstavlja korijen binarnog stabla, tip `BinaryTree` također je poistovjećen s pointerom na `celltype`. U slučaju praznog binarnog stabla taj pointer je `NULL`.

Ideja implementacije ilustrirana je na konkretnom primjeru slikom 3.16. Stablo s lijeve strane slike prikazano je strukturom s desne strane. Vidimo da je svaki čvor pretvoren u jednu klijetku koja sadrži odgovarajuću oznaku. U skladu s vezama od roditelja do djece, klijetke su međusobno povezane pointerima. Kad neki čvor nema lijevo ili desno dijete, odgovarajući pointer je `NULL`. Varijabla `T` koja predstavlja cijelo binarno stablo zapravo je pointer na klijetku koja odgovara korijenu. Smjer u kojem su postavljeni pointeri nalaže nam čitanje strukture "odozgo prema dolje". Dakle, da bi pročitali cijelo binarno stablo sa slike 3.16, najprije iz varijable `T` saznajemo adresu klijetke s oznakom A, zatim pristupamo toj klijetki pa iz nje saznajemo adrese klijetki s oznakama B i C, itd. Čitanje u smjeru "odozdo prema gore" nije izravno podržano.



Slika 3.16: Implementacija binarnog stabla pomoću pointera - struktura podataka.

Lako se vidi da se sve operacije iz apstraktnog tipa BinaryTree osim `BiDelete( )` i `BiParent( )` mogu efikasno implementirati tako da im vrijeme izvršavanja bude $\mathcal{O}(1)$. Primjerice, funkcija `BiCreate()` može se realizirati ovako:

```
void BiCreate
(labeltype l, BinaryTree TL, BinaryTree TR, BinaryTree *Tp) {
  *Tp = (celltype*)malloc(sizeof(celltype));
  (*Tp)->label = l;
  (*Tp)->leftchild = TL;
  (*Tp)->rightchild = TR;
}
```

Dakle, da bi dva postojeća binarna stabla `TL` i `TR` spojila u veće binarno stablo `T`, funkcija `BiCreate( )` najprije fizički stvara novu klijetku koja će služiti kao korijen od `T`. U tu klijetku upisuje se zadana oznaka `l` te pointeri na lijevo i desno dijete koji zapravo pokazuju na korijene od `TL` i `TR`. Sama varijabla `T` predstavljena je svojom adresom `Tp` i postavlja se tako da pokazuje na novu klijetku.

Ako nam je u radu s binarnim stablima važna funkcija `BiDelete( )` ili `BiParent( )`, tada klijetku za prikaz čvora (dakle strukturu `celltype`) možemo proširiti tako da ona također sadrži i pointer prema roditelju. Takvo proširenje omogućilo bi lagano "manevriranje" po binarnom stablu u svim smjerovima, pa time i izvršavanje svih funkcija iz apstraktnog tipa BinaryTree u konstantnom vremenu. No u prikaz binarnog stabla unijela bi se redundancija jer bi se veza između roditelja i djeteta prikazivala dvaput.

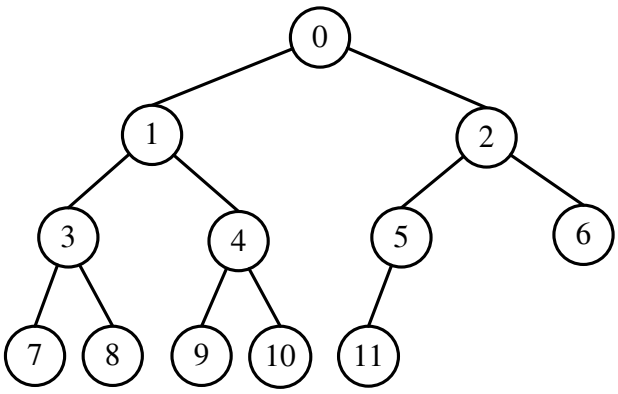
3.2.3 Implementacija potpunog binarnog stabla pomoću polja

U nekim važnim primjenama pojavljuje se posebna vrsta binarnih stabala vrlo pravilnog oblika koja zovemo *potpuna* binarna stabla. Zbog svog pravilnog oblika, ta binarna stabla mogu se jednostavnije i efikasnije prikazati u memoriji računala. Slijedi definicija.

Potpuno binarno stablo građeno je od n čvorova, s imenima $0, 1, 2, \ldots, n-1$. Pritom vrijedi:

- Lijevo dijete čvora i je čvor $2i + 1$ (ako je $2i + 1 > n - 1$ tada čvor i nema lijevo dijete);
- Desno dijete čvora i je i čvor $2i + 2$ (ako je $2i + 2 > n - 1$ tada čvor i nema desno dijete).

Primjerice, potpuno binarno stablo s $n = 12$ čvorova izgleda kao na slici 3.17.



Slika 3.17: Potpuno binarno stablo s 12 čvorova.

Potpuno binarno stablo možemo i ovako neformalno opisati.

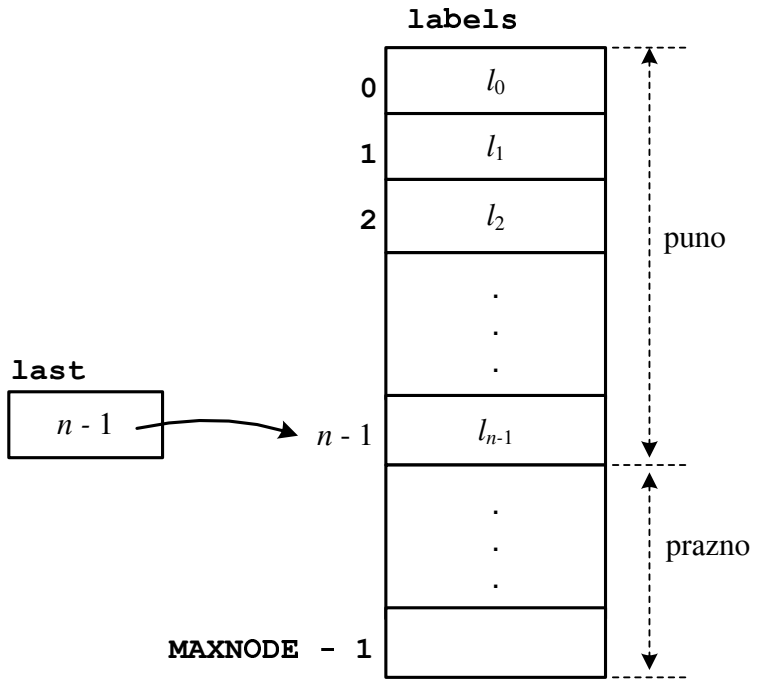
- Na svim razinama osim zadnje postoje svi mogući čvorovi.
- Čvorovi na zadnjoj razini "gurnuti" su na lijevu stranu.
- Imenovanje (numeriranje) čvorova ide redom s razine 0 na razinu 1, razinu 2, ..., itd. s lijeva na desno.

Potpuno binarno stablo treba zamišljati kao objekt sa statičkom građom. Na njega se ne namjeravaju primjenjivati operacije poput `BiCreate()`, `BiLeftSubtree()` ili `BiRightSubtree()` jer rezultat više ne bi bio potpuno binarno stablo. Umjesto toga potrebno je "manevrirati" po već zadanom binarnom stablu, te čitati ili mijenjati oznake čvorova. Eventualno se dopušta ubacivanje ili izbacivanje čvorova na desnom kraju zadnje razine.

Potpuno binarno stablo može se elegantno prikazati u računalu pomoću polja - vidi sliku 3.18. U skladu s tim prikazom, i-ta klijetka polja sadrži oznaku čvora i. Također postoji kursor koji pokazuje zadnji čvor $n - 1$. Dakle, imamo sljedeće definicije u C-u:

```
#define MAXNODE .../* dovoljno velika konstanta */
typedef int node;

typedef struct {
   int last;
   labeltype labels[MAXNODE];
} BinaryTree;
```

U gornjim definicijama `MAXNODE` je konstanta koja postavlja gornju ogradu na broj čvorova n u binarnom stablu. Budući da su imena čvorova oblika 0, 1, 2, ..., $n-1$, tip `node` poistovjećuje se sa cijelim brojevima. U skladu s prethodnim objašnjenjima, tip `BinaryTree` poistovjećuje se sa `struct`-om koji se sastoji od jednog kursora i jednog polja duljine `MAXNODE`. Elementi polja moraju biti onog tipa tipa kojeg su oznake, dakle `labeltype`.

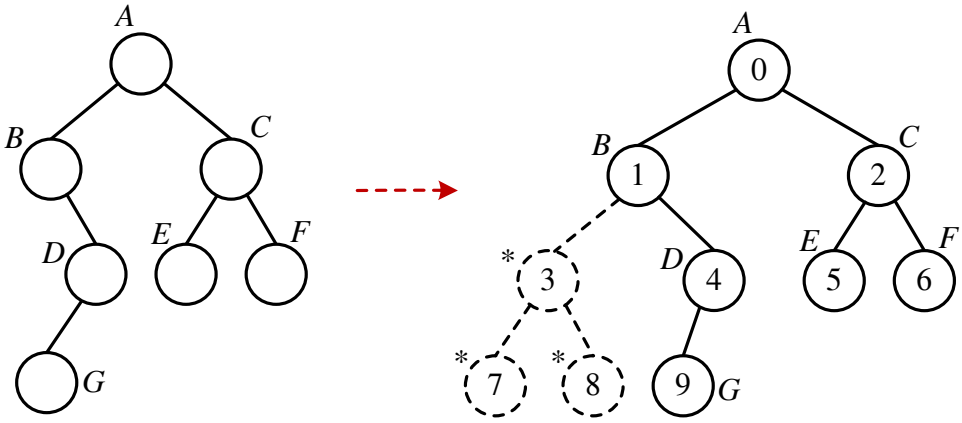


Slika 3.18: Prikaz potpunog binarnog stabla pomoću polja.

Manevriranje po ovako prikazanom potpunom binarnom stablu obavlja se na očigledan način. Korijen je predstavljen 0-tom klijetkom. Lijevo i desno dijete čvora iz i-te klijetke nalaze se u $(2i+1)$-oj i $(2i+2)$-oj klijetki (ako postoje). Roditelj čvora iz i-te klijetke je u $\lfloor (i-1)/2 \rfloor$-toj klijetki.

Primjene potpunog binarnog stabla bit će opisane u idućim poglavljima. U odjeljku 4.3. vidjet ćemo da se takvo binarno stablo zajedno sa svojim prikazom pomoću polja može vrlo uspješno upotrijebiti za implementaciju prioritetnog reda. Ista konstrukcija dalje se u odjeljku 5.4 koristi kao temelj algoritma za sortiranje pomoću hrpe (*heap sort*).

Na kraju napomenimo da se prikaz binarnog stabla pomoću polja donekle može proširiti i na binarna stabla koja nisu potpuna. U tu svrhu binarno stablo najprije nadopunimo do potpunog uvođenjem "lažnih" čvorova, pa ga zatim prikazujemo na način kako smo opisali. Ideja je ilustrirana slikom 3.19: nepotpuno binarno stablo s lijeve strane slike pretvara se u potpuno s desne strane. Lažni čvorovi iscrtani su isprekidanim linijama.



Slika 3.19: Upotpunjavanje binarnog stabla koje nije potpuno.

Naravno, da bi ovakav trik bio upotrebljiv, u prikazu moramo nekako razlikovati stvarne čvorove od lažnih. Razlikovanje se postiže tako da se lažnim čvorovima pridruži oznaka za koju smo sigurni da se nikad neće pojaviti kod stvarnih čvorova. Primjerice, polazno binarno stablo sa slike 3.19 označeno je slovima A, B, C, ... To znači da lažnim čvorovima možemo kao oznaku dati neki znak koji nije slovo, npr. $*$. Kad manevriranjem po prikazanom stablu naiđemo na čvor s oznakom $*$, tada znamo da taj čvor zapravo ne postoji.

Ako se polazno binarno stablo jako razlikuje od potpunog, tada se opisana metoda upotpunjavanja ne isplati jer broj lažnih čvorova može znatno premašiti broj stvarnih čvorova. Primjerice, ako imamo binarno stablo od n čvorova koje je maksimalno izduženo u desnu stranu (kao na slici 4.9 desno), tada ukupan broj čvorova u upotpunjenom binarnom stablu iznosi $2^n - 1$. To znači da bi za $n = 10$ morali koristiti polje od 1023 klijetke, a od njih bi samo 10, dakle manje od 1%, sadržavalo stvarne podatke.

Poglavlje 4

SKUPOVI

Ovo poglavlje bavi se apstraktnim tipovima podataka koji otprilike odgovaraju matematičkom pojmu skupa. Za razliku od lista gdje su podaci bili uređeni u niz, odnosno stabala gdje su bili organizirani u hijerarhiju, kod skupova ne postoji nikakav eksplicitni odnos, već su svi podaci ravnopravni. Poglavlje se sastoji od četiri odjeljka od kojih svaki obrađuje jednu vrstu skupova ili sličnih tvorevina. Najprije govorimo o općenitom skupu, zatim o posebnim vrstama kao što su rječnik i prioritetni red, a na kraju o preslikavanju i binarnoj relaciji.

4.1 Općeniti skup

U ovom odjeljku proučavamo prilično općenitu varijantu skupa gdje se osim operacija s pojedinim elementima primenjuju i glomaznije operacije poput unije, presjeka i razlike. U odjeljku najprije navodimo svojstva i primjene općenitog skupa, a zatim raspravljamo o njegovim implementacijama pomoću bit-vektora odnosno pomoću vezane liste.

4.1.1 Svojstva i primjene skupa

Skup (*set*) je pohranjena kolekcija podataka istog tipa koje zovemo **elementi**. U jednoj kolekciji ne mogu postojati dva podatka s istom vrijednošću. Unutar kolekcije se ne zadaje nikakav eksplicitni linearni ili hijerarhijski uređaj među podacima, niti bilo kakav drugi oblik veza među podacima. Ipak, uzimamo da za same vrijednosti elemenata postoji neka implicitna (prirodna) relacija totalnog uređaja. Dakle, za bilo koja dva elementa možemo govoriti o tome koji od njih je veći, a koji manji.

Slijede primjeri korištenja skupova. Naglasak je na primjenama gdje su potrebne skupovne operacije unija, presjek i razlika.

- Baza podataka sadrži informacije o stanovnicima nekog grada. Pretraživanjem baze dobivamo skupove imena osoba:

  $A = \{$ osobe s barem 10 godina staža $\}$,
  $B = \{$ osobe s visokom stručnom spremom $\}$.

  Tada osobe s barem deset godina staža ili s visokom stručnom spremom računamo kao $A \cup B$. Slično, osobe s barem 10 godina staža i visokom stručnom spremom

dobivamo kao $A \cap B$. Nadalje, $B \setminus A$ čine osobe s visokom stručnom spremom koje još nemaju 10 godina staža.

- Student Pero Marković nekoliko je godina studirao matematiku, no sad bi se htio prebaciti na studij računarstva. Dva studija srećom imaju dosta zajedničkih predmeta. Definirajmo sljedeće skupove:

  $A = \{$ predmeti sa studija matematike koje je Pero položio $\}$,
  $B = \{$ predmeti sa studija računarstvo $\}$.

  Da bi ga upisala na studij računarstva, službenica u studentskoj referadi najprije izdaje Peri potvrdu da mu se predmeti iz presjeka $A \cap B$ priznaju kao položeni. Zatim u novi Perin indeks upisuje samo predmete iz razlike $B \setminus A$.

- Udruga za zaštitu životinja pokrenula je inicijativu da se u statut grada Zagreba upiše odredba da je pas čovjekov najbolji prijatelj. Kao podršku svojoj inicijativi, udruga je skupljala potpise građana na dvadeset mjesta u gradu. Potpisi se nalaze u skupovima $A_1, A_2, \ldots, A_{20}$. Gradsko poglavarstvo sumnja da su se isti građani potpisivali više puta na raznim mjestima. Da bi se utvrdio stvarni broj potpisa, potrebno je izračunati uniju $A_1 \cup A_2 \cup \ldots \cup A_{20}$ pa onda u njoj prebrojiti elemente.

U nastavku slijedi definicija skupa kao apstraktnog tipa podataka. Definicija uključuje operacije koje su uobičajene u elementarnoj matematici.

Apstraktni tip podataka Set

`elementtype` ... bilo koji tip s totalnim uređajem `<=`.

`Set` ... podatak tipa `Set` je konačni skup čiji elementi su (međusobno različiti) podaci tipa `elementtype`.

`SeMakeNull(&A)` ... funkcija pretvara skup `A` u prazan skup.

`SeInsert(x,&A)` ... funkcija ubacuje element `x` u skup `A`, tj. mijenja `A` u $\texttt{A} \cup \{\texttt{x}\}$. Ako `x` već jeste element od `A`, tada `SeInsert( )` ne mijenja `A`.

`SeDelete(x,&A)` ... funkcija izbacuje element `x` iz skupa `A`, tj. mijenja `A` u $\texttt{A} \setminus \{\texttt{x}\}$. Ako `x` nije element od `A`, tada `SeDelete( )` ne mijenja `A`.

`SeMember(x,A)` ... funkcija vraća "istinu" ako je $\texttt{x} \in \texttt{A}$, odnosno "laž" ako $\texttt{x} \notin \texttt{A}$.

`SeMin(A), SeMax(A)` ... funkcija vraća najmanji odnosno najveći element skupa `A`, u smislu uređaja `<=`. Nije definirana ako je `A` prazan skup.

`SeSubset(A,B)` ... funkcija vraća "istinu" ako je $\texttt{A} \subseteq \texttt{B}$, inače vraća "laž".

`SeUnion(A,B,&C)` ... funkcija pretvara skup `C` u uniju skupova `A` i `B`. Dakle, `C` mijenja u $\texttt{A} \cup \texttt{B}$.

`SeIntersection(A,B,&C)` ... funkcija pretvara skup `C` u presjek skupova `A` i `B`. Dakle, `C` mijenja u $\texttt{A} \cap \texttt{B}$.

`SeDifference(A,B,&C)` ... funkcija pretvara skup `C` u razliku skupova `A` i `B`. Dakle, `C` mijenja u $\texttt{A} \setminus \texttt{B}$.

Ovako zadani apstraktni tip podataka Set dosta je teško implementirati. Ako postignemo efikasno obavljanje jednih operacija, sporo će se obavljati neke druge. U ovom odjeljku gledamo implementacije napravljene sa ciljem da se spretno obavljaju operacije s više skupova poput `SeUnion( )`, `SeIntersection( )`, `SeDifference( )` ili `SeSubset( )`. Implementacije koje pogoduju ostalim operacijama obrađuju se u odjeljcima 4.2. i 4.3.

4.1.2 Implementacija skupa pomoću bit-vektora

Ideja implementacije ilustrirana je slikom 4.1. Uzimamo bez velikog gubitka općenitosti da je `elementtype` $= \{0, 1, 2, \ldots, \texttt{N} - 1\}$, gdje je `N` dovoljno velika `int` konstanta. Skup prikazujemo poljem bitova (ili byte-ova tj. `char`-ova). Bit s indeksom i je 1 (odnosno 0) ako i samo ako i-ti element pripada (odnosno ne pripada) skupu. Dakle imamo sljedeće definicije tipova:

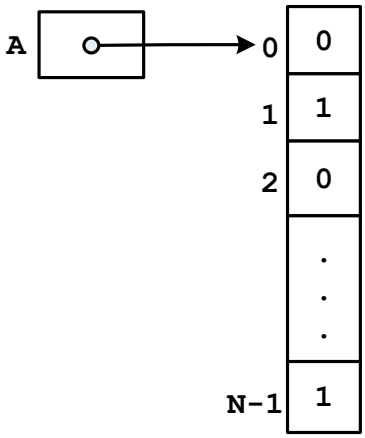
```
#define N ...  /* dovoljno velika konstanta */

typedef char *Set; /* početna adresa char polja duljine N */
```

Pojedinu varijablu tipa `Set` inicijaliziramo dinamičkim alociranjem memorije za polje, ili tako da je poistovijetimo s početnom adresom već deklariranog polja. Dakle:

```
Set A;
A = (char*) malloc(N);
```

ili:

```
...
char bv[N];
Set A = bv;
```



Slika 4.1: Implementacija skupa pomoću bit-vektora - korištena struktura podataka.

Operacije iz apstraktnog tipa podataka Set mogu se isprogramirati na očigledan način. `SeInsert( )`, `SeDelete( )` i `SeMember( )` zahtijevaju konstantno vrijeme, budući da se svode na direktan pristup i-tom bitu. S druge strane, `SeUnion( )`, `SeIntersection( )`, `SeDifference( )` i `SeSubset( )` zahtijevaju vrijeme proporcionalno `N`. Ispisujemo `SeUnion( )`, a ostale funkcije su slične:

```
void SeUnion (Set A, Set B, Set *Cp) {
  int i;
  for (i=0; i<N; i++)
    (*Cp)[i] = (A[i]==1) || (B[i]==1);
}
```

Velika mana ove implementacije jest da prostor potreban za pohranjivanje skupa ne ovisi o veličini samog skupa nego o veličini tipa `elementype` dakle, o konstanti `N`. Primjerice, ako `elementype` čine 32-bitni cijeli brojevi, tada je `N` otprilike 4 milijarde, pa bi čak i za pohranjivanje malih skupova trošili po 4 milijarde bitova. Pored toga, vrijeme izvršavanja za većinu operacija bilo bi nedopustivo dugo. Implementacija očito postaje neupotrebljiva ako je `N` iole velik.

4.1.3 Implementacija skupa pomoću sortirane vezane liste

Za razliku od implementacije pomoću bit-vektora, koja je zapravo trošila memoriju i za elemente koji ne postoje u skupu, ova implementacija nastoji to izbjeći te eksplicitno zapisuje samo elemente koji postoje u skupu. Osnovna ideja jest da se skup prikaže kao lista njegovih elemenata te da se ta lista dalje implementira na jedan on načina opisanih u odjeljku 2.1. Od dviju razmatranih implementacija liste, bolja je ona pomoću pointera. Naime, veličina skupova dobivenih operacijama $\cup, \cap, \setminus$ može jako varirati. Da bi se operacije brže obavljale, dobro je da lista bude sortirana u skladu s uređajem `<=`.

Pretpostavimo da je tip `celltype` definiran kao u odjeljku 2.1. Tada tip `Set` možemo definirati na sljedeći način:

```
typedef celltype *Set;
```

U nastavku ispisujemo programski kod za operaciju `SeIntersection( )`. Operacije `SeUnion( )`, `SeDifference( )` i `SeSubset( )` realiziraju se analogno.

```
void SeIntersection (Set ah, Set bh, Set *chp) {
  /* Računa presjek skupova A i B, prikazanih sortiranim vezanim
  listama čija zaglavlja pokazuju pointeri ah i bh.  Rezultat je
  skup C prikazan sortiranom vezanom listom čiji pointer na
  zaglavlje pokazuje chp */

  celltype *ac, *bc, *cc;
  /* tekuće klijetke u listi A i B, zadnja klijetka u listi C */

  *chp = (celltype*)malloc(sizeof(celltype));
  /* stvori zaglavlje liste C */
```

```
  ac = ah->next;
  bc = bh->next;
  cc = *chp;

  while ((ac!=NULL) && (bc!=NULL)) {
    /* usporedi tekuće elemente liste A i B */
    if ( (ac->element) == (bc->element) ) {
      /* dodaj element u presjek C */
      cc->next = (celltype*)malloc(sizeof(celltype));
      cc = cc->next;
      cc->element = ac->element;
      ac = ac->next;
      bc = bc->next;
    }
    else if ((ac->element)<(bc->element))
      /* elementi su različiti, onaj iz A je manji */
      ac = ac->next;
    else
      /* elementi su različiti, onaj iz B je manji */
      bc = bc->next;
  }
  cc->next = NULL;
}
```

Vidimo da naša funkcija `SeIntersection( )` pretpostavlja da su skupovi A i B prikazani sortiranim vezanim listama. Također, ona gradi novu takvu listu u kojoj će se nalaziti elementi iz $C = A \cap B$. U `while` petlji funkcija simultano prolazi listama za A i B. U svakom trenutku tog prolaska određeni su "tekući" elementi iz A odnosno B. Funkcija uspoređuje dva tekuća elementa: ako su oni jednaki, tada se njihova vrijednost očito mora naći u presjeku $A \cap B$, pa se ona prepisuje u treću listu, a obje kazaljke tekućih elemenata pomiču se za jedno mjesto dalje. No ako tekući elementi nisu jednaki, tada manji od njih nema šanse da se nađe u presjeku (jer su u drugoj listi preostali elementi još veći od tekućeg), pa se kazaljka tog manjeg elementa pomiče za jedno mjesto dalje i nema prepisivanja. Petlja završava onda kad se iscrpi jedna od ulaznih listi.

Očito, korektnost naše funkcije počiva na činjenici da su liste za A i B sortirane. Kad liste ne bi bile sortirane, tada ne bi mogli tako jednostavno izračunati $A \cap B$ jer bi svaki element iz A morali usporediti sa svakim elementom iz B.

Primijetimo da je vrijeme izvršavanja ovako implementirane `SeIntersection( )` proporcionalno duljini jedne od listi, dakle ograđeno veličinama zadanih skupova. Slično bi vrijedilo i za ostale operacije.

4.2 Rječnik

U ovom odjeljku bavimo se posebnom vrstom skupa koja se naziva rječnik (*dictionary*). Kod njega nisu predviđene složene operacije poput unije, presjeka, skupovne razlike, inkluzije. Umjesto toga, obavljaju se samo povremena ubacivanja i izbacivanja eleme-

nata te provjere je li neki element u skupu ili nije. Rječnik je važan jer se pojavljuje u brojnim primjenama, a zbog jednostavnog načina uporabe može se efikasnije implementirati nego općeniti skup. Nakon navođenja svojstava i primjena, odjeljak se uglavnom bavi implementacijama rječnika. Dvije najvažnije implementacije zasnovane su na *hash* tablici odnosno na binarnom stablu traženja.

4.2.1 Svojstva i primjene rječnika

U skladu s prije rečenim, rječnik najlakše možemo opisati kao restrikciju skupa gdje su dozvoljene samo operacije stvaranja praznog skupa, ubacivanja elementa u skup, izbacivanja elementa iz skupa te provjere da li element pripada skupu. Osim toga, rječnik možemo smatrati i posebnim apstraktnim tipom u skladu sa sljedećom definicijom.

Apstraktni tip podataka Dictionary

`elementtype` ... bilo koji tip s totalnim uređajem `<=`.

`Dictionary` ... podatak tipa `Dictionary` je konačni skup koji zovemo rječnik i čiji elementi su (međusobno različiti) podaci tipa `elementtype`.

`DiMakeNull(&A)` ... funkcija pretvara rječnik `A` u prazan skup.

`DiInsert(x,&A)` ... funkcija ubacuje element `x` u rječnik `A`, tj. mijenja `A` u $A \cup \{x\}$. Znači, ako `x` već jeste u `A`, tada `DiInsert( )` ne mijenja `A`.

`DiDelete(x,&A)` ... funkcija izbacuje element `x` iz rječnika `A`, tj. mijenja `A` u $A \setminus \{x\}$. Znači, ako `x` nije u `A`, tada `DiDelete( )` ne mijenja `A`.

`DiMember(x,A)` ... funkcija vraća "istinu" ako $x \in A$, odnosno "laž" ako $x \notin A$.

U nastavku slijede primjeri korištenja rječnika. Neki od njih su iz običnog života, a neki su vezani uz računarstvo.

- Pravopis je popis ispravno napisanih riječi nekog jezika. Da bismo ustanovili je li neka riječ ispravno zapisana, gledamo nalazi li se ona na tom popisu. Jezik se vremenom mijenja, pa u skladu s time u pravopis povremeno ubacujemo nove izraze ili izbacujemo zastarjele. Vidimo da je pravopis skup nad kojim se obavljaju operacije traženja elementa te ubacivanja ili izbacivanja elemenata. Ili drukčije rečeno, radi se o rječniku.

- Neki računalni programi za obradu teksta (editori, tekst procesori) opremljeni su tzv. *spelling checker*-om, dakle računalnim ekvivalentom pravopisa. Takav alat obično daje podršku za razne jezike, npr. engleski ili hrvatski. Dok korisnik upisuje tekst, *spelling checker* uspoređuje riječi iz teksta sa svojim popisom ispravnih riječi te dojavljuje pogreške. Korisnik u skladu sa svojim potrebama i specifičnom terminologijom koju koristi obično može mijenjati popis ispravnih riječi.

- Višekorisničko računalo omogućuje rad samo registriranim korisnicima. Računalo prepoznaje korisnike na osnovi njihovih imena (*login name*). Kod prijavljivanja, korisnik upisuje ime, a računalo provjerava postoji li to ime na odgovarajućem popisu. Ovlaštena osoba (tzv. sistem-administrator) dodaje na popis imena novih

korisnika ili briše imena odsutnih. Dakle vidimo da se nad skupom korisnika obavljaju rječničke operacije: traženje, ubacivanje te izbacivanje elemenata.

Budući da je rječnik jedan vrsta skupa, on se može implementirati na iste načine kao i općeniti skup. No postoje i specifične implementacije koje omogućuju efikasnije obavljanje rječničkih operacija.

4.2.2 Implementacija rječnika pomoću bit-vektora

Radi se o sasvim istoj implementaciji koju smo u odjeljku 4.1 već opisali za općeniti skup. Dakle rječnik se prikazuje poljem bitova gdje svaki bit određuje da li pojedina vrijednost tipa `elementtype` postoji u rječniku ili ne. Prikaz pomoću bit-vektora svakako je dobar onda kad se tip `elementtype` sastoji od svega nekoliko mogućih vrijednosti. Operacije `DiInsert( )`, `DiDelete( )` i `DiMember( )` tada se obavljaju u vremenu $\mathcal{O}(1)$. No tu je i dalje već prije uočenu mana da prostor potreban za pohranjivanje ne ovisi o veličini samog rječnika nego o veličini tipa `elementype`. To znači da implementacija postaje neupotrebljiva ako je `elementtype` velik.

4.2.3 Implementacija rječnika pomoću liste

Opet se radi o ideji koju smo već razmatrali u odjeljku 4.1 kad smo se bavili implementacijom općenitog skupa. Dakle, rječnik shvatimo kao listu, koja može biti sortirana (u skladu s `<=`) ili nesortirana. Tu listu dalje prikazujemo pomoću polja ili pointera onako kako je bilo opisano u odjeljku 2.1.

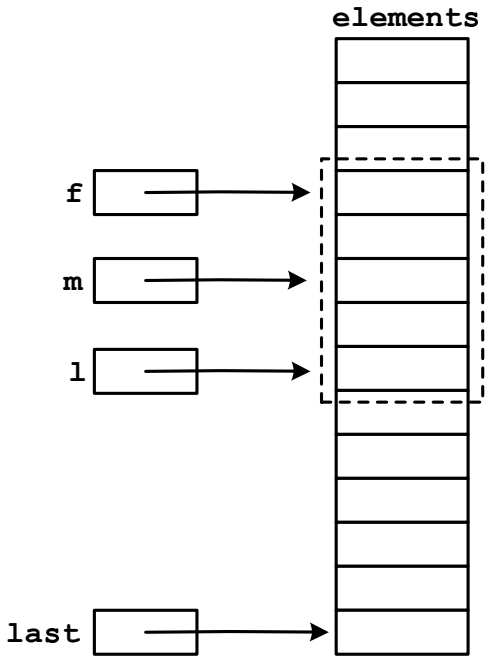
Od četiri moguće varijante detaljnije ćemo promotriti **sortiranu listu** prikazanu kao **polje**. Ta varijanta je pogodna za "statične" rječnike, dakle one za koje se često obavlja operacija `DiMember( )`, a rjeđe `DiInsert( )` ili `DiDelete( )`. Operaciju `DiMember( )` moguće je obaviti u vremenu $\mathcal{O}(\log_2 n)$, gdje je n duljina liste odnosno veličina rječnika. Da bi postigli takvo sublinearno vrijeme, služimo se algoritmom **binarnog traženja**.

Osnovni korak algoritma binarnog traženja sastoji se u tome da se traženi element `x` usporedi s elementom koji se nalazi na sredini sortirane liste `A`. Ovisno o rezultatu usporedbe `x` će odmah biti pronađen ili će se ustanoviti u kojoj polovici liste `A` (ispred ili iza srednjeg elementa) bi se on morao nalaziti. Isti korak po potrebi se ponavlja na odgovarajućoj polovici liste, zatim na odgovarajućoj četvrtini, itd. Postupak traje sve dok se `x` konačno ne pronađe ili dok razmatrani dio liste ne postane prazan.

Pretpostavimo da je tip `List` definiran kao u odjeljku 2.1 te da je tip `Dictionary` poistovjećen s tipom `List`. Tada implementacija operacije `DiMember( )` zasnovana na binarnom traženju izgleda kao što slijedi. Vidimo da programski kod za `DiMember( )` vjerno slijedi algoritam binarnog traženja. Osnovni korak binarnog traženja realiziran kao korak `while` petlje i dodatno je ilustriran je slikom 4.2. Trenutno razmatrani dio liste sastoji se od klijetki polja sa indeksima u rasponu od `f` do `l`. Indeks elementa na sredini razmatranog dijela dobiva se kao aritmetička sredina od `f` i `l`. Prazna lista prepoznaje se po tome što su se `f` i `l` "prekrižili", tj. `f` je postao veći od `l`.

```
int DiMember (elementtype x, Dictionary A) {
  int f, l; /* indeks prvog i zadnjeg elementa razmatrane podliste */
  int m; /* indeks srednjeg elementa razmatrane podliste */
```

```
  f = 0;
  l = A.last;
  while (f <= l) {
    m = (f+l)/2;
    if (A.elements[m] == x)
      return 1;
    else if (A.elements[m] < x)
      f = m+1;
    else /* A.elements[m] > x */
      l = m-1;
  }
  return 0;
}
```



Slika 4.2: Implementacija rječnika pomoću liste - operacija `DiMember( )`.

Za razliku od `DiMember( )`, preostale rječničke operacije se kod implementacije sortiranim poljem izvode sporije, tj. u linearnom vremenu. Naime, da bi se čuvala sortiranost, `DiInsert( )` mora ubaciti novi element na "pravo" mjesto u polje, što zahtijeva u najgorem slučaju $\mathcal{O}(n)$ prepisivanja elemenata. Slično, `DiDelete( )` zahtijeva $\mathcal{O}(n)$ prepisivanja zbog popunjavanja praznine nastale izbacivanjem.

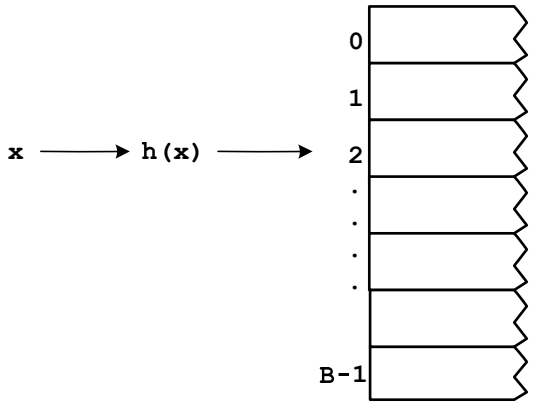
Ako se za naš rječnik često obavljaju operacije `DiInsert( )` i `DiDelete( )` a rjeđe `DiMember( )`, tada su pogodnije ostale tri varijante implementacije pomoću liste, dakle varijante s nesortiranom listom ili sa sortiranom vezanom listom. Kod sve tri varijante može se izbjeći prepisivanje elemenata prilikom ubacivanja i izbacivanja. Ipak, vrijeme za `DiInsert( )`, `DiDelete( )` i `DiMember( )` i dalje je $\mathcal{O}(n)$ zbog traženja elementa odnosno njegovog mjesta u listi.

4.2.4 Implementacija rječnika pomoću *hash* tablice

Implementaciju rječnika pomoću bit-vektora možemo promatrati kao bijekciju oblika: `elementtype` → memorija. Naime, svakoj mogućoj vrijednosti tipa `elementtype` pridružuje se njezin zasebni bit memorije. Takva implementacija u jednu ruku je idealna, jer se funkcije `DiInsert( )`, `DiDelete( )`, `DiMember( )` izvršavaju u vremenu $\mathcal{O}(1)$. S druge strane, već smo rekli da je ta implementacija obično neupotrebljiva jer zauzima prevelik komad memorije od kojeg se najveći dio uopće ne koristi. Kompromisno rješenje je umjesto bijekcije promatrati surjekciju na manji komad memorije. Tada se doduše više različitih vrijednosti može preslikati na istu adresu.

***Hash* funkcija** `h( )` je potprogram koji surjektivno preslikava skup `elementtype` na skup cijelih brojeva između 0 i `B` − 1, gdje je `B` cjelobrojna konstanta. ***Hash* tablica** je polje od `B` klijetki koje zovemo **pretinci**. Indeksi pretinaca su `0,1,2,...,B-1`.

Implementacija rječnika pomoću *hash* tablice zasniva se na tome da element `x` spremimo u pretinac s indeksom `h(x)`. Kasnije ga u istom pretincu i tražimo. Ideja je ilustrirana slikom 4.3.



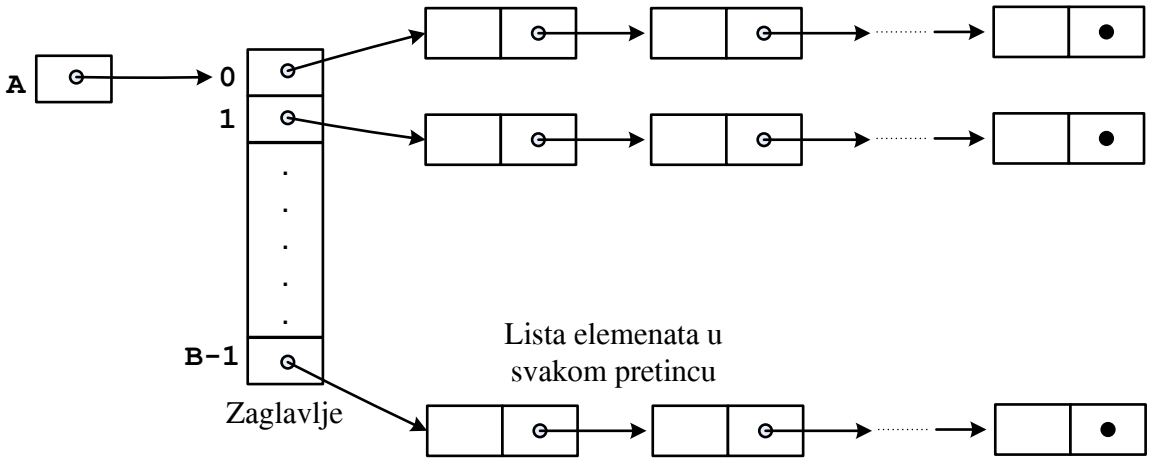
Slika 4.3: *Hash* funkcija i *hash* tablica.

Za efikasno funkcioniranje ove ideje važno je da bude što manje **kolizija**, to jest da se što rjeđe dešava da `h( )` dva različita `x`-a pošalje u isti pretinac. No kolizija će biti manje ako `h( )` jednoliko raspoređuje vrijednosti iz skupa `elementtype` na pretince `0,1,2,...,B-1`. U skladu s time, *hash* funcija smatra se to boljom što je način njezinog raspoređivanja jednolikiji. Primjerice, ako uzmemo da je `elementtype` skup svih nizova znakova duljine 10, tada se sljedeći `h( )` prema navedenom kriteriju može smatrati dobrom *hash* funkcijom:

```
int h (elementtype x) {
  int i, sum=0;
  for (i=0; i<10; i++) sum += x[i]; /* zbroj ascii kodova znakova */
  return sum % B; /* ostatak dijeljenja zbroja s B */
}
```

Postoje razne varijante implementacije pomoću *hash* tablice. One se razlikuju po građi pretinca tablice i po načinu izlaženja na kraj s kolizijama. Promotrit ćemo dvije varijante koje se zovu otvoreno odnosno zatvoreno haširanje.

Otvoreno "haširanje". Kod ove varijante haširanja pretinac je građen kao vezana lista u skladu sa slikom 4.4. Kad god treba novi element `x` ubaciti u i-ti pretinac, tada se i-ta vezana lista produlji još jednim zapisom. Na taj način, kapacitet jednog pretinca je neograničen i promjenjiv. Pointeri na početke vezanih listi organiziraju se u polje koje zovemo *zaglavlje*.



Slika 4.4: Građa *hash* tablice kod otvorenog haširanja.

Da bi otvoreno haširanje realizirali u C-u, potrebne su nam sljedeće definicije i funkcije.

```
#define B ...  /* pogodna konstanta */

typedef struct celltag {
  elementtype element;
  struct celltag *next; } celltype;

typedef celltype **Dictionary; /* početna adresa zaglavlja */

void DiMakeNull (Dictionary *Ap) {
  int i;
  for (i=0; i<B; i++) (*Ap)[i] = NULL;
}

int DiMember (elementtype x, Dictionary A) {
  celltype *current;
  current = A[h(x)]; /* adresa početka x-ovog pretinca */
  while (current != NULL)
    if (current->element == x)
      return 1;
    else
      current = current->next;
  return 0; /* x nije nađen */
}
```

```
void DiInsert (elementtype x, Dictionary *Ap) {
  int bucket;
  celltype *oldheader;
  if (DiMember(x,*Ap) == 0) {
    bucket = h(x);
    oldheader = (*Ap)[bucket];
    (*Ap)[bucket] = (celltype*)malloc(sizeof(celltype));
    (*Ap)[bucket]->element = x;
    (*Ap)[bucket]->next = oldheader;
  }
}

void DiDelete (elementtype x, Dictionary *Ap) {
  celltype *current, *temp;
  int bucket;
  bucket = h(x);
  if ( (*Ap)[bucket] != NULL) {
    if ( (*Ap)[bucket]->element == x) { /* x je u prvoj klijetki */
      temp = (*Ap)[bucket];
      (*Ap)[bucket] = (*Ap)[bucket]->next;
      free(temp);
    }
    else { /* x, ukoliko postoji, nije u prvoj klijetki */
      current = (*Ap)[bucket]; /* pokazuje prethodnu klijetku */
      while (current->next != NULL)
        if (current->next->element == x) {
          temp = current->next;
          current->next = current->next->next;
          free(temp);
          return;
        }
        else /* x još nije nađen */
          current = current->next;
    }
  }
}
```

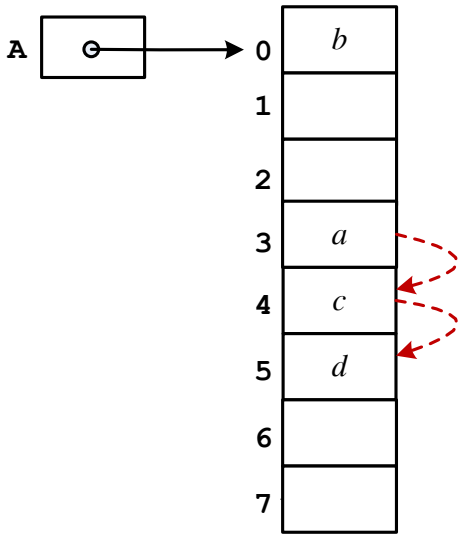
U gornjim definicijama konstanta `B` određuje veličinu *hash* tablice, dakle broj pretinaca. Tip `celltype` je tip za klijetke od kojih se grade vezane liste. Budući da će rječnik biti predočen zaglavljem (poljem pointera), a polje u C-u je isto što i adresa početnog elementa, tip `Dictionary` definira se kao pointer na pointer na `celltype`.

Da bi rječnik `A` (koji je zadan adresom `Ap`) pretvorila u prazni rječnik, funkcija `DiMakeNull( )` postavlja sve pointere na početke vezanih listi na vrijednost `NULL`, dakle pretvara sve vezane liste u prazne liste. Da bi u rječniku `A` našla element `x`, funkcija `DiMember( )` najprije računa `h(x)`, a zatim sekvencijalno čita samo `h(x)`-tu vezanu listu od `A` jer se samo u njoj može nalaziti `x`.

Funkcija `DiInsert( )` ubacuje element `x` u rječnik `A` (zadan adresom `Ap`) tako da na početak `h(x)`-te vezane liste od `A` ubaci novu klijetku s upisanim `x`-om . Klijetka se stvara dinamičkom alokacijom memorije. Ubacivanje se zaista provodi samo ako `x` nije već otprije pohranjen u `A`.

Da bi izbacila element `x` iz rječnika `A` (zadanog adresom `Ap`), funkcija `DiDelete( )` pretražuje `h(x)`-tu vezanu listu od `A`. Ako u toj listi postoji klijetka s `x` ona se isključuje iz liste, inače funkcija ne radi ništa. Isključena klijetka se "reciklira" tj. vraća u zalihu slobodne memorije. Postupak isključivanja razlikuje se u slučaju kad je klijetka na početku liste odnosno negdje drugdje u listi.

Zatvoreno "haširanje". Riječ je o varijanti haširanja gdje pretinac ima fiksni kapacitet. Zbog jednostavnosti uzimamo da u jedan pretinac stane jedan element. Dakle, *hash* tablica je polje klijetki tipa `elementtype`. Ako trebamo ubaciti element `x`, a klijetka `h(x)` je već puna, tada pogledamo na alternativne lokacije: `hh(1,x)`, `hh(2,x)`, `...` sve dok ne nađemo jednu praznu. Najčešće se funkcija `hh(i,x)` zadaje kao `hh(i,x) = (h(x)+i) % B`, dakle prazna lokacija traži se sekvencijalnim pregledom tablice - to se onda zove "linearno" haširanje. Na slici 4.5 nacrtana je situacija kada je `B` jednak 8, rječnik se gradi ubacivanjem elemenata a, b, c, d (u tom redoslijedu), pri čemu *hash* funkcija `h( )` redom daje vrijednosti 3, 0, 4 odnosno 3. Radi se o linearnom haširanju.



Slika 4.5: Djelomično popunjena *hash* tablica kod zatvorenog haširanja.

Kod zatvorenog haširanja važnu ulogu imaju prazni pretinci. Njih prepoznajemo po tome što sadrže posebnu rezerviranu vrijednost `EMPTY`, koja je različita od bilo kojeg elementa kojeg bi htjeli ubaciti. Traženje elementa `x` u tablici provodi se tako da gledamo pretince `h(x)`, `hh(1,x)`, `hh(2,x)`, `...` sve dok ne nađemo `x` ili `EMPTY`. Ovaj postupak traženja ispravan je samo ako nema izbacivanja elemenata. Da bismo ipak u nekom smislu mogli izbacivati, uvodimo još jednu rezerviranu vrijednost: `DELETED`. Element *poništavamo* tako da umjesto njega u dotični pretinac upišemo `DELETED`. Ta klijetka se kod traženja ne smatra praznom, no kasnije se kod novog ubacivanja može upotrijebiti kao da je prazna.

U nastavku slijede definicije tipova i funkcija koje su potrebne da bi zatvoreno haširanje realizirali u C-programu. Funkcije su napisane pod pretpostavkom da se koristi linearno haširanje.

```
#define EMPTY ...
#define DELETED ...
#define B ...

typedef elementtype *Dictionary; /* početna adresa polja */

void DiMakeNull (Dictionary *Ap) {
  int i;
  for (i=0; i<B; i++) (*Ap)[i] = EMPTY;
}

int locate (elementtype x, Dictionary A) {
  /* Prolazi tablicom od pretinca h(x) nadalje, sve dok ne nađe x
     ili EMPTY, ili dok se ne vrati na mjesto polaska.  Vraća
     indeks pretinca na kojem je stala iz bilo kojeg razloga */
  int initial, i;
  /* initial čuva h(x), i broji pretince koje smo prošli */
  initial = h(x);
  i = 0;
  while ((i<B) && (A[(initial+i)%B]!=x) && (A[(initial+i)%B]!=EMPTY))
    i++;
  return ((initial+i)%B);
}

int locate1 (elementtype x, Dictionary A) {
  /* slično kao locate, ali stane i onda kad naiđe na DELETED */
    ...
}

int DiMember (elementtype x, Dictionary A) {
  if ( A[locate(x,A)] == x ) return 1;
  else return 0;
}

void DiInsert (elementtype x, Dictionary *Ap) {
  int bucket;
  if (DiMember(x,*Ap)) return; /* x je već u A */
  bucket = locate1(x,*Ap);
  if ( ((*Ap)[bucket]==EMPTY) || ((*Ap)[bucket]==DELETED) )
    (*Ap)[bucket] = x;
  else
    exit(701); /* hash tablica se prepunila */
}
```

```
void DiDelete (elementtype x; Dictionary *Ap) {
  int bucket;
  bucket = locate(x,*Ap);
  if ( (*Ap)[bucket] == x) (*Ap)[bucket] = DELETED;
}
```

Vidimo da je najprije potrebno definirati konstante `EMPTY` i `DELETED` koje će nam označavati prazno odnosno poništeno mjesto u tablici. Također je potrebno definirati konstantu `B` koja određuje veličinu tablice, dakle broj pretinaca. Budući da je sama *hash* tablica polje elemenata, tip `Dictionary` definira se kao pointer na `elementtype`.

Pomoćne funkcije `locate( )` i `locate1( )` služe za pronalaženje pretinaca u tablici u skladu s pravilima linearnog haširanja. Dakle, `locate( )` prolazi poljem od indeksa `h(x)` nadalje, sve dok ne nađe `x`, ili praznu klijetku, ili dok se ne vrati na mjesto polaska. `locate( )` vraća indeks klijetke na kojoj je stala iz bilo kojeg od ovih razloga. Funkcija `locate1( )` radi slično kao `locate( )`, samo što stane i onda kad naiđe na poništenu klijetku.

Da bi `A` (zadan adresom `Ap`) pretvorila u prazni rječnik, funkcija `DiMakeNull( )` cijelo odgovarajuće polje popuni vrijednostima `EMPTY`. Da bi provjerila postojanje elementa `x` u rječniku `A`, funkcija `DiMember( )` najprije poziva pomoćnu funkciju `locate( )`. Naime, ako `x` postoji u `A`, `locate( )` će vratiti njegov indeks u polju, pa jedino što `DiMember( )` treba napraviti je provjeriti što piše na vraćenom indeksu.

Da bi ubacila element `x` u rječnik `A` (zadan adresom `Ap`), funkcija `DiInsert( )` najprije poziva pomoćnu funkciju `locate1( )`. Ako `x` već postoji u rječniku, `locate1( )` će vratiti njegov indeks u polju i tada `DiInsert( )` ne radi ništa. Ako `x` ne postoji u rječniku, `locate1( )` će vratiti indeks one prazne ili poništene klijetke u koju se po pravilima linearnog haširanja `x` treba upisati. `DiInsert( )` prepoznaje i izvanrednu situaciju prepunjenja, dakle situaciju kad u polju nema slobodnog mjesta za upis novog elementa - tada funkcija prekida program uz statusni kod 701.

Da bi izbacila element `x` iz rječnika `A` (zadanog adresom `Ap`), funkcija `DiDelete( )` najprije poziva pomoćnu funkciju `locate( )`. Ako `x` postoji u `A`, `locate( )` će vratitit njegov indeks u polju pa `DiDelete( )` na to mjesto upisuje `DELETED`. Inače `DiDelete( )` ne radi ništa.

Kod obje varijante *hash* tablice važno je da tablica bude **dobro dimenzionirana** u odnosu na rječnik koji se pohranjuje. Neka je n broj elemenata u rječniku. Preporučuje se da bude:

- $n \leq 2 \cdot \texttt{B}$ ... u slučaju otvorenog haširanja,
- $n \leq 0.9 \cdot \texttt{B}$ ... kod zatvorenog haširanja.

Dakle, broj pretinaca kod otvorenog haširanja treba biti takav da vezane liste u prosjeku nisu predugačke. S druge strane, broj pretinaca kod zatvorenog haširanja treba biti takav da u tablici ostane barem 10% slobodnog mjesta. Ako je sve ovo ispunjeno i ako je *hash* funkcija dovoljno ravnomjerna, tada možemo očekivati da će bilo koja od operacija `DiInsert( )`, `DiDelete( )`, `DiMember( )` zahtijevati svega nekoliko čitanja iz tablice. Ako se tablica previše napuni, treba je prepisati u novu, veću.

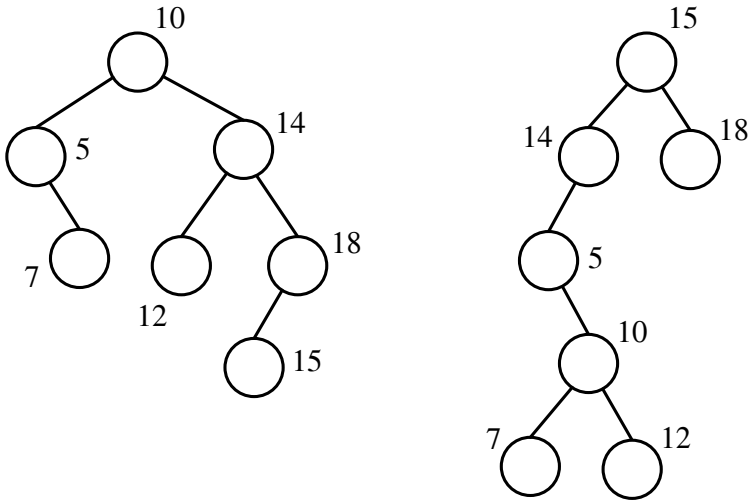
4.2.5 Implementacija rječnika pomoću binarnog stabla traženja

Najprije ćemo definirati jednu posebnu vrstu binarnog stabla. Binarno stablo T je **binarno stablo traženja** (*binary search tree*) ako su ispunjeni sljedeći uvjeti:

- Čvorovi od T su označeni podacima nekog tipa na kojem je definiran totalni uređaj.
- Neka je i bilo koji čvor od T. Tada su oznake svih čvorova u lijevom podstablu od i manje od oznake od i. Također, oznake svih čvorova u desnom podstablu od i su veće ili jednake od oznake od i.

Ideja implementacije je sljedeća: rječnik prikazujemo binarnim stablom traženja. Svakom elementu rječnika odgovara točno jedan čvor binarnog stabla i obratno. Element rječnika služi kao oznaka odgovarajućeg čvora binarnog stabla - kažemo da je element "spremljen" u tom čvoru. Primjetimo da će svi čvorovi našeg binarnog stabla imati različite oznake, makar se to ne zahtjeva u definiciji binarnog stabla traženja.

Na slici 4.6 vide se dva prikaza skupa $A = \{5, 7, 10, 12, 14, 15, 18\}$ pomoću binarnog stabla traženja. Dakle, prikaz nije jedinstven. Lagano se vidi da obilaskom binarnog stabla algoritmom `Inorder( )` dobivamo elemente skupa u sortiranom redoslijedu.



Slika 4.6: Prikazi skupa $A = \{5, 7, 10, 12, 14, 15, 18\}$ pomoću binarnog stabla traženja.

Samo binarno stablo može se prikazati pomoću pointera na način opisan u odjeljku 3.2. Tada su nam potrebne sljedeće definicije:

```
typedef struct celltag {
  elementtype element;
  struct celltag *leftchild;
  struct celltag *rightchild;
} celltype;                    /* čvor binarnog stabla */

typedef celltype *Dictionary; /* pointer na korijen */
```

Implementacija operacije `DiMakeNull( )` je trivijalna, svodi se na pridruživanje vrijednosti `NULL` pointeru. Operacija `DiMember( )` lako se implementira na sljedeći način zahvaljujući svojstvima binarnog stabla traženja:

```
int DiMember (elementtype x, Dictionary A) {
  if (A == NULL) return 0;
  else if (x == A->element) return 1;
  else if (x < A->element) return(DiMember(x,A->leftchild));
  else /* x > A->element */ return(DiMember(x,A->rightchild));
}
```

Dakle, da bi odgovorila pripada li element `x` rječniku `A`, funkcija `DiMember( )` gleda korijen odgovarajućeg binarnog stabla. Ako je binarno stablo prazno, tada `x` očito nije u `A`, pa funkcija vraća 0. Ako je `x` spremljen u korijenu binarnog stabla, tada `x` očito jest u `A` pa funkcija vraća 1. Inače se gleda da li je `x` manji ili veći od elementa u korijenu, pa se odgovor dobiva tako da se rekurzivnim pozivom iste funkcije isti `x` traži u lijevom odnosno desnom podstablu.

Operacija `DiInsert( )` radi slično. Ona poput `DiMember( )` traži mjesto u binarnom stablu gdje bi morao biti novi element te ubacuje novi čvor na to mjesto.

```
void DiInsert (elementtype x, Dictionary *Ap) {
  if (*Ap == NULL) {
    *Ap = (celltype*) malloc (sizeof(celltype));
    (*Ap)->element = x;
    (*Ap)->leftchild = (*Ap)->rightchild = NULL;
  }
  else if (x < (*Ap)->element) DiInsert(x,&((*Ap)->leftchild));
  else if (x > (*Ap)->element) DiInsert(x,&((*Ap)->rightchild));
  /* za x == (*Ap)->element ne radi ništa jer je x već u rječniku */
}
```

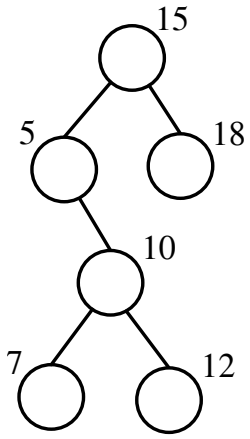
Dakle, da bi u rječnik `A` (predstavljen adresom `Ap`) ubacila element `x`, funkcija `DiInsert( )` gleda korijen odgovarajućeg binarnog stabla. Ako je binarno stablo prazno (nema korijena), stvara se nova klijetka s upisanim `x`-om i ona postaje korijen. Ako binarno stablo nije prazno (ima korijen), gleda se je li `x` manji ili veći od elementa u korijenu, pa se rekurzivnim pozivom iste funkcije `x` ubacuje u lijevo odnosno desno podstablo. U slučaju kad je `x` jednak elementu u korijenu funkcija ne radi ništa jer se `x` očito već nalazi u `A`.

Nešto je složenija operacija `DiDelete(x,&A)`. Imamo tri slučaja:

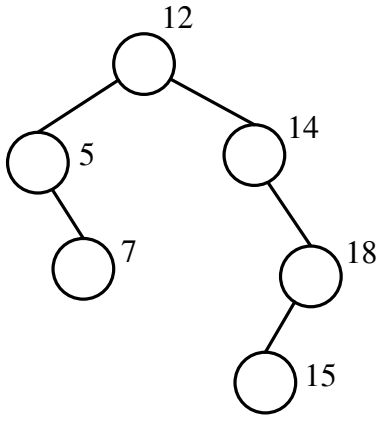
- Element `x` je u listu. Tada jednostavno izbacimo list iz stabla. Npr. to vrijedi za `x` jednak 15 u prvom binarnom stablu sa slike 4.6.

- Element `x` je u čvoru koji ima samo jedno dijete. Tada nadomjestimo čvor od `x` s njegovim djetetom. Npr. to radimo za `x` jednak 14 u drugom binarnom stablu sa slike 4.6. Čvor koji pohranjuje 14 se izbaci, a umjesto njega čvor od 5 postaje lijevo dijete čvora od 15. Rezultat se vidi na slici 4.7.

- Element `x` je u čvoru koji ima oba djeteta. Tada nađemo najmanji element `y` u desnom podstablu čvora od `x`. Izbacimo čvor od `y` (to mora biti jedan od dva prethodna slučaja). U čvor od `x` spremimo `y` umjesto `x`. Npr. za `x` jednak 10 u prvom binarnom stablu sa slike 4.6, izlazi da je `y` jednak 12. Izbacimo čvor od 12. U čvor koji sadrži 10 upišemo 12 i time izbrišemo 10. Rezultat je prikazan na slici 4.8.



Slika 4.7: Brisanje čvora s oznakom 14 iz drugog stabla sa slike 4.6.



Slika 4.8: Brisanje čvora s oznakom 10 iz prvog stabla sa slike 4.6.

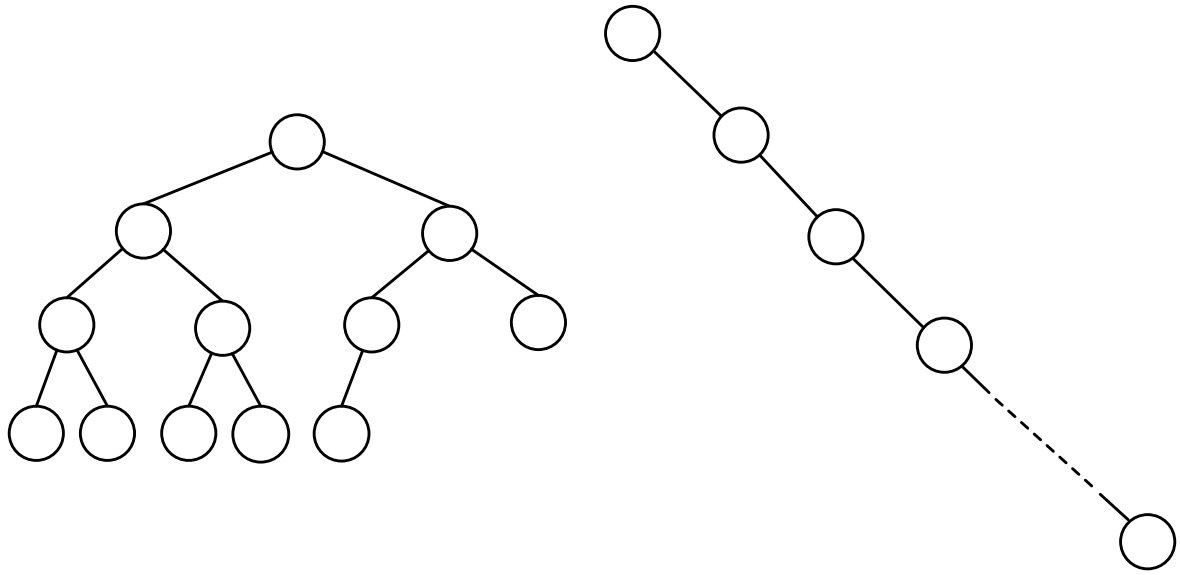
Sve se ovo spretno može zapisati u C-u ako uvedemo još jednu pomoćnu funkciju, nazovimo je `DiDeleteMin( )`. Ta funkcija iz zadanog nepraznog binarnog stabla traženja izbacuje čvor s najmanjim elementom te vraća taj najmanji element. Primijetimo da je čvor s najmanjim elementom ustvari "najljeviji" čvor. Slijedi programski kod za `DiDeleteMin( )`, a zatim i za `DiDelete( )`.

```
elementtype DiDeleteMin (Dictionary *Ap) {
  celltype *temp;
  elementtype minel;
  if ( (*Ap)->leftchild)==NULL ) {
    /* (*Ap) pokazuje najmanji element */
    minel = (*Ap)->element;
    temp = (*Ap); (*Ap) = (*Ap)->rightchild; free(temp);
  }
  else /* čvor kojeg pokazuje (*Ap) ima lijevo dijete */
    minel = DiDeleteMin( &((*Ap)->leftchild) );
  return minel;
}

void DiDelete (elementtype x, Dictionary *Ap) {
  celltype *temp;
  if ( *Ap != NULL )
    if ( x < (*Ap)->element )
      DiDelete( x, &((*Ap)->leftchild) );
    else if ( x > (*Ap)->element )
      DiDelete( x, &((*Ap)->rightchild) );
    /* ako dođemo ovamo, tada je x u čvoru kojeg pokazuje *Ap */
    else if ( ((*Ap)->leftchild==NULL) && ((*Ap)->rightchild==NULL)) {
      /* izbaci list koji čuva x */
      free(*Ap); *Ap = NULL;
    }
    else if ( (*Ap)->leftchild == NULL ) {
      /* nadomjestimo čvor od x s njegovim desnim djetetom */
      temp = *Ap; *Ap = (*Ap)->rightchild; free(temp);
    }
    else if ( (*Ap)->rightchild == NULL ) {
      /* nadomjestimo čvor od x s njegovim lijevim djetetom */
      temp = *Ap; *Ap = (*Ap)->leftchild; free(temp);
    }
    else
      /* postoje oba djeteta */
      (*Ap)->element = DiDeleteMin ( &((*Ap)->rightchild) );
}
```

U skladu s opisanom zadaćom, funkcija `DiDeleteMin( )` smije se koristiti samo onda kad je rječnik `A` (zadan adresom `Ap`) neprazan. Vidimo da funkcija svojim rekurzivnim samopozivanjem zapravo putuje po binarnom stablu od korijena prema lijevo dok god je to moguće, tj. dok se ne nađe u "najljevijem" čvoru koji sadrži najmanji element. Pronađeni najljeviji čvor isključuje se iz binarnog stabla, a element koji je u njemu pisao vraća se kao vrijednost funkcije. Isključivanje čvora nije problematično jer on nema lijevo dijete pa mora biti ili list ili čvor s jednim djetetom.

Da bi izbacila element `x` iz rječnika `A` (koji je zadan adresom `Ap`), funkcija `DiDelete( )` gleda korijen odgovarajućeg binarnog stabla. Ako je binarno stablo prazno (nema korijena), ne radi se ništa jer `x`-a sigurno nema u `A`. Ako je binarno stablo neprazno (ima korijen) gleda se je li `x` manji ili veći od elementa u korijenu, pa se rekurzivnim pozivom iste funkcije `x` izbacuje iz lijevog odnosno desnog podstabla. Ako je `x` jednak elementu u korijenu, tada funkcija isključuje korijen iz binarnog stabla. Isključenje korijena provodi se na ispravan način, ovisno o tome koliko djece taj korijen ima. Dakle, implementirani su svi prethodno opisani slučajevi izbacivanja. Slučaj s jednim djetetom implementiran je dvaput budući da to jedno dijete može biti lijevo ili desno. Slučaj s dvoje djece implementiran je tako da se pozivom `DiDeleteMin( )` izbaci najmanji element iz desnog podstabla, pa se zatiom taj najmanji element ponovo upiše u korijen.

Svaka od funkcija `DiMember( )`, `DiInsert( )`, `DiDelete( )` prolazi jednim putom od korijena binarnog stabla do nekog čvora. Zato je vrijeme izvršavanja svih operacija ograničeno visinom stabla. Neka rječnik ima n elemenata. Visina stabla tada varira između $\lceil \log_2(n+1) \rceil - 1$ i $n-1$. Ekstremni slučajevi prikazani su na slici 4.9: to su potpuno binarno stablo i "ispruženo" binarno stablo - lanac.



Slika 4.9: Ekstremni slučajevi visine binarnog stabla.

Dakle, vrijeme izvršavanja operacija varira između $\mathcal{O}(\log n)$ i $\mathcal{O}(n)$. Pitamo se koja ocjena je vjerodostojnija? Može se dokazati da vrijedi sljedeća tvrdnja:

> Neka je binarno stablo traženja od n čvorova stvoreno od praznog binarnog stabla n-strukom primjenom operacije `DiInsert( )`. Pritom je bilo koji redoslijed ubacivanja elemenata jednako vjerojatan. Tada očekivano vrijeme izvršavanja za operaciju `DiInsert( )`, `DiDelete( )` ili `DiMember( )` iznosi $\mathcal{O}(\log n)$.

Znači, imamo jake razloge da očekujemo vrijeme izvršavanja oblika $\mathcal{O}(\log n)$. Doduše, nemamo čvrste garancije za to - ekstremni slučajevi i dalje su mogući.

Opisana implementacija samo je jedna iz porodice sličnih, gdje se rječnik prikazuje pomoću neke "stablaste" strukture. Poznate su još i implementacije pomoću AVL-stabla, B-stabla, 2-3-stabla. Kod spomenute tri implementacije automatski se čuva "balansiranost" stabla, tj. svojstvo da je njegova visina $\mathcal{O}(\log n)$. Time se garantira efikasno obavljanje osnovnih operacija. Ipak, operacije `DiInsert( )` i `DiDelete( )` znatno su kompliciranije nego kod nas. Naime, te operacije preuzimaju zadatak pregradnje stabla kad god se ono suviše "iskrivi".

4.3 Prioritetni red

U ovom odjeljku bavimo se prioritetnim redom. To je opet jedan apstraktni tip koji se pojavljuje u važnim primjenama i koji se može shvatiti kao posebna vrsta skupa. Slično kao kod rječnika, posebnost prioritetnog reda je u tome što se kod njega smiju koristiti samo neke jednostavnije skupovne operacije. No popis tih operacija nešto je drukčiji nego kod rječnika. U odjeljku najprije navodimo svojstva i primjene prioritetnog reda, a zatim raspravljamo o njegovim implementacijama. Najvažnija je implementacija pomoću takozvane hrpe, ustvari jedne vrste binarnog stabla.

4.3.1 Svojstva i primjene prioritetnog reda

U nekim algoritmima pojavljuje se skup čijim su elementima pridruženi cijeli ili realni brojevi - prioriteti. Operacije su: ubacivanje novog elementa te izbacivanje elementa s najmanjim (najvažnijim) prioritetom. Takav skup nazivamo **prioritetni red** (*priority queue*).

U nastavku slijede primjeri korištenja prioritetnog reda. Opet su neki od njih iz običnog života, a neki iz računarstva.

- Ambulanta obiteljske medicine funkcionira tako da pacijenti ulaze u čekaonicu te iz nje odlaze liječniku na pregled. Uobičajeno pravilo kaže da je prvi na redu za liječnika onaj koji je prvi ušao u čekaonicu - tada se čekaonica ponaša kao običan red. No može se uvesti i pravilo da je prvi na redu onaj čije zdravstveno stanje je najteže, bez obzira kad je došao - tada imamo prioritetni red. Prioriteti odgovaraju zdravstvenim stanjima, s time da manji brojevi označavaju teža stanja.

- Tramvaj vozi gradom. Na svakoj stanici ulaze i izlaze putnici. Neki putnici zadržavaju se dulje u tramvaju, a neki kraće, ovisno o stanici gdje trebaju izaći. Dakle, redoslijed izlazaka ne poklapa se s redoslijedom ulazaka. Cijeli tramvaj možemo opisati kao prioritetni red za putnike. Prioritet putnika je redni broj njegove izlazne stanice, brojeno na cjelokupnoj ruti tramvaja. Putnici ulaze u proizvoljnom redoslijedu, no izlaze u redoslijedu svojih prioriteta.

- Operacijski sustav računala u stanju je izvoditi više procesa (programa) u isto vrijeme. To vidimo npr. onda kad na zaslonu otvorimo više prozora pa u svakom od njih pokrenemo neku drugu aktivnost. Istovremenost je, naravno, samo prividna jer su resursi računala (npr. procesor ili memorija) ograničeni. Privid se postiže tako da se računalo velikom brzinom prebacuje s jednog procesa na drugi. Koordinacija rada više procesa ostvaruje se tako da se procesima pridružuju prioriteti, te da operacijski sustav čuva prioritetni red procesa spremnih za izvođenje.

Čim se oslobode resursi, operacijski sustav uzima iz reda proces s najmanjim (najvažnijim) prioritetom, izvodi ga neko vrijeme, prekida ga, te ga nešto kasnije ponovo vraća u red. Procesi s najmanjim (najvažnijim) prioritetom su oni koji čine sam operacijski sustav ili oni koji zahtjevaju najbrži odziv. Korisnički programi obično imaju slabiji prioritet.

Spomenute probleme gdje se pojavljuju prioriteti možemo bez gubitka općenitosti svesti na jednostavniji problem ubacivanja elemenata u skup te izbacivanja *najmanjeg* elementa. Naime, umjesto originalnih elemenata x, možemo promatrati uređene parove (prioritet(x), x). Za uređene parove definiramo leksikografski uređaj:

(prioritet(x_1), x_1) je manji-ili-jednak od (prioritet(x_2), x_2) ako je (prioritet(x_1) manji od prioritet(x_2)) ili (prioritet(x_1) jednak prioritet(x_2) i x_1 manji-ili-jednak od x_2).

Ova dosjetka opravdava sljedeću definiciju apstraktnog tipa podataka za prioritetni red.

Apstraktni tip podataka PriorityQueue

`elementtype` ... bilo koji tip s totalnim uređajem `<=`.

`PriorityQueue` ... podatak ovog tipa je konačan skup čiji elementi su (međusobno različiti) podaci tipa `elementtype`.

`PrMakeNull(&A)` ... funkcija pretvara prioritetni red `A` u prazni skup.

`PrEmpty(A)` ... funkcija vraća "istinu" ako je prioritetni red `A` prazan, inače vraća "laž".

`PrInsert(x,&A)` ... funkcija ubacuje element `x` u prioritetni red `A`, tj. mijenja `A` u `A` $\cup \{$`x`$\}$.

`PrDeleteMin(&A)` ... funkcija iz prioritetnog reda `A` izbacuje najmanji element i vraća taj izbačeni element. Nije definirana ako je `A` prazan.

Primjetimo da se apstraktni tip PriorityQueue može smatrati restrikcijom apstraktnog tipa Set. Naime funkcija `PrDeleteMin( )` zapravo je kombinacija od `SeMin( )` i `SeDelete( )`. Zato nije čudno što se za implementaciju prioritetnog reda koriste slične strukture podataka kao za skup ili rječnik. U nastavku opisujemo nekoliko takvih implementacija.

4.3.2 Implementacija prioritetnog reda pomoću sortirane vezane liste

Zasniva se na ideji da prioritetni red prikažemo kao listu. Od raznih varijanti najbolja se čini sortirana vezana lista. U toj varijanti `PrDeleteMin( )` svodi se na pronalaženje i izbacivanje prvog elementa u listi, pa ima vrijeme $\mathcal{O}(1)$. No `PrInsert( )` mora ubaciti novi element na "pravo mjesto" u smislu sortiranog redoslijeda, što znači da u najgorem slučaju mora pročitati cijelu listu. Zato `PrInsert( )` ima vrijeme $\mathcal{O}(n)$, gdje je n broj elemenata u prioritetnom redu.

4.3.3 Implementacija prioritetnog reda pomoću binarnog stabla traženja

Implementacija izgleda sasvim isto kao za rječnik - vidi odjeljak 4.2. Za realizaciju operacija `PrInsert( )`, `PrDeleteMin( )` i `PrMakeNull( )` mogu se upotrijebiti preimenovane verzije prije napisanih potprograma `DiInsert( )`, `DiDeleteMin( )` odnosno `DiMakeNull( )`. Funkcija `PrEmpty( )` je trivijalna.

4.3.4 Implementacija prioritetnog reda pomoću hrpe

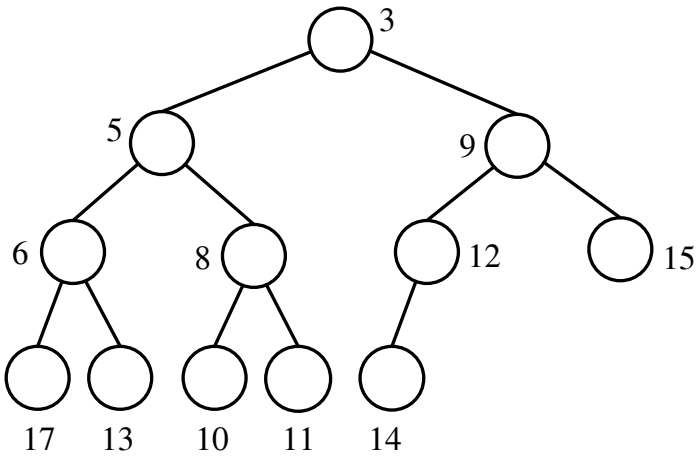
Ova implementacija donekle je slična prethodnoj jer prioritetni red opet prikazuje binarnim stablom. No razlika je u tome što ovaj put koristimo drukčiju vrstu binarnog stabla. Riječ je vrsti koja je posebno pogodna za prikaz prioritetnog reda i koju zovemo hrpa. Slijedi definicija.

Potpuno binarno stablo T je **hrpa** (gomila, *heap*), ako su ispunjeni sljedeći uvjeti:

- Čvorovi od T su označeni podacima nekog tipa na kojem je definiran totalni uređaj.
- Neka je i bilo koji čvor od T. Tada je oznaka od i manja ili jednaka od oznake bilo kojeg djeteta od i.

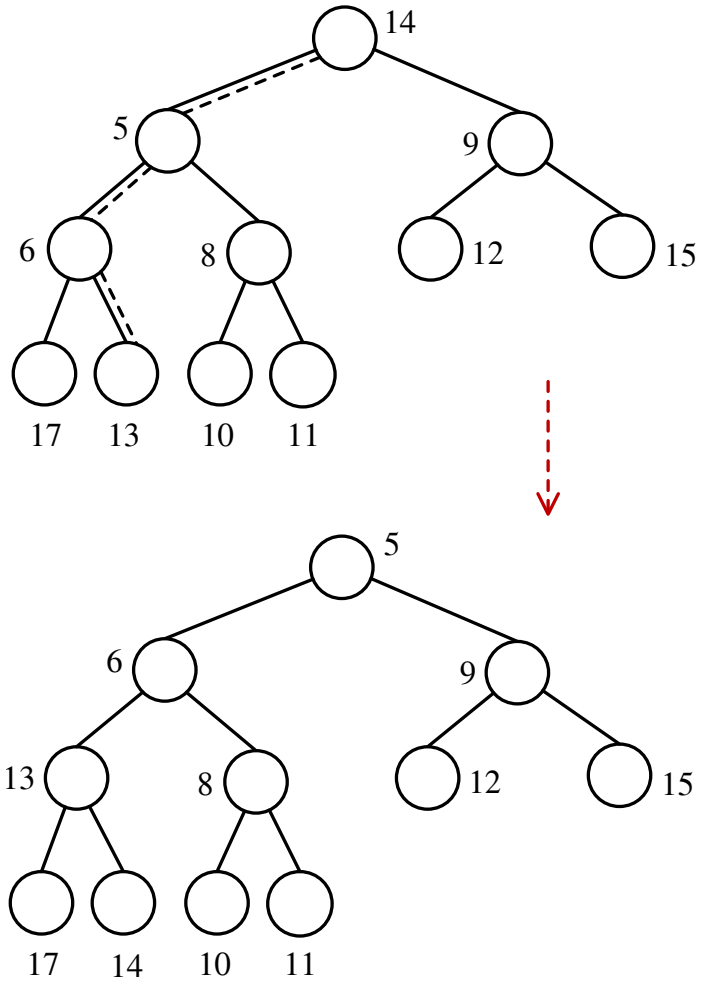
Implementacija se zasniva na ideji da prioritetni red opišemo hrpom. Svakom elementu reda odgovara točno jedan čvor hrpe i obratno. Element reda služi kao oznaka odgovarajućeg čvora hrpe - kažemo da je element "spremljen" u tom čvoru. Primijetimo da će svi čvorovi imati različite oznake, makar se to ne zahtijeva u definiciji hrpe.

Na slici 4.10 vidi se prikaz prioritetnog reda $A = \{3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 17\}$. Prikaz nije jedinstven. Iz svojstava hrpe slijedi da se najmanji element mora nalaziti u korijenu.



Slika 4.10: Prikaz prioritetnog reda $A = \{3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 17\}$ hrpom.

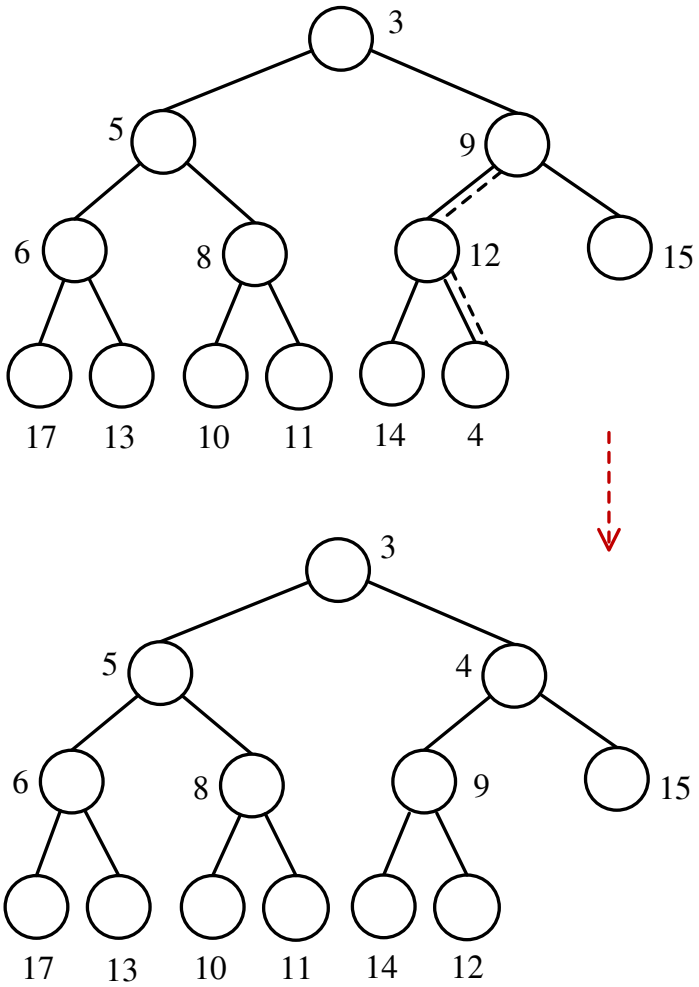
Da bismo obavili operaciju `PrDeleteMin( )`, vraćamo element iz korijena. Budući da korijen ne možemo samo tako izbaciti (binarno stablo bi se raspalo), izbacujemo zadnji čvor na zadnjoj razini, a njegov element stavimo u korijen. Time se sigurno pokvarilo svojstvo hrpe. Popravak se obavlja ovako: zamijenimo element u korijenu i manji element u korijenovom djetetu, zatim zamijenimo element u djetetu i manji element u djetetovom djetetu, ... , itd, dok je potrebno. Niz zamjena ide najdalje do nekog lista. Učinak `PrDeleteMin( )` na hrpu sa slike 4.10 vidi se na slici 4.11.



Slika 4.11: Izbacivanje najmanjeg elementa iz hrpe sa slike 4.10.

Da bismo obavili operaciju `PrInsert( )`, stvaramo novi čvor na prvom slobodnom mjestu zadnje razine te stavljamo novi element u taj novi čvor. Time se možda pokvarilo svojstvo hrpe. Popravak se obavlja ovako: zamijenimo element u novom čvoru i element u roditelju novog čvora, zatim zamijenimo element u roditelju i element u roditeljevom roditelju,..., itd., dok je potrebno. Niz zamjena ide najdalje do korijena. Učinak ubacivanja elementa 4 u hrpu sa slike 4.10 vidi se na slici 4.12.

Da bismo implementaciju razradili do kraja, potrebno je odabrati strukturu podataka za prikaz hrpe. S obzirom da je hrpa potpuno binarno stablo, možemo koristiti prikaz pomoću polja - vidi odjeljak 3.2. Dakle, potrebne su sljedeće definicije u C-u:



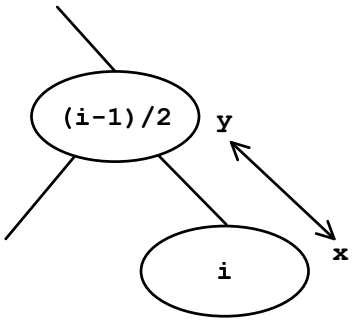
Slika 4.12: Ubacivanje elementa 4 u hrpu sa slike 4.10.

```
# define MAXSIZE ...  /* dovoljno velika konstanta */

typedef struct {
  elementtype elements[MAXSIZE];
  int last;
} PriorityQueue;
```

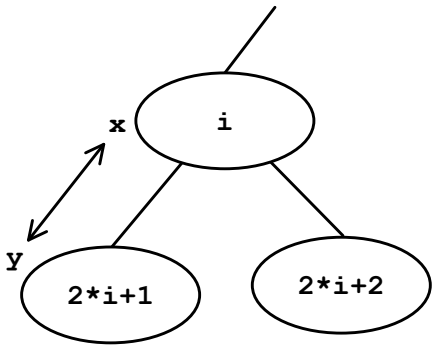
Slično kao u odjeljku 3.2, konstanta `MAXSIZE` postavlja gornju ogradu na broj čvorova u hrpi odnosno broj elemenata u prioritetnom redu. Sam prioritetni red poistovjećuje se sa `struct`-om koji se sastoji od jednog polja duljine `MAXSIZE` i jednog kursora. Polje služi za spremanje oznaki čvorova hrpe, dakle za spremanje elemenata prioritetnog reda. Kursor pokazuje gdje se zadnji čvor hrpe nalazi u polju.

U ovoj implementaciji prioritetnog reda potprogrami za `PrMakeNull( )` i `PrEmpty( )` su trivijalni pa ih nećemo ispisivati. Funkcije `PrInsert( )` i `PrDeleteMin( )` nešto su kompliciranije i svode se na manipulacije s elementima u polju. U nastavku slijedi njihov programski kod. Slike 4.13 i 4.14 služe kao dodatna ilustracija njihovog rada.

```
void PrInsert (elementtype x, PriorityQueue *Ap) {
  int i;
  elementtype temp;
  if ( Ap->last >= MAXSIZE-1 )
    exit(801); /* prioritetni red se prepunio */
  else {
    (Ap->last)++;
    Ap->elements[Ap->last] = x; /* novi čvor s elementom x */
    i = Ap->last; /* i je indeks čvora u kojem je x */
    while ( (i>0) && (Ap->elements[i] < Ap->elements[(i-1)/2]) ) {
      /* operator && u jeziku C je kondicionalan!  */
      temp = Ap->elements[i];
      Ap->elements[i] = Ap->elements[(i-1)/2];
      Ap->elements[(i-1)/2] = temp;
      i = (i-1)/2;
    }
  }
}
```



Slika 4.13: Implementacija prioritetnog reda pomoću hrpe - operacija `PrInsert( )`.



Slika 4.14: Implementacija prioritetnog reda pomoću hrpe - operacija `PrDeleteMin( )`.

```
elementtype PrDeleteMin (PriorityQueue *Ap) {
  int i, j;
  elementtype minel, temp;
  if ( Ap->last < 0 )
    exit(802); /* prioritetni red je prazan */
  else {
    minel = Ap->elements[0]; /* najmanji element je u korijenu */
    Ap->elements[0] = Ap->elements[Ap->last];
    /* vrijednost iz zadnjeg čvora prepisuje se u korijen */
    (Ap->last)--; /* izbacujemo zadnji čvor */
    i = 0;
    /* i je indeks čvora gdje se nalazi element iz izbačenog čvora */
    while ( i <= (Ap->last+1)/2-1 ) {
      /* mičemo element u čvoru i prema dolje */
      if ((2*i+1==Ap->last) || (Ap->elements[2*i+1]<Ap->elements[2*i+2]))
        j = 2*i+1;
      else
        j = 2*i+2; /* j je dijete od i koje sadrži manji element */
      if ( Ap->elements[i] > Ap->elements[j] ) {
        /* zamijeni elemente iz čvorova i,j */
        temp = Ap->elements[i];
        Ap->elements[i] = Ap->elements[j];
        Ap->elements[j] = temp;
        i = j;
      }
      else
        return(minel); /* ne treba dalje pomicati */
    }
    return(minel); /* pomicanje je došlo sve do lista */
  }
}
```

Vidimo da naše funkcije `PrInsert( )` i `PrDeleteMin( )` vjerno slijede prethodno opisane algoritme ubacivanja elemenata u hrpu odnosno izbacivanja najmanjeg elementa iz hrpe. S obzirom da je sama hrpa prikazana poljem, manevriranje po hrpi obavlja se onako kako je bilo objašnjeno pri kraju odjeljka 3.2. Dakle, roditelja čvora iz i-te kliijetke polja tražimo u klijetki s indeksom $\lfloor (i-1)/2 \rfloor$ - vidi sliku 4.13. Ako je i jednak 0, tada roditelj čvora i ne postoji. Slično, lijevo i desno dijete čvora iz i-te klijetke pronalazimo u klijetkama s indeksima $(2i+1)$ i $(2i+2)$ - vidi sliku 4.14. Ako $(2i+1)$ ili $(2i+2)$ ispadne veće od vrijednosti kursora `last`, tada dotično dijete čvora i ne postoji. Zadnji unutrašnji čvor je onaj s indeksom $\lfloor (\texttt{last}+1)/2 \rfloor - 1$.

Slično kao u mnogom drugim implementacijama, i u ovoj implementaciji postoje izvanredne situacije kad funkcije ne mogu izvršiti svoj zadatak pa prekidaju program uz dojavu statusnog koda. Zaista, `PrInsert( )` prepoznaje izvanrednu situaciju prepunjenja polja i tada vraća kod 801. Za `PrDeleteMin( )` izvanredna situacija je prazno polje iz kojeg se nema što izbaciti - odgovarajući kod je 802.

Primjetimo da funkcija `PrInsert( )` ima jednu manjkavost: ona stvara novi čvor s elementom `x` čak i onda kad `x` već jeste u prioritetnom redu `A`. Ova manjkavost obično nam ne smeta jer u primjenama najčešće ubacujemo samo one elemente za koje smo sigurni da se ne nalaze u prioritetnom redu.

Potprogrami `PrInsert( )` i `PrDeleteMin( )` obilaze jedan put u potpunom binarnom stablu. Zato je njihovo vrijeme izvršavanja u najgorem slučaju $\mathcal{O}(\log n)$, gdje je n broj čvorova stabla (odnosno broj elemenata u prioritetnom redu). Ovo je bolja ocjena nego za implementaciju pomoću binarnog stabla traženja gdje smo imali logaritamsko vrijeme samo u prosječnom slučaju. Zaključujemo da je implementacija prioritetnog reda pomoću hrpe bolja od implementacije prioritetnog reda pomoću binarnog stabla traženja. Prednost binarnog stabla traženja pred hrpom je: mogućnost efikasnog implementiranja dodatnih skupovnih operacija: `SeMember( )`, `SeDelete( )`, `SeMin( )`, `SeMax( )`.

4.4 Preslikavanje i binarna relacija

U ovom odjeljku proučavamo apstraktne tipove podataka koji odgovaraju matematičkom pojmu preslikavanja odnosno binarne relacije. Budući se i preslikavanje i relacija mogu shvatiti kao skupovi uređenih parova podataka, odgovarajući apstraktni tipovi srodni su apstraktnom tipu za skup te se mogu implementirati slično kao npr. rječnik. U odjeljku najprije navodimo svojstva i primjene preslikavanja te raspravljamo o njegovim implementacijama. Zatim na sličan način obrađujemo binarnu relaciju.

4.4.1 Svojstva i primjene preslikavanja

Često je potrebno pamtiti pridruživanja među podacima koja se mogu opisati matematičkim pojmom preslikavanja (funkcije). Naša definicija bit će malo općenitija nego što je uobičajeno u matematici, naime dozvoljavat ćemo da je preslikavanje definirano samo na dijelu domene.

Preslikavanje (*mapping*) M je skup uređenih parova oblika (d, r), gdje su svi d-ovi podaci jednog tipa, a svi r-ovi podaci drugog tipa. Pritom za zadani d u M postoji najviše jedan par (d, r).

Prvi tip nazivamo **domena** ili područje definicije (*domain*), a drugi tip zovemo **kodomena** ili područje vrijednosti (*range*). Ako za zadani d preslikavanje M sadrži par (d, r), tada taj jedinstveni r označavamo s $r = M(d)$ i kažemo da je $M(d)$ definirano. Ako za zadani d preslikavanje M ne sadrži par (d, r), kažemo da $M(d)$ nije definirano.

Slijede primjeri preslikavanja. Oni donekle liče na primjere za rječnik, no malo su kompliciraniji.

- Dok je pravopis služio kao primjer za rječnik, rječnik stranih riječi može nam biti primjer za preslikavanje. Riječnik stranih riječi je takav popis gdje uz svaku stranu riječ stoji i njezin prijevod ili tumačenje. Korištenje se sastoji ne samo od pronalaženja strane riječi na popisu, nego i od čitanja pripadnog prijevoda. Dakle zaista je riječ o preslikavanju, gdje domenu čine strane riječi, a kodomenu prijevodi.

- Popis korisnika višekorisničkog računala već smo spominjali kao primjer za rječnik. No to je bilo zato što smo pretpostavljali da korisnici nemaju svoje lozinke. Zajedno s lozinkama (*passwords*), popis korisnika pretvara se u preslikavanje. Kod tog preslikavanja, domenu čine korisnička imena, a kodomenu njihove lozinke. Kad se korisnik prijavljuje za rad na računalu, on najprije upisuje svoje korisničko ime, a zatim lozinku. Operacijski sustav primjenjuje preslikavanje da bi na osnovi upisanog imena sam odredio lozinku. Korisniku se dozvoljava rad samo onda ako se lozinka koju je on upisao poklapa s onom koju je utvrdio operacijski sustav.

- Zapisi u bazi podataka obično sadrže jedan važan podatak koji se zove *primarni ključ*. Primjerice, u zapisu o stanovniku nekog grada primarni ključ mogao bi biti OIB tog stanovnika. Fizička organizacija baze omogućuje da se na osnovi primarnog ključa brzo dođe do ostalih podataka iz zapisa. Dakle, ako znamo OIB neke osobe, tada trenutačno možemo dobiti njezino prezime, ime, adresu prebivališta, itd. Odnos između primarnog ključa i ostalih podataka u zapisu može se shvatiti kao preslikavanje. Domenu čine vrijednosti primarnog ključa, a kodomenu ostali podaci. U slučaju stanovnika grada, domenu čine OIB-i, a kodomenu prezimena, imena, adrese, itd.

Iz navedenih primjera vidljivo je da je najvažnija operacija s preslikavanjem njegovo izvrednjavanje, dakle pronalaženje jedinstvenog elementa kodomene koji je pridružen zadanom elementu domene. Također su nužne operacije kojima se samo preslikavanje definira ili se ta definicija mijenja. Kad to sve uzmemo u obzir, dobivamo ovakav apstraktni tip podataka.

Apstraktni tip podataka Mapping

`domain` ... bilo koji tip (domena).

`range` ... bilo koji tip (kodomena).

`Mapping` ... podatak tipa `Mapping` je preslikavanje čiju domenu čine podaci tipa `domain`, a kodomenu podaci tipa `range`.

`MaMakeNull(&M)` ... funkcija pretvara preslikavanje `M` u nul-preslikavanje, tj. takvo koje nije nigdje definirano.

`MaAssign(&M,d,r)` ... funkcija definira `M(d)` tako da bude `M(d)` jednako `r`, bez obzira da li je `M(d)` prije bilo definirano ili nije.

`MaDeassign(&M,d)` ... funkcija uzrokuje da `M(d)` postane nedefinirano, bez obzira da li je `M(d)` prije bilo definirano ili nije.

`MaCompute(M,d,&r)` ... ako je `M(d)` definirano, tada funkcija vraća "istinu" i pridružuje varijabli `r` vrijednost `M(d)`, inače funkcija vraća "laž".

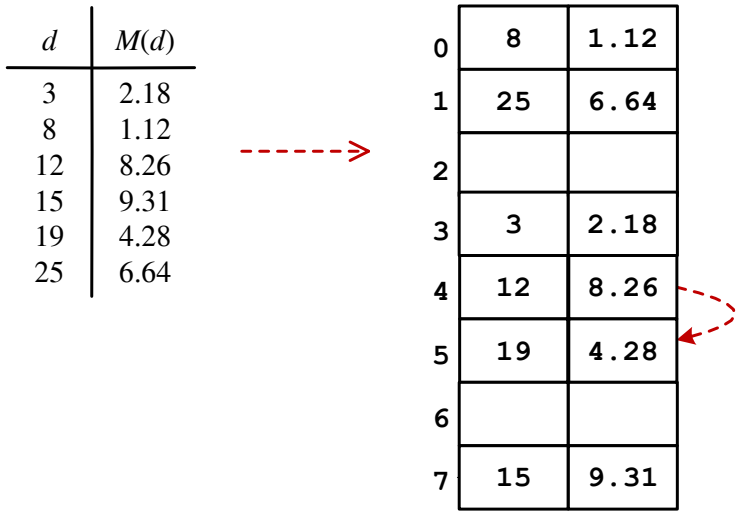
Za implementaciju preslikavanja koriste se iste strukture podataka kao za rječnik, dakle: polje, lista, *hash* tablica, binarno stablo traženja. U skladu s definicijom, preslikavanje M pohranjujemo kao skup uređenih parova oblika $(d, M(d))$, gdje d prolazi svim podacima za koje je $M(d)$ definirano. Pritom se mjesto uređenog para $(d, M(d))$ u strukturi određuje samo na temelju d, tj. onako kao kad bi se pohranjivao sam d. To omogućuje da kasnije pronađemo par $(d, M(d))$ na osnovi zadanog d. Operacije

`MaMakeNull( )`, `MaAssign( )`, `MaDeassign( )` i `MaCompute( )` iz apstraktnog tipa Mapping implementiraju se analogno kao `DiMakeNull( )`, `DiInsert( )`, `DiDelete( )` odnosno `DiMember( )` iz apstraktnog tipa Dictionary. Ove ideje ilustrirat ćemo konkretnim primjerima.

4.4.2 Implementacija preslikavanja *hash* tablicom ili binarnim stablom

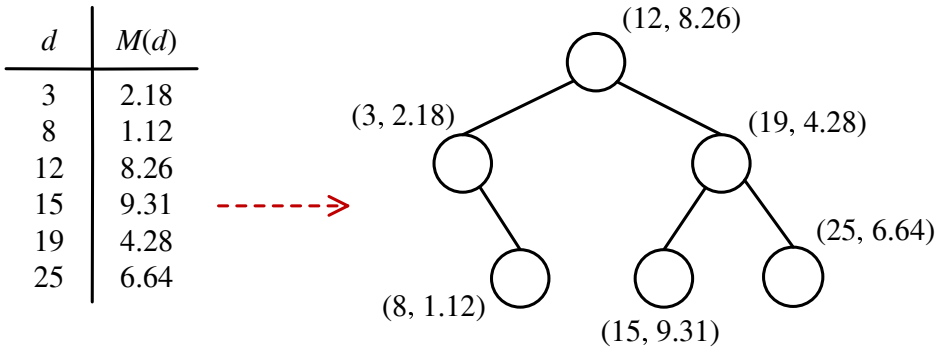
Promatramo preslikavanje M čija domena je tip `int`, a kodomena tip `float`. M je zadano tablicom koja se vidi na lijevom dijelu slike 4.15 ili slike 4.16. Za cijele brojeve d koji se ne pojavljuju u tablici $M(d)$ nije definirano. M tada shvaćamo kao skup $M = \{(3, 2.18), (8, 1.12), \ldots, (25, 6.64)\}$. Taj skup shvaćamo kao rječnik i prikazujemo ga na načine opisane u odjeljku 4.2.

Npr. možemo koristiti prikaz pomoću zatvorene *hash* tablice s `B` pretinaca. U svaki pretinac stane jedan uređeni par oblika $(d, M(d))$. *Hash* funkcija `h( )` preslikava područje definicije od M na skup $\{0, 1, 2, \ldots, \texttt{B} - 1\}$. Uređeni par $(d, M(d))$ sprema se u pretinac `h(`d`)` te se traži u istom pretincu. Konkretno, ako uzmemo `B = 8`, zadamo `h(x) = x % 8`, podatke ubacujemo u pretince u redoslijedu kako su zadani te u slučaju kolizije primjenjujemo linearno haširanje, tada *hash* tablica izgleda kao na slici 4.15.



Slika 4.15: Preslikavanje M prikazano pomoću zatvorene *hash* tablice.

Dalje, M možemo prikazati pomoću binarnog stabla traženja. Čitavi uređeni parovi $(d, M(d))$ smještaju se u čvorove stabla, no grananje u stablu se provodi samo na osnovi d. Dakle, vrijedi uređaj za oznake čvorova: $(d_1, M(d_1))$ manje-ili-jednako $(d_2, M(d_2))$ ako i samo ako je d_1 manji-ili-jednak d_2. Ubacivanjem podataka u redoslijedu kako su bili zadani dobivamo binarno stablo kao na slici 4.16.



Slika 4.16: Preslikavanje M prikazano pomoću binarnog stabla traženja.

4.4.3 Svojstva i primjene binarne relacije

Često je potrebno pamtiti pridruživanja među podacima koja su općenitija od onih obuhvaćenih pojmom "preslikavanje". Tada govorimo o (binarnoj) relaciji.

Binarna relacija R, ili kraće **relacija** (*relation*) R, je skup uređenih parova oblika (d_1, d_2), gdje se kao d_1 pojavljuju podaci jednog tipa, a kao d_2 podaci drugog tipa. Ova dva tipa nazivamo **prva** odnosno **druga domena** (*domains*). Ako relacija R za zadani d_1 i d_2 sadrži uređeni par (d_1, d_2), tada kažemo da je d_1 u relaciji R s d_2 i pišemo $d_1\ R\ d_2$. U protivnom kažemo da d_1 nije u relaciji R s d_2 i pišemo $d_1 \not R\ d_2$.

Binarne relacije vrlo često susrećemo i u običnom životu i u računarstvu. Slijedi nekoliko primjera.

- Studenti na fakultetu upisuju izborne predmete (kolegije). Odnos između studenata i predmeta predstavlja relaciju. Prva domenu čini skup studenata, a drugu skup predmeta. Relaciju koristimo u oba smjera: dakle za zadanog studenta pronalazimo koje sve predmete je on upisao, a za zadani predmet dobivamo popis studenata koji su ga upisali. Relaciju je potrebno ažurirati svaki put kad neki student upiše neki novi predmet ili poništi neki od prije upisanih predmeta.

  Sličan primjer relacije dobili bi promatranjem odnosa između filmova i cineplex dvorana koje ih prikazuju, ili promatranjem odnosa između zračnih luka i aviokompanija koje slijeću na njih.

- Veliki programski sustav može se sastojati od desetaka ili stotina programa te od još većeg proja potprograma. Isti potprogram može biti sastavni dio raznih programa. Odnos između programa i potprograma može se tumačiti kao relacija. U njoj prvu domenu čine programi, a drugu domenu potprogrami. Relaciju je opet potrebno "čitati" u oba smjera. Dakle, ako želimo sastaviti određeni program, zanima nas od kojih sve potprograma se on sastoji. Obratno, ako ustanovimo da u nekom potprogramu postoji greška, zanima nas koje sve programe zbog toga trebamo popraviti.

  Sličan primjer relacije predstavlja odnos između industrijskih proizvoda (primjerice televizora) i njihovih sastavnih dijelova (čipova, kablova, video panela, kućišta, itd.), ili odnos između koktela i sastavnih pića.

- Rasprostranjena računalna mreža (*wide-area network*) sastoji se od paketnih sklopki (*switch*-eva) povezanih telekomunikacijskim vezama. Svaka sklopka povezana je s nekim drugim sklopkama, no ne sa svima. Strukturu povezanosti među sklopkama možemo tumačiti kao relaciju. U toj relaciji obje domene predstavlja jedan te isti skup sklopki. Relacija se koristi tako da za zadanu sklopku utvrdimo s kojim drugim sklopkama je ona povezana.

  Sličan primjer relacije predstavlja odnos između gradova povezanih cestama.

Kao što vidimo iz ovih primjera, najvažnija operacija s relacijom je njezino "čitanje" u oba smjera, dakle pronalaženje svih elemenata jedne domene koji su u relaciji sa zadanim elementom druge domene, i obratno. Također, moramo imati operacije kojima se uspostavlja ili razvrgava veza između elemenata dviju domena. Sve to zajedno daje nam ovakav apstraktni tip podataka.

Apstraktni tip podataka Relation

`domain1` ... bilo koji tip (prva domena).

`domain2` ... bilo koji tip (druga domena).

`set1` ... podatak tipa `set1` je konačan skup podataka tipa `domain1`.

`set2` ... podatak tipa `set2` je konačan skup podataka tipa `domain2`.

`Relation` ... podatak tipa `Relation` je binarna relacija čiju prvu domenu čine podaci tipa `domain1`, a drugu domenu podaci tipa `domain2`.

`ReMakeNull(&R)` ... funkcija pretvara relaciju `R` u nul-relaciju, tj. takvu u kojoj niti jedan podatak nije niti s jednim u relaciji.

`ReRelate(&R,d1,d2)` ... funkcija postavlja da je `d1` R `d2`, bez obzira da li je to već bilo postavljeno ili nije.

`ReUnrelate(&R,d1,d2)` ... funkcija postavlja da `d1` $\not R$ `d2`, bez obzira da li je to već bilo postavljeno ili nije.

`ReCompute2(R,d1,&S2)` ... za zadani `d1` funkcija skupu `S2` pridružuje kao vrijednost $\{$`d2` | `d1` R `d2`$\}$.

`ReCompute1(R,&S1,d2)` ... za zadani `d2` funkcija skupu `S1` pridružuje kao vrijednost $\{$`d1` | `d1` R `d2`$\}$.

Za implementaciju relacije koriste se iste strukture podataka kao za rječnik. U skladu s našom definicijom, relaciju R pohranjujemo kao skup uređenih parova oblika (d_1, d_2) takvih da je $d_1\ R\ d_2$. Operacije `ReMakeNull( )`, `ReRelate( )` i `ReUnrelate( )` iz apstraktnog tipa Relation implementiraju se analogno kao `DiMakeNull( )`, `DiInsert( )` odnosno `DiDelete( )` iz apstraktnog tipa Dictionary. Da bi se mogle efikasno obavljati i operacije `ReCompute1( )` i `ReCompute2( )`, potrebne su male modifikacije i nadopune strukture. Pokazat ćemo dva primjera takvih modificiranih struktura.

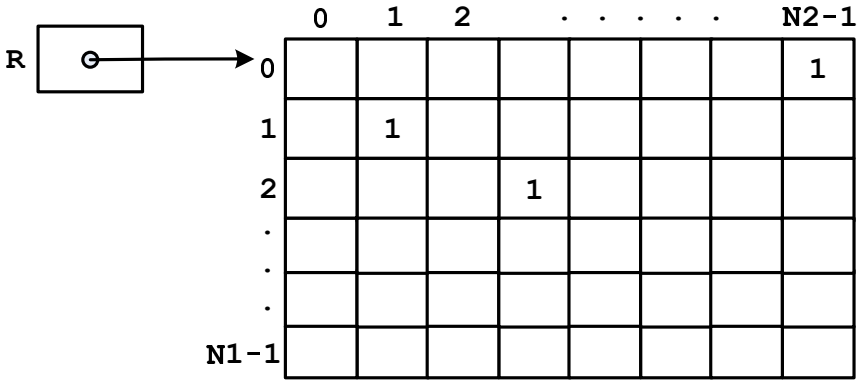
4.4.4 Implementacija relacije pomoću bit-matrice

Nastaje na osnovi implementacije skupa pomoću bit-vektora - vidi odjeljak 4.1. Jednodimenzionalno polje (vektor) presložimo u dvodimenzionalno (matricu). Uzimamo da je

`domain1` $= \{0, 1, \ldots,$ `N1`$-1\}$, a `domain2` $= \{0, 1, \ldots,$ `N2`$-1\}$, gdje su `N1` i `N2` dovoljno velike `int` konstante. Relaciju prikazujemo dvodimenzionalnim poljem bitova ili `char`-ova sljedećeg oblika:

```
# define N1 ...
# define N2 ...  /* dovoljno velike konstante */
typedef char[N2] *Relation; /* početna adresa char polja veličine N1xN2 */
```

U tom polju (i, j)-ti bit je 1 (odnosno 0) ako i samo ako je i-ti podatak u relaciji s j-tim podatkom. Operacije `ReCompute2(R,i,&S2)` odnosno `ReCompute1(R,&S1,j)` svode se na čitanje `i`-tog retka odnosno `j`-tog stupca polja. Ideja je ilustrirana slikom 4.17.



Slika 4.17: Implementacija relacije pomoću bit-matrice - korištena struktura podataka.

4.4.5 Implementacija relacije pomoću multiliste

Podsjeća na implementaciju skupa pomoću vezane liste - vidi odjeljak 4.1. Umjesto jedne vezane liste sa svim uređenim parovima (d_1, d_2), imamo mnogo malih listi koje odgovaraju rezultatima operacija `ReCompute2( )` i `ReCompute1( )`. Jedan uređeni par (d_1, d_2) prikazan je zapisom tipa:

```
typedef struct celltag {
  domain1 element1;
  domain2 element2;
  struct celltag *next1;
  struct celltag *next2;
} celltype;
```

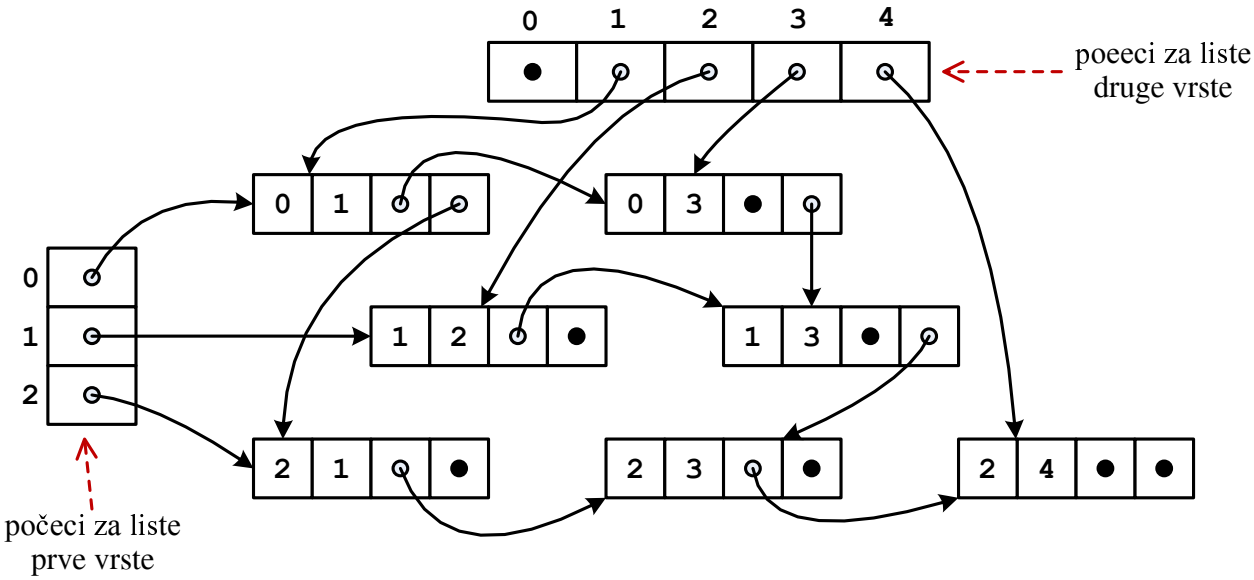
Jedan zapis istovremeno je uključen u:

- vezanu listu prve vrste, koja povezuje sve zapise s istim d_1;
- vezanu listu druge vrste, koja povezuje sve zapise s istim d_2.

Za povezivanje listi prve odnosno druge vrste služi pointer `next1` odnosno `next2`. Liste mogu biti sortirane i nesortirane.

Pointer na početak liste prve (odnosno druge) vrste zadaje se preslikavanjem čija domena je `domain1` (odnosno `domain2`) a kodomena pointeri na `celltype`. Ova dva preslikavanja mogu se implementirati na razne načine, kao što je opisano u odjeljcima 4.1 i 4.2. Operacija `ReCompute2( )` (odnosno `ReCompute1( )`) svodi se na primjenu preslikavanja, tj. pronalaženje pointera za zadani d_1 (odnosno d_2) te na prolazak odgovarajućom vezanom listom prve (odnosno druge) vrste.

Na slici 4.18 prikazana je relacija $R = \{(0,1),(0,3),(1,2),(1,3),(2,1),(2,3),(2,4)\}$. Dva preslikavanja prikazana su kao polja pointera.



Slika 4.18: Implementacija relacije pomoću multiliste - korištena struktura podataka.

Da bismo u strukturi sa Slike 4.18 npr. ustanovili koji sve elementi iz prve domene su u relaciji s elementom 3 iz druge domene, čitamo najprije polje početaka lista druge vrste te na indeksu 3 tog polja pronalazimo adresu početka odgovarajuće vezane liste druge vrste. Zatim prolazimo tom listom prateći pointere `next2`. U svakoj klijetki kroz koju prođemo čitamo vrijednost podatka `element1`. Dakle, traženi elementi iz prve domene su 0, 1 i 2.

Poglavlje 5

ALGORITMI ZA SORTIRANJE

Ovo poglavlje može se shvatiti kao studijski primjer gdje se na svrsishodan način primjenjuju znanja o strukturama podataka i o analizi algoritama. U poglavlju se bavimo problemom sortiranja podataka. Pokazujemo da se jedan te isti problem sortiranja može riješiti na razne načine, dakle različitim algoritmima. Ti algoritmi su zanimljivi jer se razlikuju po svojim osobinama, na primjer imaju različita vremena izvršavanja. Osim toga, neki od njih na zanimljiv način koriste rekurziju (dakle stog) ili binarna stabla.

Poglavlje je podijeljeno u četiri odjeljka. Svaki odjeljak obrađuje po dva algoritma sortiranja, zapisuje ih u jeziku C, pokazuje primjere njihovog rada, te analizira njihovo vrijeme izvršavanja.

Općenito, **problem (uzlaznog) sortiranja** glasi ovako. Zadana je lista duljine n, označimo je s $(a_1, a_2, \ldots, a_n)$. Vrijednosti elemenata liste mogu se uspoređivati totalnim uređajem $\leq$. Te vrijednosti treba permutirati tako da nakon permutiranja vrijedi $a_1 \leq a_2 \leq \ldots \leq a_n$. **Algoritam za sortiranje** je bilo koji algoritam koji rješava problem sortiranja.

Zbog jednostavnosti, u svim odjeljcima pretpostavljat ćemo da se naša lista sastoji od cijelih brojeva, te da je implementirana poljem duljine barem n. Dakle, umjesto o sortiranju liste $(a_1, a_2, \ldots, a_n)$ zapravo ćemo govoriti o sortiranju polja `a[ ]` koje se sastoji od elemenata `a[0]`, `a[1]`, ..., `a[`n`-1]` tipa `int`. Ovakva pojednostavnjenja ne predstavljaju bitno ograničenje općenitosti. Naime, mnogi od algoritama koje ćemo izložiti mogu se prepraviti tako da umjesto s poljem rade s nekom drugom implementacijom liste, na primjer s vezanom listom. Također, svi algoritmi mogu se umjesto na cijele brojeve primijeniti na podatke nekog drugog tipa, pod uvjetom da za te podatke imamo odgovarajuću operaciju uspoređivanja $\leq$.

Osim toga, u svim odjeljcima bavit ćemo se gotovo isključivo uzlaznim sortiranjem, dakle dovođenjem podataka u poredak od najmanjeg prema najvećem. To opet nije nikakvo ograničenje. Naime, svaki od algoritama za uzlazno sortiranje može se na trivijalan način pretvoriti u silazno sortiranje: dovoljno je uređaj $\leq$ zamijeniti s $\geq$.

5.1 Sortiranje zamjenom elemenata

U ovom odjeljku bavimo se najjednostavnijim algoritmima za sortiranje koji se svode na zamjene elemenata. Točnije, obradit ćemo dva takva jednostavna postupka: **sortiranje izborom najmanjeg elementa** (*selection sort*), odnosno **sortiranje zamjenom sus-**

jednih elemenata (*bubble sort*). Oba algoritma sa zamjenom elemenata imaju slično (kvadratično) vrijeme izvršavanja i sporiji su od većine onih koji će biti opisani u idućim odjeljcima.

5.1.1 Sortiranje izborom najmanjeg elementa

Algoritam radi na sljedeći način. Prolazi se poljem i pronalazi se najmanji element u njemu. Zatim se najmanji element zamijeni s početnim. Dalje se promatra ostatak polja (bez početnog elementa) i ponavlja isti postupak. Da bi se cijelo polje sortiralo potrebno je $n - 1$ prolazaka.

Početno polje:	17	31	3	43	11	24	8
Nakon 1. zamjene:	3	31	17	43	11	24	8
Nakon 2. zamjene:	3	8	17	43	11	24	31
Nakon 3. zamjene:	3	8	11	43	17	24	31
Nakon 4. zamjene:	3	8	11	17	43	24	31
Nakon 5. zamjene:	3	8	11	17	24	43	31
Nakon 6. zamjene:	3	8	11	17	24	31	43

Slika 5.1: Primjer sortiranja algoritmom izbora najmanjeg elementa.

Primjer sortiranja izborom najmanjeg elemenata vidi se na slici 5.1. U svakom prolasku pronađeni najmanji element označen je sjenčanjem. Promatrani dio polja nalazi se s desne strane okomite crte. Sedmi prolazak nije potreban jer je nakon šestog prolaska preostali dio polja duljine 1.

Algoritam sortiranja izborom najmanjeg elementa može se u jeziku C realizirati sljedećom funkcijom `SelectionSort()`. Ta funkcija prima kao argumente cjelobrojno polje `a[ ]` i njegovu duljinu `n`. Funkcija mijenja polje `a[ ]` tako da ono postane sortirano. Pomoćna funkcija `swap()` služi za zamjenu vrijednosti dviju cjelobrojnih varijabli sa zadanim adresama.

```
void SelectionSort (int a[], int n) {
  int i, j, min;
  for (i = 0; i < n; i++) {
    min = i;
    for (j = i+1; j < n; j++)
      if (a[j] < a[min]) min = j;
    swap(&a[i], &a[min]);
  }
}
```

```
void swap (int *x, int *y) {
  int aux;
  aux = *x;
  *x = *y;
  *y = aux;
}
```

Analiza vremena izvršavanja algoritma sortiranja izborom najmanjeg elemenata izgleda ovako.

- U prvom prolasku imamo $n-1$ usporedbi, a u svakom idućem prolasku broj usporedbi se smanji za 1. Dakle ukupni broj usporedbi iznosi:

$$(n-1)+(n-2)+(n-3)+\cdots+2+1 = n(n-1)/2.$$

- U svakom prolasku postoji i zamjena, pa imamo još $3(n-1)$ operacija pridruživanja.
- Dakle ukupni broj operacija je:

$$n(n-1)/2 + 3(n-1) = \mathcal{O}(n^2).$$

- Algoritam uvijek obavlja istu količinu posla bez obzira na početno stanje polja.

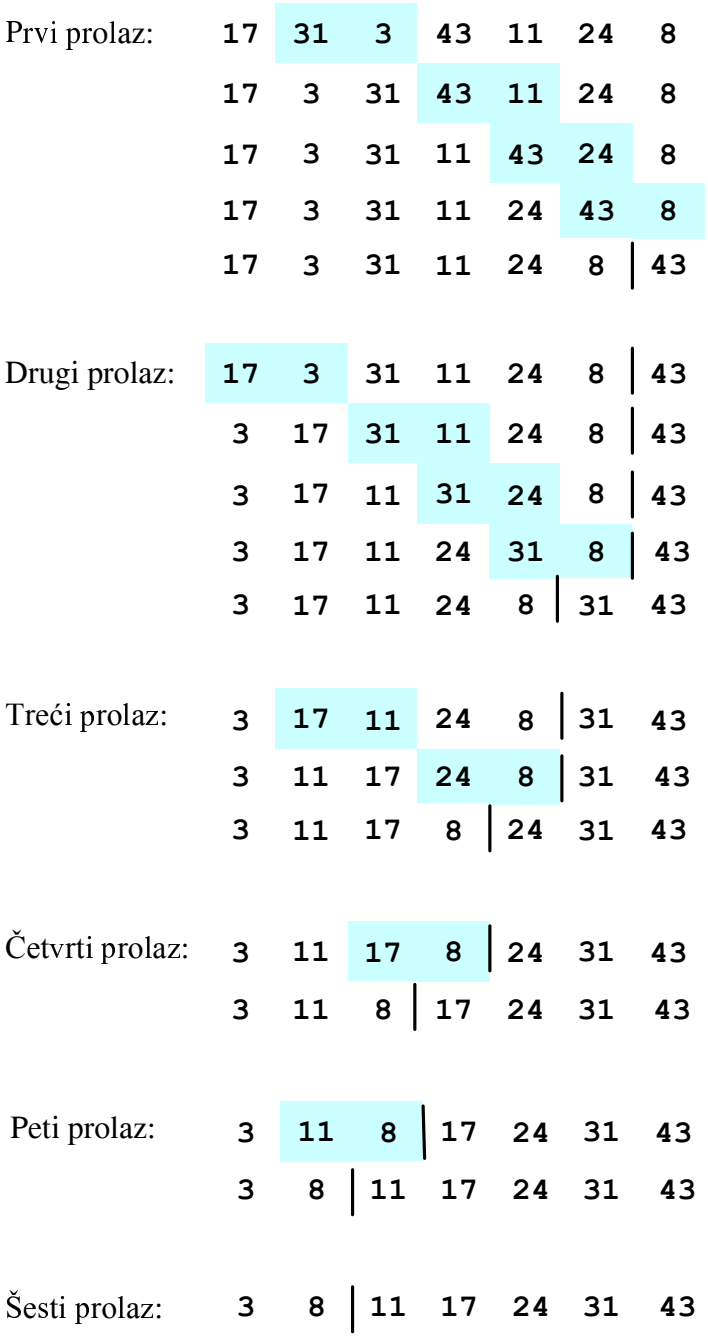
5.1.2 Sortiranje zamjenom susjednih elemenata

Opis algoritma ide ovako. Prolazimo poljem od početka prema kraju i uspoređujemo susjedne elemente. Ako je neki element veći od sljedećeg elementa, zamijenimo im vrijednosti. Kad na taj način dođemo do kraja polja, najveća vrijednost doći će na posljednje mjesto. Nakon toga ponavljamo postupak na skraćenom polju (bez zadnjeg elementa). Algoritam se smije zaustaviti čim on u nekom prolazu ustanovi da nema parova elemenata koje bi trebalo zamijeniti.

Primjer sortiranja zamjenom susjednih elemenata vidi se na slici 5.2. U svakom koraku, dva susjedna elementa koji trebaju zamijeniti vrijednosti označeni su sjenčanjem. Promatrani dio polja nalazi se s lijeve strane okomite crte. Šesti prolaz služi zato da ustanovi da više nema parova susjednih elemenata koje bi trebalo zamijeniti.

Sljedeća funkcija `BubbleSort()` implementira algoritam sortiranja zamjenom susjednih elemenata u jeziku C. Funkcija opet prima kao argumente cjelobrojno polje `a[ ]` i njegovu duljinu `n`, te mijenja polje `a[ ]` tako da ono postane sortirano. Koristi se ista pomoćna funkcija `swap( )` kao u prethodnom algoritmu.

```
void BubbleSort (int a[], int n) {
  int i, j;
  for (i = 0; i < n-1; i++)
  for (j = 0; j < n-1-i; j++)
    if (a[j+1] < a[j])
      swap(&a[j], &a[j+1]);
}
```



Slika 5.2: Primjer sortiranja algoritmom zamjene susjednih elemenata.

Implementacija se može poboljšati tako da se postupak zaustavi čim se ustanovi da u nekom prolazu nije bilo ni jedne zamjene susjednih elemenata. Poboljšana verzija funkcije `BubbleSort()` izgleda ovako.

```
void BubbleSort1 (int a[], int n) {
  int i, j, chg;
  for (i = 0, chg = 1; chg; i++) {
    chg = 0;
    for (j = 0; j < n-1-i; j++)
      if (a[j+1] < a[j]) {
        swap(&a[j], &a[j+1]);
        chg = 1;
      }
  }
}
```

U nastavku slijedi analiza vremenske složenosti opisanog algoritma sortiranja zamjenom susjednih elemenata.

- U prvom prolazu imamo u najgorem slučaju $n-1$ usporedbi i $n-1$ zamjena elemenata.
- U drugom prolazu imamo u najgorem slučaju $n-2$ usporedbi i $n-2$ zamjena elemenata.
- I tako dalje ...
- U $(n-1)$-vom prolazu imamo u najgorem slučaju 1 usporedbu i 1 zamjenu elemenata.
- Dakle ukupan broj operacija u najgorem slučaju iznosi:

$$4(n-1) + 4(n-2) + \cdots + 4 \cdot 1 = 4n(n-1)/2 = 2n(n-1).$$

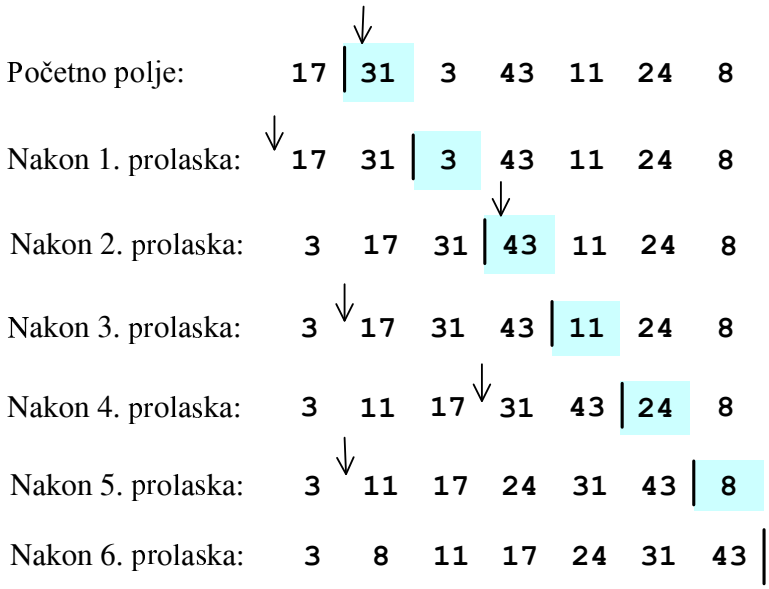
- Ocijenjeni broj operacija zaista se dobiva kad su elementi početnog polja sortirani silazno.
- S druge strane, ako je početno polje već sortirano uzlazno, obavit će se samo jedan prolaz s $n-1$ uspoređivanja i 0 zamjena.
- Za bilo koje drugo uređenje početnog polja, broj operacija je između $(n-1)$ i $2n(n-1)$. Dakle, vrijeme izvođenja u najgorem slučaju je $\mathcal{O}(n^2)$.

5.2 Sortiranje umetanjem

U ovom odjeljku razmatramo algoritme za sortiranje koji se svode na umetanje novog elementa u već sortirani niz elemenata. Najprije promatramo **jednostavnu verziju** takvog algoritma (*insertion sort*) gdje se osnovni korak umetanja iterira dovoljan broj puta. Dalje promatramo **složeniju no bržu verziju** algoritma (*Shell sort*) koja se dobiva višestrukom primjenom jednostavne verzije.

5.2.1 Jednostavno sortiranje umetanjem

Za vrijeme rada algoritma, početni komad polja već je sortiran, a ostatak polja nije sortiran. U jednom prolasku algoritam uzima prvi element iz nesortiranog dijela te ga umeće na "pravo mjesto" (u smislu sortiranog redoslijeda) u sortirani dio, pri čemu dolazi do pomicanja nekih elemenata za jedno mjesto. Dakle, jednim prolaskom duljina početnog sortiranog dijela poveća se za 1, a duljina nesortiranog dijela smanji se za 1.



Slika 5.3: Primjer jednostavnog sortiranja umetanjem.

Primjer rada jednostavnog algoritma za sortiranje umetanjem vidi se na slici 5.3. U svakom prolasku, element iz nesortiranog dijela polja koji se umeće u sortirani dio označen je sjenčanjem. Mjesto umetanja označeno je strelicom. Sortirani dio polja nalazi se s lijeve strane okomite crte, a nesortirani s desne strane.

Implementacija jednostavnog sortiranja umetanjem u jeziku C svodi se na sljedeću funkciju `InsertionSort()`. Funkcija opet kao argument prima cjelobrojno polje `a[ ]` koje treba sortirati te njegovu duljinu `n`.

```
void InsertionSort (int a[], int n) {
  int i, j;
  int aux;
  for (i = 1; i < n; i++) {
    aux = a[i];
    for (j = i; j >= 1 && a[j-1] > aux; j--)
      a[j] = a[j-1];
    a[j] = aux;
  }
}
```

Slijedi analiza vremenske složenosti opisanog jednostavnog algoritma za sortiranje umetanjem.

- U k-tom prolasku natraške prolazimo sortiranim dijelom polja duljine k. Zatečene elemente pomičemo za jedno mjesto dalje dok god su oni veći od elementa kojeg želimo umetnuti na "pravo mjesto". To u najgorem slučaju daje k usporedbi i otprilike isto toliko pridruživanja.

- Dakle ukupni broj operacija u najgorem slučaju je otprilike

$$2 \cdot 1 + 2 \cdot 2 + \cdots + 2(n-1) = n(n-1).$$

- Red veličine za vrijeme izvođenja je opet $\mathcal{O}(n^2)$. Unatoč takvoj asimptotskoj ocjeni, ovaj algoritam se u praksi ipak pokazuje bržim od prije opisanih algoritama sa zamjenom elemenata.

5.2.2 Višestruko sortiranje umetanjem

Za zadani k promatra se k različitih potpolja sastavljenih od elemenata originalnog polja međusobno udaljenih za točno k mjesta. Preciznije, promatraju se potpolja:

P_0 : `a[0], a[`k`], a[`$2k$`], ...`
P_1 : `a[1], a[`$k+1$`], a[`$2k+1$`], ...`
$\vdots$
P_{k-1} : `a[`$k-1$`], a[`$2k-1$`], a[`$3k-1$`], ...`

Proces k-struke primjene jednostavnog sortiranja umetanjem (k *insertion sort*-ova) u svrhu zasebnog sortiranja svakog od ovih k potpolja zove se k–subsort. Algoritam višestrukog sortiranja umetanjem (*Shell sort*) radi tako da za definirani padajući niz $k_1 > k_2 > \cdots > k_m = 1$ obavi k_1–subsort, zatim k_2–subsort, ..., na kraju k_m–subsort. Promijetimo da zadnji k_m–subsort (ustvari 1–subsort) izvodi obični *insertion sort* na cijelom originalnom polju. Zato korektnost algoritma nije upitna. Prethodni k–subsortovi služe zato da donekle urede cijelo polje tako da idući k–subsortovi imaju sve manje posla.

Početno polje:	**17**	**31**	**3**	**43**	**11**	**24**	**8**
Ulaz za 4-subsort	**17**	**31**	**3**	**43**	**11**	**24**	**8**
Izlaz za 4-subsort	**11**	**24**	**3**	**43**	**17**	**31**	**8**
Ulaz za 2-subsort	**11**	**24**	**3**	**43**	**17**	**31**	**8**
Izlaz za 2-subsort	**3**	**24**	**8**	**31**	**11**	**43**	**17**
Ulaz za 1-subsort	**3**	**24**	**8**	**31**	**11**	**43**	**17**
Izlaz za 1-subsort	**3**	**8**	**11**	**17**	**24**	**31**	**43**

Slika 5.4: Primjer višestrukog sortiranja umetanjem.

Primjer rada algoritma višestrukog sortiranja umetanjem vidi se na slici 5.4. Promatramo rad algoritma s padajućim nizom k-ova: $4 > 2 > 1$. Unutar svakog retka isti način sjenčanja pozadine označava elemente koji u okviru dotičnog k–subsorta čine jedno potpolje.

Implementacija višestrukog sortiranja umetanjem (*Shell sort*) u jeziku C dobiva se nadogradnjom prije prikazane implementacije jednostavnog sortiranja umetanjem. Potrebno je dodati još jednu petlju koja odgovara odabranom nizu k-subsortova. Sljedeća funkcija `ShellSort( )` koristi niz k-ova oblika $k_1 = n/2$, $k_2 = n/4$, ..., $k_m = 1$.

```
void ShellSort (int a[], int n) {
  int i, j, step;
  int aux;
  for (step = n/2; step > 0; step /= 2) {
    for (i = step; i < n; i++) {
      aux = a[i];
      for (j = i; j >= step && a[j-step] > aux; j -= step)
        a[j] = a[j-step];
      a[j] = aux;
    }
  }
}
```

Algoritam višestrukog sortiranja umetanjem predstavlja nepoznanicu i tvrd orah za istraživače. Naime, za razliku od drugih algoritama, za taj algoritam još uvijek ne postoje jasne i cjelovite ocjene vremena izvršavanja.

- Prosječno vrijeme izvođenja je otvoreni problem.
- Ako je padajući niz $k_1 > k_2 > \cdots > k_m = 1$ oblika $2^l - 1 > 2^{l-1} - 1 > \cdots > 7 > 3 > 1$, tada se može pokazati da je vrijeme u najgorem slučaju $\mathcal{O}(n^{3/2})$, dakle bolje nego kod svih dosadašnjih algoritama.
- Postoje još neki specijalni oblici niza $k_1 > k_2 > \cdots > k_m = 1$ za koje su izvedene ocjene vremena izvršavanja, no ni jedna od tih ocjena ne pokriva sve slučajeve za niz k-ova.
- Eksperimentalna mjerenja pokazuju da je algoritam iznenađujuće brz, u svakom slučaju brži nego što pokazuju raspoložive ocjene.

5.3 Rekurzivni algoritmi za sortiranje

U ovom odjeljku razmatramo rekurzivne algoritme za sortiranje, dakle algoritme koji u svrhu sortiranja polja pozivaju sami sebe. Obradit ćemo dva takva algoritma: **sortiranje pomoću sažimanja** (*merge sort*) i **brzo sortiranje** (*quick sort*). Također, na početku odjeljka obradit ćemo pomoćni postupak **sažimanja sortiranih polja** (*merge*) koji se pojavljuje kao osnovni korak u algoritmu sortiranja sažimanjem.

Sličnost dvaju razmatranih rekurzivnih algoritama sortiranja je u tome što oba dijele zadano polje na dva manja polja, zatim rekurzivnim pozivima sortiraju ta dva manja

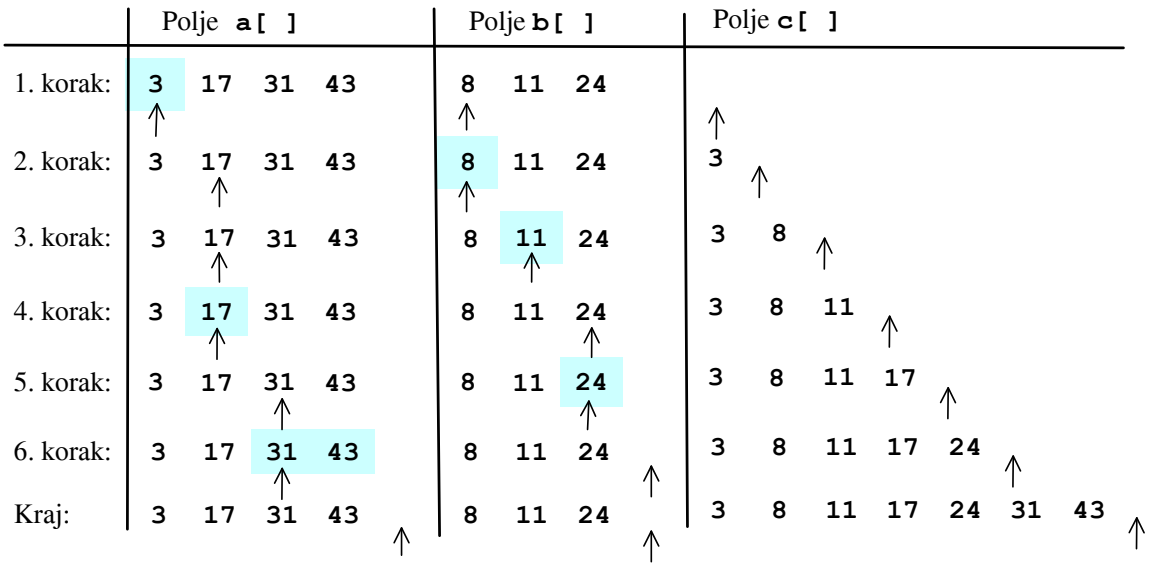
polja te ih na kraju spajaju u jedno sortirano polje. Razlika je u načinu dijeljenja velikog polja na manja polja te u načinu spajanja manjih polja u veće. Zahvaljujući tim razlikama, algoritmi imaju drukčije osobine u pogledu vremena izvršavanja.

5.3.1 Sažimanje sortiranih polja

Kao što smo najavili, najprije rješavamo jedan pomoćni problem koji je usko vezan s problemom sortiranja. Zadana su dva uzlazno sortirana polja: `a[ ]` duljine n i `b[ ]` duljine m. Sadržaj tih dvaju polja treba na što jednostavniji način sažeti zajedno i prepisati u treće polje `c[ ]` duljine $n + m$ koje je opet uzlazno sortirano.

Postupak sažimanja ide ovako. Simultano čitamo `a[ ]` i `b[ ]` od početka do kraja te istovremeno prepisujemo pročitane elemente u `c[ ]`. Dakle pamtimo kazaljku tekućeg elementa u polju `a[ ]` odnosno `b[ ]` te kazaljku na prvo slobodno mjesto u `c[ ]`. Uspoređujemo tekuće elemente u `a[ ]` i `b[ ]`.

- Ako je jedan od tih tekućih elemenata manji, tada ga prepišemo u `c[ ]` te pomaknemo odgovarajuću kazaljku u `a[ ]` odnosno `b[ ]`.
- Ako su tekući elementi jednaki, oba prepišemo u `c[ ]`, te pomaknemo obje kazaljke u `a[ ]` i u `b[ ]`.
- Kod svakog prepisivanja pomičemo kazaljku u `c[ ]`.
- Kad jedno od polja `a[ ]` i `b[ ]` pročitamo do kraja, tada ostatak drugog polja izravno prepišemo u `c[ ]`.



Slika 5.5: Primjer sažimanja sortiranih polja.

Primjer sažimanja sortiranih polja `a[ ]` i `b[ ]` u sortirano polje `c[ ]` prikazan je na slici 5.5. Strelice označavaju kazaljke tekućih elemenata u `a[ ]` i `b[ ]`, odnosno kazaljku prvog slobodnog mjesta u `c[ ]`. Sjenčanje označava elemente koji su u određenom koraku odabrani za prepisivanje.

Postupak sažimanja moguće je u C-u implementirati sljedećom funkcijom `merge( )`, koja pretpostavlja da su prije spomenuta polja `a[ ]`, `b[ ]` i `c[ ]` ustvari dijelovi nekog još većeg polja `p[ ]`. Pritom `a[ ]` odgovara elementima od `p[ ]` s indeksima između `l` i `k-1`, a `b[ ]` odgovara elementima od `p[ ]` s indeksima između `k` i `r`. Nakon sažimanja, `c[ ]` će zauzimati isti fizički prostor, dakle dio od `p[ ]` s rasponom indeksa između `l` i `r`. Kao privremeni radni prostor, funkcija koristi pomoćno polje `aux[ ]`. Polazni dijelovi polja `p[ ]` moraju biti sortirani.

```
void merge (int p[], int aux[], int l, int k, int r) {
  int i, k1, k2, count;
  k1 = k-1; /* indeks kraja prvog malog polja */
  k2 = l; /* kazaljka za pisanje u veliko polje */
  count = r-l+1; /* broj elemenata u velikom polju */

  while (l <= k1 && k <= r) { /* glavna petlja za sazimanje */
    if (p[l] <= p[k])
      aux[k2++] = p[l++];
    else
      aux[k2++] = p[k++];
  }
  while (l <= k1) /* Kopiraj ostatak prvog malog polja */
    aux[k2++] = p[l++];
  while (k <= r) /* Kopiraj ostatak drugog malog polja */
    aux[k2++] = p[k++];
  for (i = 0; i < count; i++, r--) /* Kopiraj veliko polje natrag */
    p[r] = aux[r];
}
```

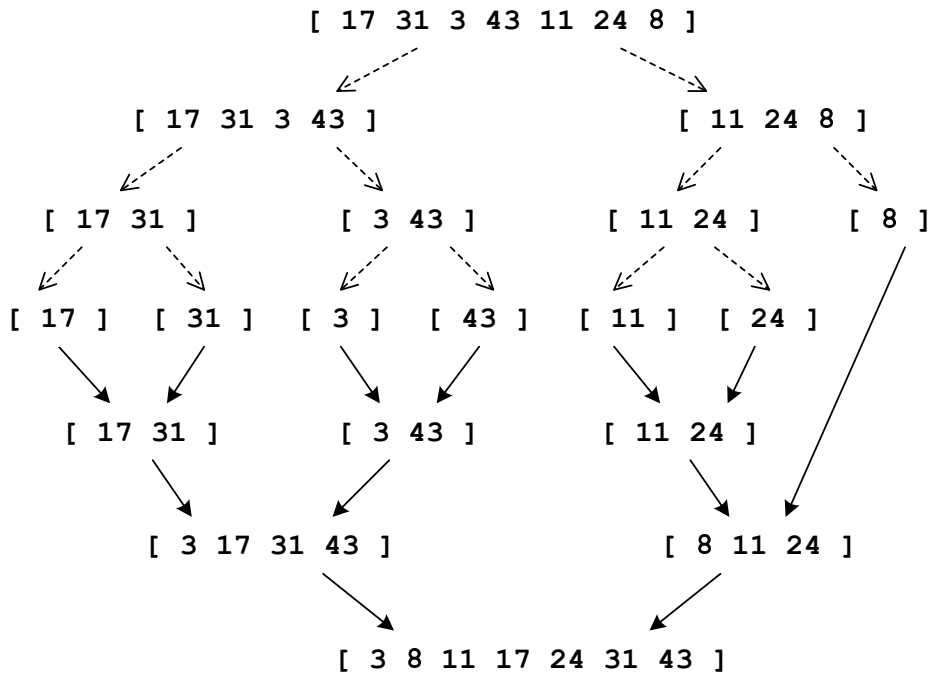
Analiza vremena izvršavanja algoritma sažimanja glasi ovako. Očito je da je to vrijeme proporcionalno sa zbrojem duljina zadanih polja odnosno s duljinom rezultirajućeg polja. Dakle red veličine za vrijeme iznosi $\mathcal{O}(n+m)$.

5.3.2 Sortiranje sažimanjem

Kao što smo najavili, riječ je o rekurzivnom algoritmu. Ako se zadano polje sastoji samo od 1 elementa, tada je već sortirano. Inače se zadano polje podijeli se na dva manja polja podjednakih duljina. Ta dva manja polja zasebno se sortiraju rekurzivnim pozivima istog algoritma. Zatim se mala sortirana polja sažimlju u jedno (sortirano) uz pomoć prethodno opisanog postupka sažimanja.

Primjer sortiranja sažimanjem vidi se na slici 5.6. Uglate zagrade označavaju pojedina polja koja nastaju u postupku sortiranja. Crtkane strelice označavaju podjele većih polja na manja. Pune strelice označavaju sažimanje manjih sortiranih polja u veća.

Algoritam sortiranja sažimanjem može se implementirati u jeziku C kao sljedeća funkcija `MergeSort( )`. Riječ je o funkciji koja ostvaruje algoritam pozivajući samu sebe i funkciju za sažimanje `merge( )`. Jedan poziv `MergeSort( )` sortira dio polja `a[ ]` koji čine elementi s indeksima u rasponu od `left` do `right`.



Slika 5.6: Primjer sortiranja sažimanjem.

Da bi sortirao polje `a[ ]` duljine `n`, glavni program mora najprije alocirati pomoćno polje `aux[ ]` iste duljine, te zatim pozvati `MergeSort( )` sa sljedećim argumentima: `MergeSort(a, aux, 0, n-1)`.

```
void MergeSort (int a[], int aux[], int left, int right) {
  int mid;
  if (left < right) {
    mid = (left + right) / 2;
    MergeSort (a, aux, left, mid);
    MergeSort (a, aux, mid+1, right);
    merge (a, aux, left, mid+1, right);
  }
}
```

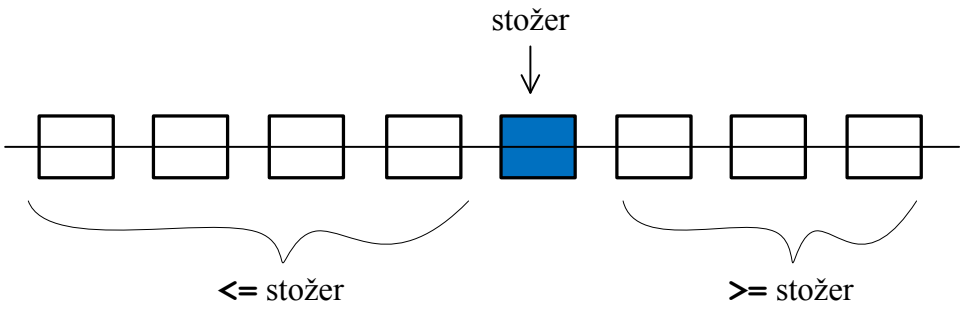
U nastavku provodimo analizu vremena izvršavanja opisanog algortima sortiranja sažimanjem.

- Svaki rekurzivni poziv algoritma ima vrijeme računanja proporcionalno duljini rezultirajućeg (sažetog) polja kojeg će on proizvesti.
- Skup svih rekurzivnih poziva koji nastaju pri rješavanju polaznog problema može se prikazati binarnim stablom, kao na slici 5.6.
- Kad gledamo sve rekurzivne pozive na istoj razini stabla, primjećujemo da oni rade s poljima čiji zbroj duljina je jednak duljini polaznog polja n. Dakle zbrojeno vrijeme računanja za sve rekurzivne pozive na istoj razini je $\mathcal{O}(n)$.

- Budući da razina ima $\log_2 n+1$, ukupna vrijeme izvršavanja algoritma u najgorem slučaju je $(\log_2 n + 1) \cdot \mathcal{O}(n) = \mathcal{O}(n \log n)$.
- Zahvaljujući ovakvoj ocjeni u najgorem slučaju, riječ o jednom od najbržih poznatih algoritama za sortiranje.
- Prednost u odnosu na druge "brze" algoritme je mogućnost sortiranja velikih polja (datoteki) pohranjenih u vanjskoj memoriji računala.
- Mana u odnosu na druge "brze" algoritme je dodatni utrošak memorije potreban zbog prepisivanja polja u sklopu sažimanja.

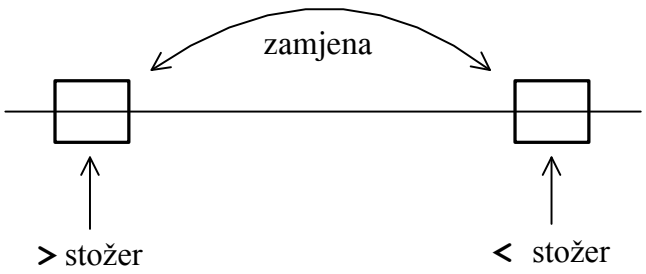
5.3.3 Brzo sortiranje

Opet je riječ o rekurzivnom algoritmu za sortiranje. Odabere se jedan element u polju, takozvani **stožer**. Svi ostali elementi razvrstaju se ispred (lijevo od) odnosno iza (desno od) stožera ovisno o tome da li su $\leq$ ili $\geq$ od stožera. Ideja je ilustrirana slikom 5.7.



Slika 5.7: Brzo sortiranje - učinak razvrstavanja.

Očigledno, nakon ovakvog razvrstavanja stožer se nalazi na svom "konačnom mjestu" u smislu traženog sortiranog poretka. Da bi se polje sortiralo do kraja, dovoljno je zasebno (neovisno) sortirati potpolje lijevo odnosno desno od stožera. Sortiranje potpolja postiže se rekurzivnim pozivima istog algoritma, ili na trivijalni način za potpolje duljine 0 ili 1.



Slika 5.8: Brzo sortiranje - detalji postupka razvrstavanja.

Samo razvrstavanje elemenata lijevo odnosno desno od stožera efikasno se obavlja u skladu sa slikom 5.8, tako da pomičemo kazaljku s lijevog odnosno desnog kraja polja, sve dok su tekući elementi $\leq$ odnosno $\geq$ od stožera. Kad se obje kazaljke zaustave, našli smo element slijeva koji je $>$ od stožera i element s desna koji je $<$ od stožera. Ta dva elementa na "krivim stranama" međusobno zamijenimo i zatim nastavimo s pomicanjem kazaljki. Razvrstavanje je završeno kad se kazaljke prekriže.

Početno polje, stožer je početni element:	**17**	**31**	**3**	**43**	**11**	**24**	**8**
Pomicanje i zaustavljanje kazaljki:	**17**	**31** $\uparrow_L$	**3**	**43**	**11**	**24**	**8** $\uparrow_D$
Zamjena elemenata na koje pokazuju kazaljke:	**17**	**8** $\uparrow_L$	**3**	**43**	**11**	**24**	**31** $\uparrow_D$
Pomicanje i zaustavljanje kazaljki:	**17**	**8**	**3**	**43** $\uparrow_L$	**11** $\uparrow_D$	**24**	**31**
Zamjena elemenata na koje pokazuju kazaljke:	**17**	**8**	**3**	**11** $\uparrow_L$	**43** $\uparrow_D$	**24**	**31**
Pomicanje i zaustavljanje kazaljki:	**17**	**8**	**3**	**11** $\uparrow_D$	**43** $\uparrow_L$	**24**	**31**
Kazaljke su se prekrižile. Ubacujemo stožer:	**[8**	**3**	**11]**	**17**	**[43**	**24**	**31]**
Dva potpolja sortiramo rekurzivnim pozivima:	**[3**	**8**	**11]**	**17**	**[24**	**31**	**43]**

Slika 5.9: Brzo sortiranje - primjer rada algoritma.

Primjer rada algoritma brzog sortiranja prikazan je na slici 5.9. Simboli $\uparrow_{\mathrm{L}}$ odnosno $\uparrow_{\mathrm{D}}$ označavaju položaj kazaljke s lijevog odnosno desnog kraja polja u raznim fazama razvrstavanja elemenata. Okomita crta pokazuje da je stožer izdvojen tijekom rasvrstavanja. Uglate zagrade zatvaraju potpolja koja se sortiraju rekurzivnim pozivima istog algoritma. Nisu prikazani detalji unutar tih rekurzivnih poziva.

Algoritam brzog sortiranja može se implementirati u jeziku C na više načina s obzirom na razne mogućnosti izbora stožera. Sljedeća rekurzivna funkcija `QuickSort( )` predstavlja najjednostavniju varijantu implementacije, gdje se u svakom rekurzivnom pozivu kao stožer bira početni element potpolja kojeg upravo treba sortirati. Opet se koristi pomoćna funkcija `swap()` za zamjenu vrijednosti dviju cjelobrojnih varijabli (vidi pododjeljak 5.1.1). Jedan poziv `QuickSort( )` sortira dio polja `a[ ]` čiji elementi imaju indekse u rasponu od `l` do `r`. Poziv koji sortira cijelo polje `a[ ]` duljine `n` izgleda ovako: `QuickSort(a, 0, n-1)`.

```
void QuickSort(int a[], int l, int r) {
  int i,j;

  if (l>=r) return;
```

```
  i = l+1; /* razvrstavanje elemenata s obzirom na stozer */
  j = r;
  while ((i <= j) && (i<=r) && (j>l)) {
    while ((a[i] <= a[l]) && (i<=r)) i++;
    while ((a[j] >= a[l]) && (j>l)) j--;
    if (i<j)
      swap(&a[i], &a[j]);
  }
  if (i>r) { /* stozer je najveci u polju */
    swap(&a[r], &a[l]);
    QuickSort(a, l, r-1);
  }
  else if (j<=l) { /* stozer je najmanji u polju */
    QuickSort(a, l+1, r);
  }
  else { /* stozer je negdje u sredini */
    swap(&a[j], &a[l]);
    QuickSort(a, l, j-1);
    QuickSort(a, j+1, r);
  }
}
```

Slijedi popis najvažnijih rezultata vezanih uz vrijeme izvršavanja algoritma brzog sortiranja.

- Može se lako provjeriti da algoritam u najgorem slučaju ima složenost $\mathcal{O}(n^2)$. Najgori slučaj nastupa kad je stožer početni element i kad je polje već sortirano.
- Postoji matematički dokaz da je prosječno vrijeme izvršavanja $\mathcal{O}(n \log n)$.
- Praktična iskustva pokazuju da je algoritam u stvarnosti izuzetno brz, najbrži od svih poznatih algoritama, dakle brži nego što se to može prepoznati iz teorijskih ocjena.
- Ponašanje algoritma u priličnoj mjeri ovisi o načinu biranja stožera. Često korištene mogućnosti za izbor stožera su: početni element (kao u našem primjeru i funkciji); odnosno medijan izabran između tri elementa koji se nalaze na početku, kraju odnosno sredini polja.

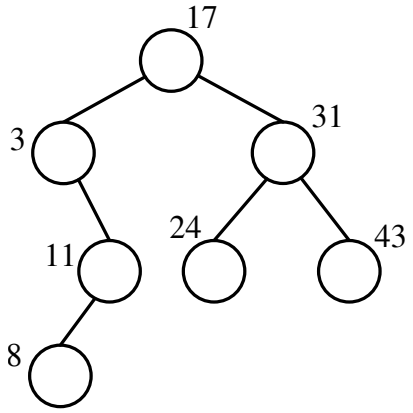
5.4 Sortiranje pomoću binarnih stabala

U ovom potpoglavlju obradit ćemo još dva algoritma za sortiranje. To su: **sortiranje obilaskom binarnog stabla traženja** (*tree sort*), odnosno **sortiranje pomoću hrpe** (*heap sort*). Dva algoritma liče jedan na drugi, naime svaki od njih sortira polje tako da elemente polja najprije redom ubaci u binarno stablo, a zatim ih izvadi iz tog stabla u sortiranom redoslijedu. Razlika je u vrsti binarnog stabla koje se koristi: u prvom slučaju to je binarno stablo traženja, a u drugom slučaju to je hrpa.

5.4.1 Sortiranje obilaskom binarnog stabla traženja

Iz definicije binarnog stabla traženja (vidi pododjeljak 4.2.5) i obilaska `Inorder( )` (vidi pododjeljak 3.1.2) očigledno je da `Inorder( )` posjećuje čvorove binarnog stabla traženja u sortiranom redoslijedu s obzirom na njihove oznake. Zaista, `Inorder( )` najprije posjećuje sve čvorove čije oznake su manje od oznake korijena, zatim korijen, te na kraju čvorove čije oznake su veće od oznake korijena – dakle korijen se u nizu posjećenih čvorova nalazi na svom "pravom mjestu" u smislu sortiranog poretka po oznakama. Isto svojstvo vrijedi zbog rekurzije i za sve ostale čvorove, pa je cijeli niz posjećenih čvorova ispravno sortiran.

Uočeno svojstvo binarnog stabla traženja i obilaska `Inorder( )` može se iskoristiti za oblikovanje algoritma za sortiranje koji se zove *tree sort*. Algoritam kreće od praznog binarnog stabla traženja. Podaci iz polja koje treba sortirati redom se ubacuju u binarno stablo kao oznake čvorova. Nakon što su svi podaci ubačeni, obilazimo binarno stablo postupkom `Indorder( )`. Upisujemo oznake iz binarnog stabla natrag u polje, i to onim redom kako ih `Indorder( )` posjećuje.



Slika 5.10: Sortiranje obilaskom binarnog stabla traženja.

Kao primjer rada algoritma za sortiranje obilaskom binarnog stabla traženja, sortirajmo polje sa sljedećim sadržajem:

$$17, 31, 3, 43, 11, 24, 8.$$

Najprije sve podatke iz polja redom ubacujemo u binarno stablo traženja. Dobivamo binarno stablo prikazano na slici 5.10. Kad podatke iz tog binarnog stabla u redoslijedu `Inorder( )` prepišemo natrag u polje, dobivamo:

$$3, 8, 11, 17, 24, 31, 43.$$

Algoritam sortiranja obilaskom binarnog stabla traženja mogao bi se implementirati kombiniranjem funkcija za rad s binarnim stablima traženja iz pododjeljka 4.2.5 te funkcija za obilazak binarnih stabala nalik onima iz pododjeljka 3.1.2. Ipak, u nastavku ćemo izložiti kompaktniju i donekle modificiranu verziju takvog programskog koda.

Sortiranje polja `a[ ]` duljine `n` obavlja se pozivom `TreeSort(a,n)`. Rad funkcije `TreeSort()` ostvaruje se pozivanjem triju pomoćnih funkcija: `insert()`, `writeInorder()`

i `destroy()`. Uzastopni pozivi `insert()` dinamički alociraju memoriju i grade binarno stablo traženja u kojem su prepisani svi podaci iz `a[ ]`. Pritom jedan poziv `insert()` ubacuje u binarno stablo jedan čvor s elementom iz `a[ ]` na način sličan kao u pododjeljku 4.2.5, s time da se sada dopuštaju i čvorovi s jednakim podacima. Funkcija `writeInorder()` obavlja inorder obilazak izgrađenog binarnog stabla te usputno prepisivanje podataka iz binarnog stabla natrag u polje u redoslijedu obilaska. Funkcija `destroy()` razgrađuje binarno stablo i dealocira memoriju. Sve funkcije se oslanjaju na definiciju tipa `celltype` koji odgovara čvoru binarnog stabla.

```
typedef struct celltag {
  int element;
  struct celltag *leftchild;
  struct celltag *rightchild;
} celltype;

void TreeSort (int a[], int n) {
  int i, next;
  celltype *tree;
  tree = NULL;
  for (i=0; i<n; i++)
    tree = insert(a[i], tree);
  next=0;
  writeInorder(tree, a, &next);
  destroy(tree);
}

celltype *insert (int x, celltype *node) {
  if (node == NULL) {
    node = (celltype*) malloc(sizeof(celltype));
    node->element = x;
    node->leftchild = node->rightchild = NULL;
  }
  else if (x < node->element)
    node->leftchild = insert(x, node->leftchild);
  else
    node->rightchild = insert(x, node->rightchild);
  return node;
}

void writeInorder (celltype *node, int a[], int *np) {
  if (node != NULL) {
    writeInorder(node->leftchild, a, np);
    a[*np] = node->element;
    *np += 1;
    writeInorder(node->rightchild, a, np);
  }
}
```

```
void destroy (celltype *node) {
  if (node != NULL) {
    destroy(node->leftchild);
    destroy(node->rightchild);
    free(node);
  }
}
```

Slijedi analiza vremena izvršavanja opisanog algoritma sortiranja zasnovanog na obilasku binarnog stabla traženja.

- Ako su podaci u polaznom polju dobro izmiješani, tada će dobiveno binarno stablo traženja biti dobro razgranato, tj njegova visina bit će otprilike $\log_2 n$, gdje je n broj podataka koje sortiramo.
- Budući da svaka operacija ubacivanja zahtijeva vrijeme proporcionalno visini binarnog stabla te budući da imamo n ubacivanja, gradnja binarnog stabla zahtijeva vrijeme $\mathcal{O}(n \log n)$.
- Obilazak `Inorder( )` očito ima linearnu složenost $\mathcal{O}(n)$.
- Kad se to sve zbroji, ispada da cijeli algoritam sortiranja ima vrijeme $\mathcal{O}(n \log n)$.
- Ipak, ta ocjena vrijedi samo za vrijeme u prosječnom slučaju.
- Najgori slučaj nastupa na primjer onda kad je niz podataka već sortiran: binarno stablo traženja tada se pretvara u "koso" binarno stablo s visinom n, pa cijeli algoritam sortiranja zahtijeva $\mathcal{O}(n^2)$ operacija.

5.4.2 Sortiranje pomoću hrpe

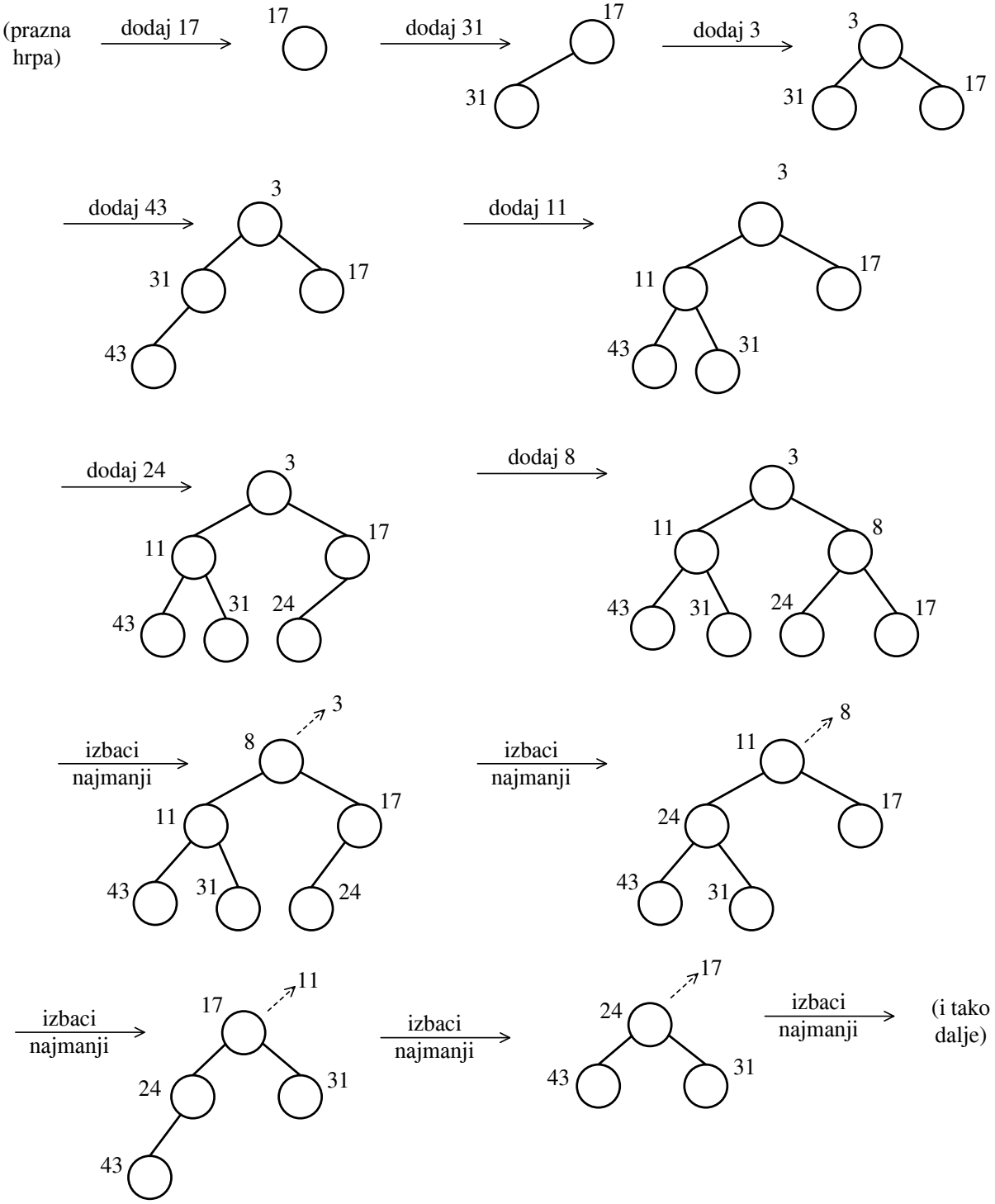
Slično kao binarno stablo traženja, i hrpa (vidi pododjeljak 4.3.4) se može primijeniti za sortiranje. Podaci iz polja koje želimo sortirati najprije se redom ubacuju u hrpu. Zatim se podaci redom skidaju s hrpe primjenom operacije izbacivanja najmanjeg elementa te se vraćaju u polje. Zahvaljujući svojstvima hrpe odnosno svojstvima operacije izbacivanja najmanjeg elementa, podaci izlaze iz hrpe u sortiranom poretku, dakle od najmanjeg prema najvećem.

Rezultirajući algoritam za sortiranje naziva se sortiranje pomoću hrpe (*heap sort*). Algoritam se obično implementira tako da se hrpa pohrani u polju, pa se u konačnici svodi na manipuliranje s podacima u polju (slično kao i ostali prikazani algoritmi za sortiranje).

Kao primjer korištenja algoritma za sortiranje pomoću hrpe promatramo sortiranje polja koje ima sljedeći sadržaj:

$$17, 31, 3, 43, 11, 24, 8.$$

Postupak je prikazan slikom 5.11. Algoritam najprije dodaje podatke iz polja u hrpu, a zatim ih skida s hrpe, pa ih tako dobiva natrag u sortiranom redoslijedu. Kad se podaci



Slika 5.11: Sortiranje pomoću hrpe.

ponovo unesu u polje onim redom kako su izašli iz hrpe, i to od početka polja prema kraju, dobiva se sljedeći sadržaj:

$$3, 8, 11, 17, 24, 31, 43.$$

Algoritam sortiranja pomoću hrpe mogao bi se implementirati kombiniranjem funkcija za rad s hrpom iz pododjeljka 4.3.4. U nastavku dajemo kompaktniju verziju sličnog programskog koda. Sortiranje polja `a[ ]` duljine `n` obavlja se pozivom funkcije `HeapSort( )`. Koriste se pomoćne funkcije `buildHeap( )` i `adjust( )` koje su navedene ovdje ispod `HeapSort( )` te `swap( )` iz pododjeljka 5.1.1.

```
void HeapSort (int a[], int n) {
  int i;
  buildHeap (a, n);
  for (i = n; i >= 2; i--) {
    swap(&a[0], &a[i-1]);
    adjust(a, 0, i-1);
  }
}

void buildHeap (int a[], int n) {
  int i;
  for (i = n/2-1; i >= 0; i--)
    adjust(a, i, n);
}

void adjust(int a[], int i, int n) {
  int j;
  int x;
  j = 2*i+1;
  x = a[i];
  while (j <= n-1 ) {
    if ((j < n-1) && (a[j] > a[j+1])) j++;
    if (x <= a[j]) break;
    a[(j-1)/2] = a[j];
    j = 2*j+1;
  }
  a[(j-1)/2] = x;
}
```

Polje `a[ ]` koje treba sortirati interpretira se kao prikaz potpunog binarnog stabla u skladu s poodjeljkom 3.2.3. Na početku to binarno stablo nije hrpa jer oznake čvorova nisu raspoređene na propisani način. No pozivom pomoćne funkcije `buildHeap( )` oznake se premještaju tako da se zadovolji svojstvo hrpe.

Sam `buildHeap( )` svodi se na niz poziva pomoćne funkcije `adjust( )`. Jedan poziv `adjust( )` odnosi se na unutrašnji čvor `i` potpunog binarnog stabla od `n` čvorova, a njime se obavlja ciklička zamjena oznaka po putu od čvora `i` prema nekom listu, onako kako je bilo opisano u pododjeljku 4.3.4.

Nakon što je binarno stablo postalo hrpa, najmanja oznaka nalazi se u korijenu. Izbacivanje najmanje oznake iz hrpe obavlja se tako da se početni element polja `a[ ]` zamijeni sa zadnjim te se duljina polja skrati za 1. Da bi skraćeno polje opet postalo hrpa, potrebno ga je podesiti dodatnim pozivom `adjust( )`. Nizom ovakvih iteracija izbacit ćemo iz hrpe sve oznake od najmanje prema najvećoj.

Opisana implementacija je spretna zato što se podaci izbačeni iz hrpe redom slažu u nekorištenom dijelu polja, dakle tamo gdje bi se na kraju i trebali naći. No primijetimo da ćemo dobiti *silazno* sortirano polje umjesto uzlaznog. Ako nam to smeta, tada u priloženom programskom kodu moramo invertirati uređajnu relaciju, dakle sve uvjete $\leq$ moramo zamijeniti s $\geq$ i obratno.

U nastavku slijedi analiza vremena izvršavanja opisanog algoritma sortiranja pomoću hrpe.

- Algoritam se svodi na n-struku primjenu operacije ubacivanja elementa u hrpu, odnosno izbacivanja najmanjeg elementa iz hrpe.
- Budući da jedno ubacivanje odnosno izbacivanje troši vrijeme reda $\mathcal{O}(\log n)$, slijedi da je vrijeme u najgorem slučaju za cijelo sortiranje reda $\mathcal{O}(n \log n)$.
- Eksperimenti pokazuju da sortiranje pomoću hrpe zaista spada među najbrže poznate algoritme za sortiranje i da uspješno sortira vrlo velika polja.

Poglavlje 6

OBLIKOVANJE ALGORITAMA

Na osnovi dugogodišnjeg iskustva, ljudi su identificirali nekoliko općenitih "tehnika" (metoda, strategija) za oblikovanje algoritama. U ovom poglavlju proučit ćemo najvažnije takve tehnike: podijeli-pa-vladaj, dinamičko programiranje, pohlepni pristup, *backtracking* i lokalno traženje. Svaka od njih bit će obrađena u zasebnom odjeljku.

Kad želimo sami oblikovati algoritam za rješavanje nekog novog problema, tada je preporučljivo početi tako da se upitamo: "Kakvo rješenje bi dala metoda podijeli-pa-vladaj, što bi nam dao pohlepni pristup, itd.". Nema garancije da će neka od ovih metoda zaista dati korektno rješenje našeg problema - to tek treba provjeriti. Također, rješenje ne mora uvijek biti egzaktno, već može biti i približno. Na kraju, dobiveni algoritam ne mora biti efikasan. Ipak, spomenute metode su se pokazale uspješne u mnogim situacijama, pa je to dobar razlog da ih bar pokušamo primijeniti.

6.1 Metoda podijeli-pa-vladaj

Podijeli-pa-vladaj vjerojatno je najprimjenjivija strategija za oblikovanje algoritama. Ona je izraz prirodne ljudske težnje izbjegavanja kompliciranih problema te njihovog svođenja na jednostavnije. U ovom odjeljku najprije ćemo općenito opisati tu metodu, a zatim ćemo obraditi tri algoritma koji su oblikovani u skladu s njom.

6.1.1 Općenito o podijeli-pa-vladaj

Metoda **podijeli-pa-vladaj** (*divide-and-conquer*) sastoji u tome da zadani primjerak problema razbijemo u nekoliko manjih (istovrsnih) primjeraka, i to tako da se rješenje polaznog primjerka može konstruirati iz rješenja manjih primjeraka. Dobiveni algoritam je rekurzivan - naime svaki od manjih primjeraka dalje se rješava na isti način, tj. razbije se u još manje primjerke, itd.

Da bi algoritam tipa podijeli-pa-vladaj zaista bio izvediv, potrebno je da su zadovoljeni sljedeći uvjeti:

- Mora postojati relativno lak i brz način da se rješenje većeg primjerka problema konstruira iz rješenja manjih primjeraka.
- Mora postojati način da se primjerci dovoljno male veličine riješe izravno (bez daljnjeg razbijanja).

Također, zbog efikasnosti algoritma poželjno je da manji primjerci koji nastaju razbijanjem većeg primjerka budu dobro "balansirani" (sličnih veličina i težina) te da se isti manji primjerci što manje ponavljaju u različitim dijelovima rekurzije.

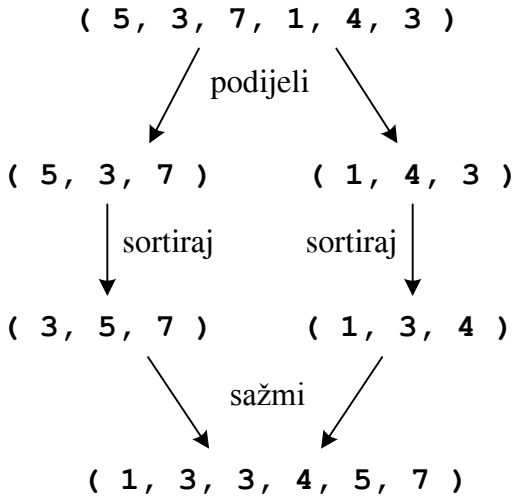
6.1.2 Opet sortiranje sažimanjem

Algoritam sortiranja liste sažimanjem (*merge sort*) koji smo već upoznali u pododjeljku 5.3.2, može se ovako tumačiti:

- Što je lista dulja, to ju je teže sortirati. Zato polaznu listu treba razbiti na dvije manje i svaku od njih treba sortirati zasebno.
- Nakon takvih zasebnih sortiranja, tražena velika sortirana lista dobiva se relativno jednostavnim postupkom sažimanja manjih sortiranih listi.

Slika 6.1 ilustrira ovu ideju na konkretnom primjeru liste.

U skladu s ovakvim tumačenjem, zaključujemo da je sortiranje sažimanjem zapravo algoritam koji slijedi strategiju podijeli-pa-vladaj. Analiza vremena izvršavanja iz pododjeljka 5.3.2 pokazala je da je ujedno riječ o vrlo brzom algoritmu. To znači da imamo primjer koji dokazuje da primjena podijeli-pa-vladaj može biti vrlo uspješna.



Slika 6.1: Sortiranje sažimanjem kao primjer za podijeli-pa-vladaj.

Osim sortiranja sažimanjem, postoje i još neki algoritmi sortiranja za koje se može smatrati da djeluju u skladu sa strategijom podijeli-pa-vladaj. To svakako vrijedi za brzo sortiranje (*quick sort*). Isto tako, sortiranje umetanjem (*insertion sort*) može se interpretirati kao algoritam tipa podijeli-pa-vladaj, gdje se polazna lista duljine n najprije dijeli na dvije nejednake liste duljina $n-1$ odnosno 1, a zatim se te dvije liste nakon sortiranja spajaju tako da se jedini element iz manje liste ubaci na pravo mjesto u veću listu.

Rekli smo da je za efikasnost algoritama tipa podijeli-pa-vladaj važno da manji primjerci problema budu dobro izbalansirani (sličnih veličina). To vrijedi i za algoritme

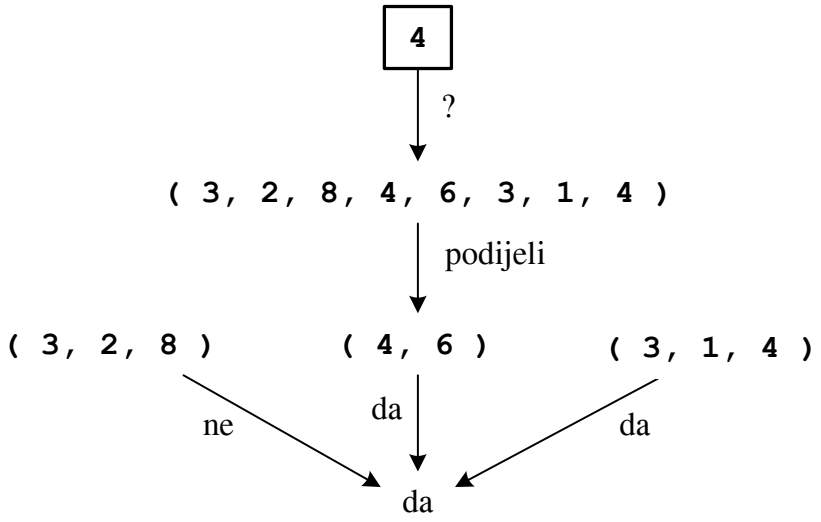
sortiranja. Zaista, sortiranje umetanjem loše balansira primjerke pa je zato njegovo vrijeme sortiranja liste duljine n u najgorem slučaju $\mathcal{O}(n^2)$. S druge strane, sortiranje sažimanjem polaznu listu dijeli na dvije manje liste podjednakih duljina, pa zato postiže bolje vrijeme u najgorem slučaju $\mathcal{O}(n \log n)$. Slično vrijedi i za brzo sortiranje: njegovo vrijeme bitno ovisi o tome da li odabrani stožer pravilno raspolavlja listu.

6.1.3 Traženje elementa u listi

Želimo utvrditi da li u zadanoj listi postoji element sa zadanom vrijednošću. Metoda podijeli-pa-vladaj sugerira sljedeće rješenje:

- Što je lista dulja, to ju je teže pretraživati; zato treba polaznu listu razbiti na npr. tri manje. U svakoj maloj listi treba zasebno tražiti zadanu vrijednost.
- Tražena vrijednost pojavljuje se u velikoj listi ako i samo ako se pojavljuje u bar jednoj maloj listi. Znači, konačni odgovor dobije se tako da se parcijalni odgovori kombiniraju logičkom operacijom "ili".

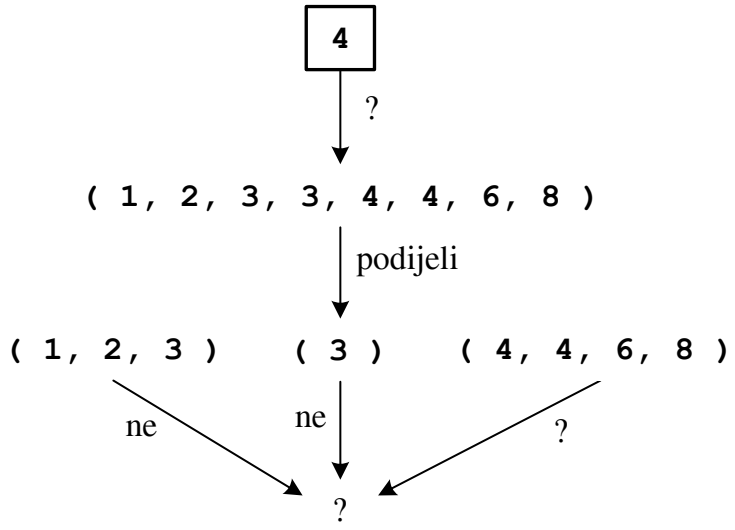
Ovakav način rjšavanja ilustriran je konkretnim primjerom na slici 6.2.



Slika 6.2: Traženje elementa u listi metodom podijeli-pa-vladaj.

Lako se uvjeriti da opisani algoritam za traženje elementa u listi nije osobito efikasan, to jest on u najgorem slučaju troši isto vrijeme kao sekvencijalni pregled cijele liste od početka do kraja. No ako je polazna lista sortirana, tada se algoritam može bitno pojednostaviti na način koji je ilustriran slikom 6.3. Naime:

- Podjelu na manje liste možemo provesti tako da srednja mala lista ima duljinu 1, a ostale dvije su podjednake duljine.
- Jedna operacija uspoređivanja (zadana vrijednost s elementom u sredini) tada nam omogućuje da se vrijednost odmah pronađe ili da se dvije liste eliminiraju iz daljnjeg razmatranja.



Slika 6.3: Traženje elementa u sortiranoj listi metodom podijeli-pa-vladaj.

Lako je uočiti da je upravo opisani algoritam traženja elementa u sortiranoj listi zapravo algoritam binarnog traženja kojeg smo već upoznali u odjeljku 4.2. Dakle binarno traženje može se shvatiti kao primjena metode podijeli-pa-vladaj. Opet je riječ o uspješnoj primjeni te metode koja je rezultirala vrlo brzim algoritmom.

6.1.4 Množenje dugačkih cijelih brojeva

Promatramo problem množenja dvaju n-bitnih cijelih brojeva X i Y. Klasični algoritam iz osnovne škole zahtijeva računanje n parcijalnih produkata veličine n; dakle njegova složenost je $\mathcal{O}(n^2)$ ukoliko operacije s jednim bitom smatramo osnovnima. Pristup podijeli-pa-vladaj sugerira ovakav algoritam:

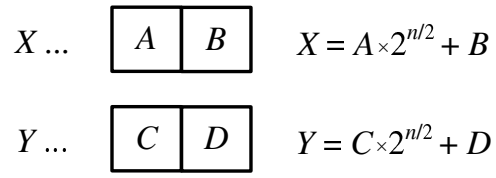
- Svaki od brojeva X odnosno Y treba podijeliti na dva dijela duljine $n/2$ bitova (zbog jednostavnosti uzimamo da je n potencija od 2).
- Neka su A i B dijelovi od X, a C i D dijelovi od Y. Produkt od X i Y tada se može ovako izraziti:

$$XY = AC2^n + (AD + BC)2^{\frac{n}{2}} + BD.$$

Podjela velikih brojeva na manje ilustrirana je slikom 6.4.

Znači, ideja je da jedno množenje "velikih" n-bitnih brojeva reduciramo na nekoliko množenja "manjih" $n/2$-bitnih brojeva. Prema gornjoj formuli, treba napraviti: 4 množenja $n/2$-bitnih brojeva (AC, AD, BC i BD), tri zbrajanja brojeva duljine najviše $2n$ bitova (tri znaka +), te dva "šifta" (množenje s 2^n odnosno $2^{\frac{n}{2}}$). Budući da i zbrajanja i šiftovi zahtjevaju $\mathcal{O}(n)$ koraka, možemo pisati sljedeću rekurziju za ukupan broj koraka $T(n)$ koje implicira gornja formula:

$$\begin{aligned} T(1) &= 1, \\ T(n) &= 4T(n/2) + cn \quad \text{(ovdje je } c \text{ neka konstanta).} \end{aligned}$$



Slika 6.4: Prikaz brojeva duljine n bitova pomoću brojeva duljine $n/2$ bitova.

Lako se vidi da je rješenje ove rekurzije $T(n) = \mathcal{O}(n^2)$. Znači, naš algoritam dobiven metodom podijeli-pa-vladaj nije ništa bolji od klasičnog.

Ipak, poboljšanje se može dobiti ukoliko gornju formulu preinačimo tako da glasi:

$$XY = AC2^n + [(A - B)(D - C) + AC + BD]\, 2^{\frac{n}{2}} + BD.$$

Broj množenja $n/2$-bitnih brojeva se sada smanjio na tri. Doduše, povećao se broj zbrajanja odnosno oduzimanja, no to nije važno jer je riječ o operacijama s manjim vremenom izvršavanja. Rekurzija za $T(n)$ sada glasi:

$$\begin{aligned} T(1) &= 1, \\ T(n) &= 3T(n/2) + cn \quad \text{(ovdje je } c \text{ konstanta veća nego prije).} \end{aligned}$$

Rješenje rekurzije je $T(n) = \mathcal{O}(n^{\log_2 3}) = \mathcal{O}(n^{1.59})$. Znači, nova verzija algoritma podijeli-pa-vladaj brža je od klasičnog algoritma. Doduše, poboljšanje se zaista osjeća tek kod izuzetno velikih cijelih brojeva ($n \approx 500$).

Slijedi skica algoritma u jeziku C. Ta skica je zbog ograničenja standardnog tipa `int` zapravo ispravna samo za male n-ove. Prava implementacija algoritma zahtijevala bi da razvijemo vlastiti način pohranjivanja cjelih brojeva: npr svaku cjelobrojnu varijablu mogli bi zamijeniti dugačkim poljem bitova. Također, morali bi sami implementirati aritmetičke operacije s tako prikazanim cijelim brojevima.

```
int Mult (int X, int Y, int n) {
  /* X i Y su cijeli brojevi s predznakom, duljine n bitova.  */
  /* Pretpostavljamo da je n potencija od 2.  Funkcija vraća X*Y. */

  int s; /* predznak od X*Y */
  int m1, m2, m3; /* tri "mala" produkta */
  int A, B, C, D; /* lijeve i desne polovice od X i Y */
  s = (X>=0?1:-1) * (Y>=0?1:-1);
  X = abs(X);
  Y = abs(Y); /* radimo kao da su X i Y pozitivni */

  if (n == 1)
    if ( (X==1) && (Y==1) )
      return s;
    else
      return 0;
```

```
  else {
    A = lijevih n/2 bitova od X;
    B = desnih n/2 bitova od X;
    C = lijevih n/2 bitova od Y;
    D = desnih n/2 bitova od Y;
    m1 = Mult(A, C, n/2);
    m2 = Mult(A-B, D-C, n/2);
    m3 = Mult(B, D, n/2);
    return (s * (m1<<n +(m1+m2+m3)<<(n/2) + m3) );
    /* ovdje je << operator za šift bitova ulijevo */
  }
}
```

6.2 Dinamičko programiranje

Metoda podijeli-pa-vladaj koji put rezultira neefikasnim algoritmom. Naime, dešava se da broj manjih primjeraka problema koje treba riješiti raste eksponencijalno s veličinom zadanog primjerka. Pritom ukupan broj različitih manjih primjeraka možda i nije tako velik, no jedan te isti primjerak pojavljuje se na mnogo mjesta u rekurziji (pa ga uvijek iznova rješavamo). U opisanoj situaciji preporuča se **dinamičko programiranje** (*dinamic programming*).

Ovaj odjeljak započinje s nekoliko općenitih napomena o dinamičkom programiranju. Zatim se obrađuju dva algoritma oblikovana u skladu s tom metodom. U oba slučaja dobija se bolji algoritam od onog koji bi se dobio metodom podijeli-pa-vladaj.

6.2.1 Ideja dinamičkog programiranja

Slično kao podijeli-pa-vladaj, metoda dinamičkog programiranja konstruira rješenje velikog primjerka problema služeći se rješenjima manjih primjeraka. No način i redoslijed rješavanja je drukčiji. Naime, metoda zahtijeva da se svi manji primjerci redom riješe, te da se rješenja spreme u odgovarajuću tabelu. Kad god nam treba neko rješenje, tada ga ne računamo ponovo već ga samo pročitamo iz tabele. Naziv "dinamičko programiranje" potječe iz teorije upravljanja, i danas je izgubio svoj prvobitni smisao.

Za razliku od algoritama tipa podijeli-pa-vladaj koji idu "s vrha prema dolje" (od većeg primjerka problema prema manjima), algoritmi dinamičkog programiranja idu s "dna prema gore"(od manjih primjeraka prema većem). Znači, prvo se u tabelu unose rješenja za primjerke najmanje veličine, zatim rješenja za malo veće primjerke, itd., sve dok se ne dosegne veličina zadanog primjerka. Važan je redoslijed ispunjavanja tabele.

Postupak dinamičkog programiranja koji put zahtijeva da riješimo i neke manje primjerke koji nam na kraju neće biti potrebni za rješenje zadanog primjerka. To se još uvijek više isplati nego rješavanje istih primjeraka mnogo puta.

6.2.2 Određivanje šanse za pobjedu u sportskom nadmetanju

Promatramo ovakvu situaciju. Dva sportaša ili tima A i B nadmeću se u nekoj sportskoj igri (disciplini). Igra je podijeljena u dijelove (setove, runde, partije, ...). U svakom dijelu igre točno jedan igrač bilježi 1 poen. Igra traje sve dok jedan od igrača ne skupi n poena gdje je n je unaprijed fiksiran - tada je taj igrač pobjednik. Pretpostavljamo da su A i B podjednako jaki, tako da svaki od njih ima 50% šanse da dobije poen u bilo kojem dijelu igre.

Označimo s $P(i,j)$ vjerojatnost da će A biti konačni pobjednik, u situaciji kad A treba još i poena za pobjedu a B treba još j poena. Npr. za $n = 4$, ako je A već dobio 2 poena a B je dobio 1 poen, tada je $i = 2$ i $j = 3$. Vidjet ćemo da je $P(2,3) = 11/16$, znači A ima više šanse za pobjedu nego B. Htjeli bi pronaći algoritam koji za zadane i, j računa $P(i,j)$.

Lako se vidi da vrijedi relacija:

$$P(i,j) = \begin{cases} 1, & \text{za } i = 0, j > 0, \\ 0, & \text{za } i > 0, j = 0, \\ \frac{1}{2}P(i-1,j) + \frac{1}{2}P(i,j-1), & \text{za } i > 0, j > 0. \end{cases}$$

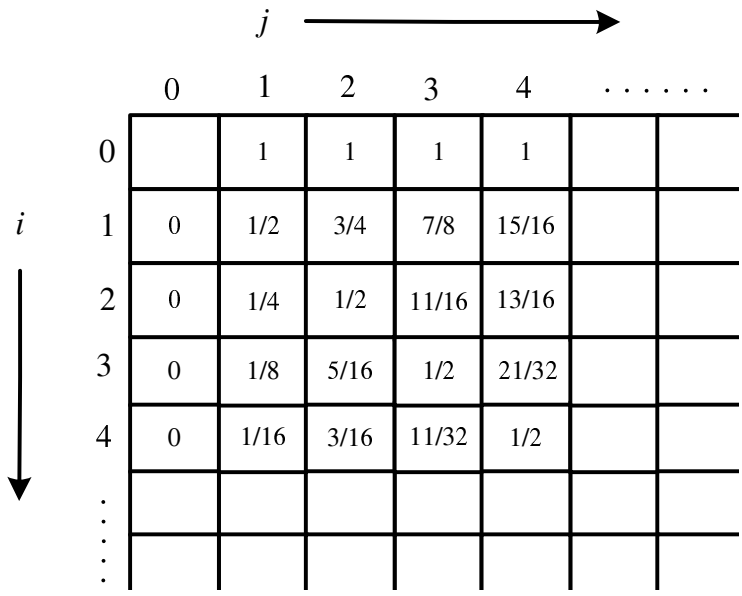
Prva dva reda u gornjoj formuli očigledno su ispravni. U uvjetima trećeg reda igra se bar još jedan dio igre, u kojem A ima 50% šanse da dobije poen; ako A dobije taj poen, tada mu je vjerojatnost konačne pobjede $P(i-1,j)$ inače mu je ta vjerojatnost $P(i,j-1)$.

Prethodna relacija može se iskoristiti za rekurzivno računanje $P(i,j)$, u duhu metode podijeli-pa-vladaj. No, pokazuje se da je vrijeme izvršavanja takvog računa $\mathcal{O}(2^{i+j})$. Razlog za toliku složenost je taj što se iste vrijednosti pozivaju (računaju) mnogo puta. Npr. da bi izračunali $P(2,3)$, trebali bi nam $P(1,3)$ i $P(2,2)$; dalje $P(1,3)$ i $P(2,2)$ oba zahtijevaju $P(1,2)$, što znači da će se $P(1,2)$ računati dvaput.

Bolji način za računanje $P(i,j)$ je ispunjavanje tabele prikazane na slici 6.5. Gornji red tabele ispunjen je jedinicama, a lijevi stupac nulama. Prethodna rekurzivna formula kaže da se bilo koji od preostalih elemenata tabele može dobiti kao aritmetička sredina elementa iznad i elemenata ulijevo. Znači, dobar način za popunjavanje tabele je da se ide po dijagonalama počevši od gornjeg lijevog ugla.

Algoritam se može zapisati sljedećom C funkcijom, koji radi nad globalnim dvodimenzionalnim realnim poljem `P[][]`.

```
float Odds(int i, j) {
  int s, k;
  for (s=1; s<=(i+j); s++) {
    /* računaj dijagonalu elemenata sa zbrojem indeksa s */
    P[0][s] = 1.0;
    P[s][0] = 0.0;
    for (k=1; k<s; k++)
      P[k][s-k] = ( P[k-1][s-k] + P[k][s-k-1] )/2.0;
  }
  return P[i][j];
}
```



Slika 6.5: Šanse za pobjedu u sportskom nadmetanju.

Unutrašnja petlja u gornjem potprogramu traje $\mathcal{O}(s)$, a vanjska petlja se ponavlja za $s = 1, 2, \ldots, (i+j)$. Dakle, računanje $P(i, j)$ sve skupa traje $\mathcal{O}\left(\sum_{s=1}^{i+j} s\right) = \mathcal{O}\left((i+j)^2\right)$.

6.2.3 Rješavanje 0/1 problema ranca

0/1 problem ranca (*0/1 knapsack problem*) glasi ovako. Zadano je n predmeta $O_1, O_2, \ldots, O_n$ i ranac. Predmet O_i ima težinu w_i i vrijednost p_i. Kapacitet ranca je c težinskih jedinica. Pitamo se: koje predmete treba staviti u ranac tako da ukupna težina ne prijeđe c, a da ukupna vrijednost bude maksimalna. Uzimamo da su w_i $(i = 1, 2, \ldots, n)$ i c pozitivni cijeli brojevi.

Primijetimo da 0/1 problem ranca spada u takozvane probleme **optimizacije**. Općenito, kod problema optimizacije od svih dopustivih rješenja tražimo samo ono koje je optimalno u nekom smislu. Dopustivo rješenje je ono koje zadovoljava ograničenja problema. Optimalnost se mjeri **funkcijom cilja** koju treba minimizirati odnosno maksimizirati. Ovisno o primjeni, ta funkcija cilja intepretira se kao cijena, trošak, vrijednost, zarada, itd. U našem slučaju svaki odabir predmeta koji ne premašuje kapacitet ranca predstavlja jedno dopustivo rješenje. Funkcija cilja koju želimo maksimizirati je ukupna vrijednost predmeta u rancu, dakle zbroj njihovih vrijednosti.

0/1 problem ranca mogao bi se riješiti algoritmom tipa podijeli-pa-vladaj koji bi generirao sve moguće podskupove skupa $\{O_1, O_2, \ldots, O_n\}$ i izabrao onaj s najvećom vrijednošću uz dopustivu težinu. No, takav algoritam očito bi imao složenost $\mathcal{O}(2^n)$.

Bolji algoritam dobiva se dinamičkim programiranjem. Označimo s M_{ij} maksimalnu vrijednost koja se može dobiti izborom predmeta iz skupa $\{O_1, O_2, \ldots, O_i\}$ uz kapacitet j. Prilikom postizavanja M_{ij}, predmet O_i je stavljen u ranac ili nije. Ako O_i nije stavljen, tada je $M_{ij} = M_{i-1,j}$. Ako je O_i stavljen, tada prije izabrani predmeti predstavljaju

optimalni izbor iz skupa $\{O_1, O_2, \ldots, O_{i-1}\}$ uz kapacitet $j - w_i$. Znači, vrijedi relacija:

$$M_{ij} = \begin{cases} 0, & \text{za } i = 0 \text{ ili } j \leq 0, \\ M_{i-1,j}, & \text{za } j < w_i \text{ i } i > 0, \\ \max\{M_{i-1,j}, (M_{i-1,j-w_i} + p_i)\}, & \text{inače.} \end{cases}$$

Algoritam se sastoji od ispunjavanja tabele s vrijednostima M_{ij}. Mi ustvari tražimo M_{nc}. Znači, tabela treba sadržavati M_{ij} za $0 \leq i \leq n$, $0 \leq j \leq c$, tj. mora biti dimenzije $(m+1) \times (c+1)$. Redoslijed računanja je redak-po-redak (j se brže mijenja nego i). Vrijeme izvršavanja algoritma je $\mathcal{O}(nc)$.

Algoritam ispunjavanja tabele zapisan je sljedećom funkcijom `ZeroOneKnapsack( )`. Koriste se globalna polja sljedećeg oblika s očiglednim značenjem, s time da se podaci u njih upisuju od indeksa 1 nadalje.

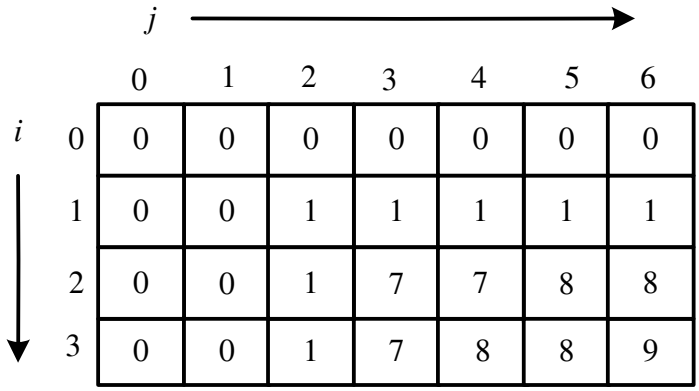
```
float M[D1][D2];
int w[D1];
float p[D1];
```

Funkcija vraća maksimalnu vrijednost koja se može prenijeti u rancu kapaciteta `c` ako se ograničimo na prvih `n` predmeta.

```
float ZeroOneKnapsack (int n, int c) {
  int i, j;
  for (i=0; i<=n; i++) M[i][0] = 0.0;
  for (j=0; j<=c; j++) M[0][j] = 0.0;
  for (i=1; i<=n; i++)
    for (j=1; j<=c; j++) {
      M[i][j] = M[i-1][j];
      if ( j >= w[i] )
        if ( (M[i-1][j-w[i]] + p[i]) > M[i-1][j] )
          M[i][j] = M[i-1][j-w[i]] + p[i];
    }
  return M[n][c];
}
```

Neka su npr. zadani ulazni podaci $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 7, 8)$, $c = 6$. Naš algoritam tada računa tabelu prikazanu na slici 6.6. Znači, maksimalna vrijednost koja se može nositi u rancu je $M_{3,6} = 9$. Ona se postiže izborom prvog i trećeg predmeta. Ukupna težina predmeta u rancu iznosi $2 + 4 = 6$.

Zapisana verzija algoritma (tabele) zapravo daje samo maksimalnu vrijednost koja se može ponijeti u rancu, a ne i optimalni izbor predmeta. Da bismo mogli reproducirati i optimalni izbor, algoritam trebamo malo usavršiti tako da uz svaki element M_{ij} čuvamo i "zastavicu" (logičku vrijednost) koja označava da li je M_{ij} dobiven kao $M_{i-1,j}$ ili kao $M_{i-1,j-w_i} + p_i$. Zastavica uz M_{nc} će nam najprije reći da li je predmet O_n upotrebljen u optimalnom izboru. Ovisno o ne-izboru (odnosno izboru) O_n, dalje gledamo zastavicu uz $M_{n-1,c}$ (odnosno zastavicu uz $M_{n-1,c-w_n}$) da bismo ustanovili da li je O_{n-1} bio izabran, itd.



Slika 6.6: Rješavanje 0/1 problema ranca metodom dinamičkog programiranja.

6.3 Pohlepni pristup

Pohlepni pristup je metoda oblikovanja algoritama koja je primjenjiva isključivo na probleme optimizacije. Rješenje je obično približno (dakle ne sasvim optimalno), no dobiva se vrlo brzo u nizu jednostavnih koraka. U ovom odjeljku uz općeniti opis pohlepnog pristupa slijedi nekoliko primjera dobrih pohlepnih algoritama.

6.3.1 Općenito o pohlepnom pristupu

Zamislimo da radimo u dućanu i da mušteriji trebamo vratiti 62 kune. Na raspolaganju nam stoje apoeni (novčanice, kovanice) od 50, 20, 10, 5 i 1 kune. Gotovo instinktivno postupit ćemo tako da vratimo 1×50, 1×10 i 2×1 kuna. Ne samo da smo vratili točan iznos, nego smo također izabrali najkraću moguću listu novčanica (4 komada).

Algoritam koji smo nesvjesno koristili bio je da izaberemo najveći apoen koji ne prelazi 62 kune (to je 50 kuna), stavimo ga u listu za vraćanje, pa vraćenu vrijednost odbijemo od 62 tako da dobijemo 12. Zatim biramo najveći apoen koji ne prelazi 12 kuna (to je 10 kuna), dodajemo ga na listu, itd.

Ovaj algoritam je primjer **pohlepnog pristupa** (*greedy approach*). Riječ je o pristupu koji služi isključivo za rješavanje problema optimizacije. Rješenje zadanog primjerka problema konstruiramo u nizu koraka (faza). U svakom koraku biramo mogućnost koja je "lokalno optimalna" u nekom smislu. Nadamo se da ćemo tako doći do "globalno optimalnog" ukupnog rješenja. U našem slučaju, lokalni kriterij optimalnosti je veličina apoena, a globalni kriterij (ustvari funkcija cilja) je duljina konačne liste za vraćanje. Nadamo se da ćemo uzastopnim izborom što većih apoena osigurati da broj vraćenih apoena bude minimalan.

Pohlepni pristup ne mora uvijek dovesti do optimalnog rješenja. Algoritam vraćanja ostatka funkcionirao je samo zahvaljujući specijalnim svojstvima raspoloživih apoena. Ako bi apoeni bili 11, 5 i 1 kuna, a trebali bismo vratiti 15 kuna, tada bi pohlepni algoritam najprije izabrao 11 kuna, a zatim bi morao 4 puta uzeti po 1 kunu, što bi dalo 5 novčanica. S druge strane, 3×5 kuna bi bila kraća lista.

Postoje problemi kod kojih pohlepni pristup daje korektni algoritam, dakle onaj koji zaista uvijek daje optimalno rješenje (to onda treba dokazati). Postoje također

problemi kod kojih pohlepni algoritam nije sasvim korektan, to jest ne daje uvijek zaista optimalno rješenje, ali predstavlja dobru heuristiku koja pronalazi rješenje blizu optimalnom (to onda treba eksperimentalno provjeriti).

6.3.2 Optimalni plan sažimanja sortiranih listi

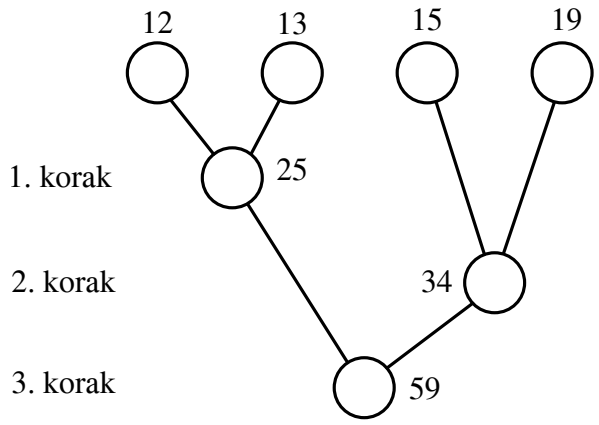
Imamo n sortiranih listi s duljinama $w_1, w_2, \ldots, w_n$. Trebamo ih sažeti u jednu veliku sortiranu listu. Sažimanje provodimo u $n-1$ koraka, tako da u svakom koraku sažmemo dvije odabrane liste u jednu (uobičajenim algoritmom *merge*). Zanima nas **optimalni plan sažimanja**, tj. takav izbor listi u pojedinom koraku koji će dovesti do najmanjeg ukupnog broja operacija. Ovaj problem optimizacije bio je obrađen na vježbama. Pokazalo se da se optimalni plan može konstruirati "pohlepnim" **Huffmanovim algoritmom**. Algoritam jednostavno kaže sljedeće:

> U svakom koraku sažimanja treba odabrati dvije najkraće liste koje nam stoje na raspolaganju.

Ili drukčije rečeno:

> Svaki korak sažimanja treba izvesti tako da imamo najmanje posla u tom koraku.

Ideja je ilustrirana primjerom na slici 6.7, gdje treba sažeti četiri sortirane liste s duljinama 12, 13, 15 i 19.



Slika 6.7: Optimalno sažimanje sortiranih listi pohlepnim Huffmanovim algoritmom.

Nije sasvim očigledno da ćemo ovom strategijom zaista postići najmanji ukupan broj operacija. No, može se dokazati (pomoću binarnih stabala i primjenom matematičke indukcije) da ipak hoćemo.

6.3.3 Rješavanje kontinuiranog problema ranca

Kontinuirani problem ranca (*continuous knapsack problem*) je problem optimizacije sličan 0/1 rancu iz odjeljka 6.2. No sada se predmeti koje stavljamo u ranac mogu "rezati", tj. ako ne stane cijeli predmet tada možemo staviti samo njegov dio i time dobiti odgovarajući (proporcionalni) dio vrijednosti. Stroga formulacija problema glasi:

Zadan je prirodni broj n i pozitivni realni brojevi c, w_i $(i = 1, 2, \ldots, n)$, p_i $(i = 1, 2, \ldots, n)$. Traže se realni brojevi x_i, $(i = 1, 2, \ldots, n)$, takvi da $\sum_{i=1}^{n} p_i x_i \rightarrow \max$, uz ograničenja $\sum_{i=1}^{n} w_i x_i \leq c$, $0 \leq x_i \leq 1$, $(i = 1, 2, \ldots, n)$.

Interpretacija zadanih podataka je ista kao u potpoglavlju 6.2. Broj x_i kaže koliki dio i-tog predmeta stavljamo u ranac.

Pohlepni pristup rješavanja problema zahtijeva da za svaki predmet izračunamo njegovu "profitabilnost" p_i/w_i (tj. vrijednost po jedinici težine). Predmete sortiramo silazno po profitabilnosti te ih u tom redoslijedu stavljamo u ranac dok god ima mjesta. Kod svakog stavljanja u ranac nastojimo spremiti što veći dio predmeta. Postupak je u jeziku C opisan funkcijom `ContKnapsack( )` koja radi nad globalnim poljima `w[ ], p[ ], x[ ]` s očiglednim značenjem. Pretpostavljamo da su podaci u poljima `w[]` i `p[]` već uređeni tako da je `p[i]/w[i] >= p[i+1]/w[i+1]`.

```
#define MAXLENGTH ...  /* dovoljno velika konstanta */
float w[MAXLENGTH], p[MAXLENGTH], x[MAXLENGTH];

void ContKnapsack (int n, float c) {
  float cu:
  int i;
  for (i=1; i<=n; i++) x[i] = 0.0;
  cu = c;
  for (i=1; i<=n; i++) {
    if (w[i] > cu) {
      x[i] = cu/w[i];
      return;
    }
    x[i] = 1.0;
    cu -= w[i];
  }
  return;
}
```

Kao primjer, promatrajmo rad našeg pohlepnog algoritma na podacima iz odjeljka 6.2, dakle za $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 7, 8)$, $c = 6$. Profitabilnosti predmeta su $p_1/w_1 = 1/2$, $p_2/w_2 = 7/3$, $p_3/w_3 = 2$. Znači, najprofitabilniji je drugi predmet, a idući po profitabilnosti je treći predmet. Zato algoritam smješta u ranac cijeli drugi predmet, a ostatak kapaciteta popunjava s 3/4 trećeg predmeta. Dakle rješenje je $(x_1, x_2, x_3) = (0, 1, 3/4)$, a maksimalna vrijednost ranca je 13.

Što se tiče korektnosti ovog algoritma, vrijedi sljedeća tvrdnja:

Opisani pohlepni algoritam za rješavanje kontinuiranog problema ranca uvijek daje optimalno rješenje problema.

U nastavku slijedi dokaz ove tvrdnje. Zaista, možemo uzeti da je $p_1/w_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n$. Neka je $X = (x_1, x_2, \ldots, x_n)$ rješenje generirano našim pohlepnim algoritmom.

Ako su svi x_i jednaki 1, tada je vrijednost ranca očito maksimalna. Zato, neka je j najmanji indeks takav da je $x_j < 1$. Zbog pretpostavke o sortiranosti po profitabilnosti slijedi da je $x_i = 1$ za $1 \leq i < j$, $x_i = 0$ za $j < i \leq n$. Također mora biti $\sum_{i=1}^{n} w_i x_i = c$.

Neka je $Y = (y_1, y_2, \ldots, y_n)$ jedno optimalno rješenje. Možemo uzeti da je $\sum_{i=1}^{n} w_i y_i = c$. Ako je $X = Y$ tada smo pokazali da je X optimalan. Zato pretpostavimo da je $X \neq Y$. Tada se može naći najmanji indeks k takav da je $x_k \neq y_k$. Tvrdimo najprije da je $k \leq j$, a zatim da je $y_k < x_k$. Da bi se u to uvjerili, promatramo tri mogućnosti:

1. Za $k > j$ izlazi da je $\sum_{i=1}^{n} w_i y_i > c$ što je nemoguće.

2. Za $k < j$ izlazi da je $x_k = 1$. No $y_k \neq x_k$ pa mora biti $y_k < x_k$.

3. Za $k = j$, zbog $\sum_{i=1}^{j} w_i x_i = c$ i $y_i = x_i$ $(1 \leq i < j)$ slijedi da je ili $y_k < x_k$ ili $\sum_{i=1}^{n} w_i y_i > c$.

Uvećajmo sada y_k da postane jednak x_k, te umanjimo vrijednosti $(y_{k+1}, \ldots, y_n)$ koliko treba da ukupna težina ostane ista. Time dobivamo novo rješenje $Z = (z_1, z_2, \ldots, z_n)$ sa svojstvima: $z_i = x_i$ $(1 \leq i \leq k)$, $\sum_{i=k+1}^{n} w_i(y_i - z_i) = w_k(z_k - y_k)$. Z također mora biti optimalno rješenje jer vrijedi:

$$\begin{aligned}
\sum_{i=1}^{n} p_i z_i &= \sum_{i=1}^{n} p_i y_i + (z_k - y_k)p_k - \sum_{i=k+1}^{n} (y_i - z_i)p_i \\
&= \sum_{i=1}^{n} p_i y_i + (z_k - y_k)w_k \frac{p_k}{w_k} - \sum_{i=k+1}^{n} (y_i - z_i)w_i \frac{p_i}{w_i} \\
&= \ldots \text{zbog svojstva } \frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \\
&\geq \sum_{i=1}^{n} p_i y_i + \left[(z_k - y_k)w_k - \sum_{i=k+1}^{n} (y_i - z_i)w_i \right] \frac{p_k}{w_k} \\
&\geq \sum_{i=1}^{n} p_i y_i + 0 \cdot \frac{p_k}{w_k} \\
&= \sum_{i=1}^{n} p_i y_i.
\end{aligned}$$

Dakle našli smo novo optimalno rješenje Z koje se "bolje poklapa" s X nego što se Y poklapa s X. Iteriranjem ove konstrukcije prije ili kasnije ćemo doći do optimalnog rješenja koje se u potpunosti poklapa s X. Znači, X je optimalan. Time je dokaz tvrdnje završen.

Na kraju, primijetimo da se i za 0/1 problem ranca također može definirati sličan pohlepni algoritam, no taj algoritam ne bi morao obavezno davati korektno (tj. optimalno) rješenje. Promatrajmo opet primjer iz odjeljka 6.2, dakle $c = 6$, $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 7, 8)$. Pohlepni pristup zahtijevao bi da se u ranac najprije stavi najprofitabilniji drugi predmet. Nakon toga treći predmet (idući po profitabilnosti) više ne bi stao, pa bi (zbog nemogućnosti rezanja) mogli dodati još samo prvi predmet. Dobili bi suboptimalno rješenje s vrijednošću 8. Naime, u odjeljku 6.2 pokazali smo da optimalno rješenje ima vrijednost 9.

6.4 *Backtracking*

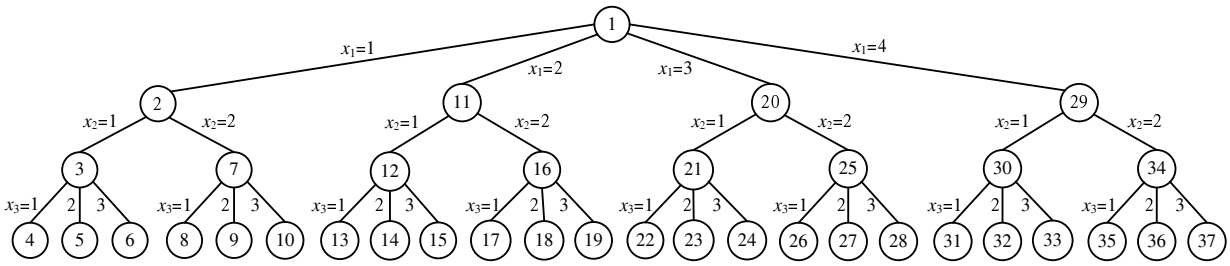
Backtracking je vrlo općenita metoda koja se primjenjuje za teške kombinatorne probleme. Rješenje problema traži se sistematskim ispitivanjem svih mogućnosti za konstrukciju tog rješenja.

U ovom odjeljku najprije ćemo dati općeniti opis *backtracking*-a. Zatim ćemo promatati dva kombinatorna problema koji su podobni za rješavanje tom metodom. Prvi od njih, problem n kraljica, riješit ćemo standardnim *backtracking* algoritmom. Drugi od njih, problem trgovačkog putnika, zapravo je problem optimizacije, pa ćemo ga rješavati optimizacijskom varijantom *backtracking*-a koja se naziva *branch-and-bound*.

6.4.1 Opći oblik *backtracking* algoritma

Metoda *backtracking* zahtijeva da se rješenje promatranog problema izrazi kao n-torka oblika $(x_1, x_2, \ldots, x_n)$, gdje je x_i element nekog konačnog skupa S_i. Kartezijev produkt $S_1 \times S_2 \times \ldots \times S_n$ zove se **prostor rješenja**. Da bi neka konkretna n-torka iz prostora rješenja zaista predstavljala rješenje, ona mora zadovoljiti još i neka dodatna **ograničenja** (ovisna o samom problemu).

Prostor rješenja treba zamišljati kao uređeno **stablo rješenja**. Korijen tog stabla predstavlja sve moguće n-torke. Dijete korijena predstavlja sve n-torke gdje prva komponenta x_1 ima neku određenu vrijednost. Unuk korijena predstavlja sve n-torke gdje su prve dvije komponente x_1 i x_2 fiksirane na određeni način, itd. List stabla predstavlja jednu konkretnu n-torku. Npr. za $n = 3$, $S_1 = \{1, 2, 3, 4\}$, $S_2 = \{1, 2\}$, $S_3 = \{1, 2, 3\}$, stablo rješenja izgleda kao na slici 6.8.



Slika 6.8: Primjer stabla rješenja.

Backtracking je u osnovi rekurzivni algoritam koji se sastoji od simultanog generiranja i ispitivanja čvorova u stablu rješenja. Čvorovi se stavljaju na stog. Jedan korak algoritma sastoji se od toga da se uzme čvor s vrha stoga te da se provjeri da li taj čvor predstavlja rješenje problema. Ukoliko čvor predstavlja rješenje, tada se poduzme odgovarajuća akcija - na primjer ispis rješenja. Ukoliko čvor nije rješenje, tada se pokušaju generirati njegova djeca te se ona stavljaju na stog. Algoritam počinje tako da na stogu bude samo korijen stabla; završetak je onda kad nađemo rješenje ili kad se isprazni stog. Na prethodnoj slici 6.8 redoslijed obrađivanja čvorova (skidanja sa stoga) poklapa se s numeracijom čvorova.

Veličina prostora rješenja raste eksponencijalno s veličinom problema n. Zbog toga dobar *backtracking* algoritam nikad ne generira cijelo stablo rješenja, već "reže" one grane (podstabla) za koje uspije utvrditi da ne vode do rješenja. Naime, u samom

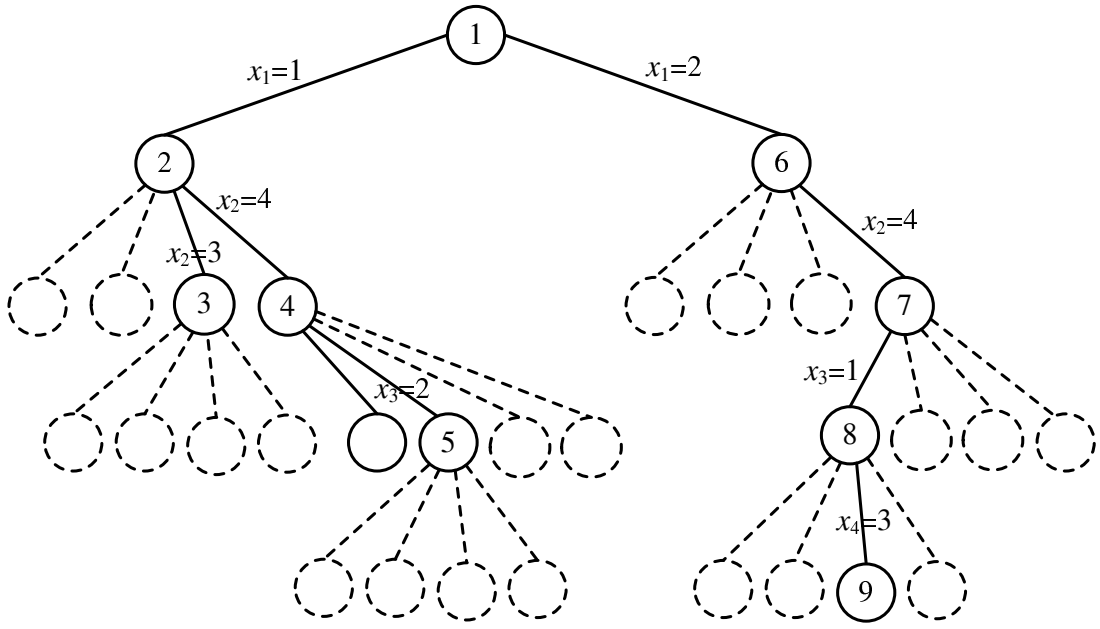
postupku generiranja čvorova provjeravaju se ograničenja koja n-torka $(x_1, x_2, \ldots, x_n)$ mora zadovoljiti da bi zaista bila rješenje. Čvor na i-tom nivou predstavlja n-torke gdje je prvih i komponenti $x_1, x_2, \ldots, x_i$ fiksirano na određeni način. Ukoliko se već na osnovi vrijednosti tih i komponenti može utvrditi da ograničenja nisu zadovoljena, tada dotični čvor ne treba generirati jer ni on ni njegova djeca neće dati rješenje. Npr. ako u kontekstu slike 6.8 vrijedi ograničenje da komponente x_1, x_2, x_3 moraju biti međusobno različite, tada "najljevija" generirana grana stabla postaje ona za $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, a ukupan broj generiranih čvorova pada na 19.

6.4.2 Rješavanje problema n kraljica

Problem n kraljica glasi ovako: na šahovsku ploču dimenzije $n \times n$ treba postaviti n kraljica tako da se one međusobno ne napadaju. Pokazuje se da problem ima rješenje za sve n-ove veće od 3, dapače broj rješenja raste s n. Za standardnu šahovsku ploču gdje je $n = 8$ postoje 92 rješenja, dakle 92 različite konfiguracije od po 8 kraljica koje se međusobno ne napadaju. Od ta 92 rješenja 12 je bitno različitih, a ostala se mogu dobiti iz tih 12 rotacijama ili simetrijama šahovske ploče.

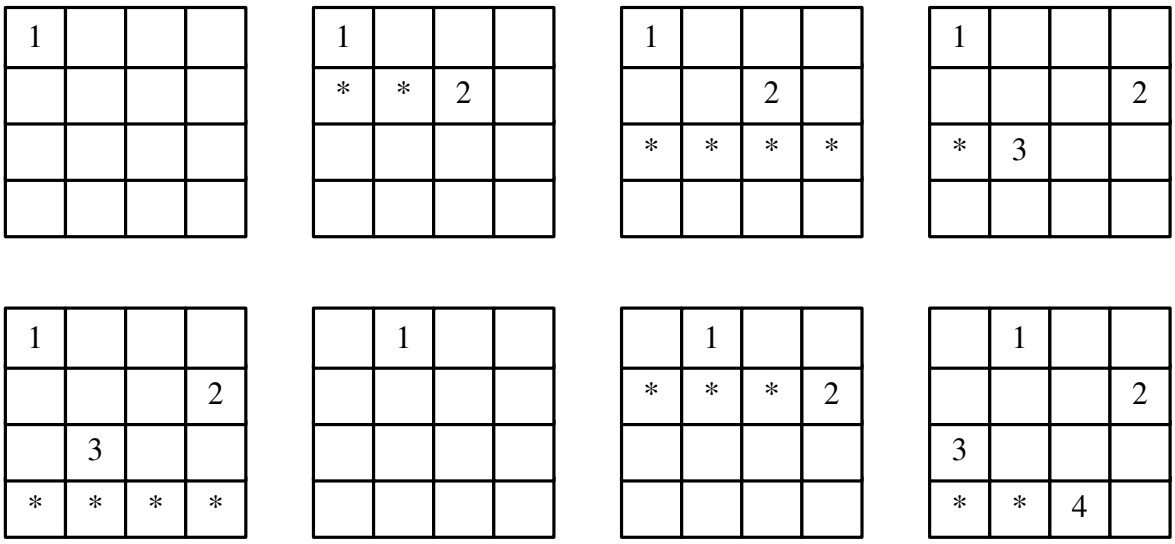
U nastavku objašnjavamo način kako da se problem opiše i riješi u skladu s metodom *backtracking*. Očito svaka kraljica mora biti u posebnom retku ploče. Zbog toga možemo uzeti da je i-ta kraljica u i-tom retku. Rješenje problema se može predočiti kao n-torka $(x_1, x_2, \ldots, x_n)$, gdje je x_i indeks stupca u kojem se nalazi i-ta kraljica. Znači $S_i = \{1, 2, \ldots, n\}$ za sve i. Broj n-torki u prostoru rješenja je n^n. Ograničenja koja rješenje $(x_1, x_2, \ldots, x_n)$ mora zadovoljiti izvode se iz zahtjeva da se nikoje dvije kraljice ne smiju naći u istom stupcu niti na istoj dijagonali.

Za $n = 4$ *backtracking* algoritam generira stablo rješenja prikazano na slici 6.9. Rad algoritma prekinut je čim je pronađeno prvo rješenje. Crtkano su označeni čvorovi koje je algoritam u postupku generiranja odmah poništio jer krše ograničenja.



Slika 6.9: Stablo rješenja za problem n kraljica.

Slika 6.10 ilustrira ove iste korake algoritma kao pomake figura na šahovskoj ploči. Zvjezdice označavaju mjesta gdje je algoritam pokušao smjestiti kraljicu, ali je odmah odustao zbog napada druge kraljice.



Slika 6.10: Koraci algoritma za problem n kraljica.

Slijedi precizni zapis *backtracking* algoritma za problem n kraljica u jeziku C. Pretpostavljamo da postoji globalno polje za smještanje rješenja $(x_1, x_2, \ldots, x_n)$ oblika:

```
#define MAXLENGTH ...  /* dovoljno velika konstanta */
int x[MAXLENGTH]; /* 0-ti element polja se ne koristi!  */
```

Najprije slijedi pomoćna logička funkcija `place( )` koja kaže da li se `k`-ta kraljica može postaviti u `k`-ti redak i `x[k]`-ti stupac tako da je već postavljene 1.,2.,...,`(k-1)`-va kraljica ne napadaju. Funkcija `Queens( )` ispisuje sva rješenja problema.

```
int place (int k) {
  /* pretpostavljamo da elementi polja x s indeksima */
  /* 1,2,...,k već imaju zadane vrijednosti */
  int i;
  for (i=1; i<k; i++)
    if ( (x[i]==x[k]) || (abs(x[i]-x[k])==abs(i-k)) )
      return 0;
  return 1;
}

void Queens (int n) {
  int k;
  k=1; x[1]=0; /* k je tekući redak, x[k] je tekući stupac*/
```

```
  while (k > 0) { /* ponavljaj za sve retke (kraljice) */
    x[k]++;
    while ( (x[k]<=n) && (place(k)==0) ) x[k]++; /* nađi stupac */
    if (x[k] <= n) /* stupac za kraljicu je nađen */
      if (k == n) /* rješenje je kompletirano */
        ...ispis...
      else { /* promatraj sljedeći redak (kraljicu) */
        k++; x[k]=0;
      }
    else k--; /* vrati se u prethodni redak */
  }
}
```

Makar je *backtracking* u osnovi rekurzivan algoritam, naša funkcija `Queens( )` ne koristi rekurziju. Razlog je taj što smo rekurziju uspješno preveli u iteraciju. Za takvo prevođenje u pravilu je potreban stog. No u našem slučaju sama šahovska ploča (opisana poljem `x[ ]`) funkcionira kao zamjena za stog budući da ona pamti trenutne položaje kraljica te omogućuje vraćanje u prethodne konfiguracije.

6.4.3 Rješavanje problema trgovačkog putnika

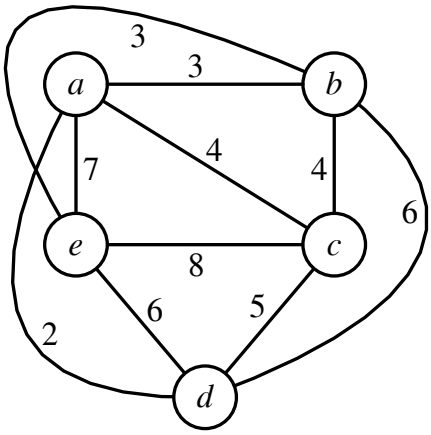
Problem trgovačkog putnika (*traveling salesman problem*) glasi ovako. Zadan je potpuni neusmjereni graf čijim bridovima su pridružene "cijene". Treba pronaći Hamiltonov ciklus s minimalnom cijenom. Potpuni graf je onaj u kojem postoji brid između bilo koja dva vrha. Dakle, ako je n broj vrhova a m broj bridova, tada je $m = n(n-1)/2$. Hamiltonov ciklus je kružni put u grafu koji prolazi svakim vrhom grafa točno jednom. Cijena puta je zbroj cijena odgovarajućih bridova.

Graf na slici 6.11 opisuje jedan konkretan primjerak problema trgovačkog putnika za $n = 5$ i $m = 10$. Hamiltonovi ciklusi u tom grafu određeni su permutacijama vrhova. Na primjer, jedan Hamiltonov ciklus je onaj koji obilazi vrhove u redoslijedu (a, b, c, d, e, a) uz cijenu $3 + 4 + 5 + 6 + 7 = 25$. Redoslijed obilaska (a, c, b, e, d, a) ima manju cijenu $4 + 4 + 3 + 6 + 2 = 19$. Vidjet ćemo da je ovaj drugi ciklus ustvari optimalan, to jest on predstavlja rješenje problema.

Primijetimo da, za razliku od n kraljica, trgovački putnik spada u probleme optimizacije. Rekli smo već da kod problema optimizacije od svih dopustivih rješenja tražimo ono koje je najbolje sa stanovišta odabrane funkcije cilja. U našem slučaju svaki Hamiltonov ciklus predstavlja dopustivo rješenje. Funkcija cilja koju želimo minimizirati je cijena Hamiltonovog ciklusa, dakle zbroj cijena uključenih bridova.

Kad metodu *backtracking* koristimo za probleme optimizacije, tada u njoj možemo napraviti neke preinake koje joj dodatno ubrzavaju rad. Naime, vidjeli smo da u stablu rješenja uvijek treba rezati one grane koje ne vode do rješenja. No kod problema optimizacije također se mogu rezati i one grane koje ne vode do boljeg rješenja (u odnosu na ono koje već imamo). Takva poboljšana varijanta *backtracking*-a obično se naziva ***branch-and-bound***.

Uzmimo zbog konkretnosti da rješavamo problem minimizacije - dakle funkcija cilja je cijena koju želimo minimizirati. Za realizaciju *branch-and-bound*-a tada nam je potreban postupak određivanja donje ograde za cijenu po svim rješenjima iz zadanog pod-



Slika 6.11: Primjerak problema trgovačkog putnika.

stabla stabla rješenja. Ako ta donja ograda izađe veća od cijene najboljeg trenutno poznatog rješenja, tada se dotično podstablo može izbaciti iz razmatranja jer ono ne vodi do boljeg rješenja. Za uspjeh *branch-and-bound* algoritma također je važno da se što prije otkrije "dobro" rješenje. Uobičajena heuristika je da se pri obradi čvorova-braće prednost daje onom koji ima manju donju ogradu.

U nastavku opisujemo jedan mogući način rješavanja problema trgovačkog putnika metodom *branch-and-bound*. Dakle opisujemo: prikaz rješenja problema pomoću m-torke, izgled prostora rješenja odnosno stabla rješenja, ograničenja kojima se režu grane koje ne vode do rješenja te način računanja donjih ograda za cijene Hamiltonovih ciklusa iz zadanog podstabla stabla rješenja.

Uzmimo da su imena vrhova elementi nekog dobro uređenog skupa, npr. $\{a, b, c, d, e\}$. Bridove tada možemo poredati leksikografski: (a, b), (a, c), (a, d), (a, e), (b, c), (b, d), (b, e), (c, d), (c, e), (d, e), što znači da svaki brid dobiva svoj redni broj između 1 i m. Rješenje problema može se predočiti kao m-torka $(x_1, x_2, \ldots, x_m)$ gdje je x_i logička vrijednost koja kaže da li je i-ti brid uključen ili nije. Znači $S_i = \{0, 1\}$ za sve i. Broj m-torki u prostoru rješenja je 2^m. Stablo rješenja izgleda kao na slici 6.12, dakle kao binarno stablo. Jedan čvor u stablu rješenja predstavlja sve konfiguracije bridova gdje su neki određeni bridovi "obavezni" a neki drugi određeni bridovi su "zabranjeni".

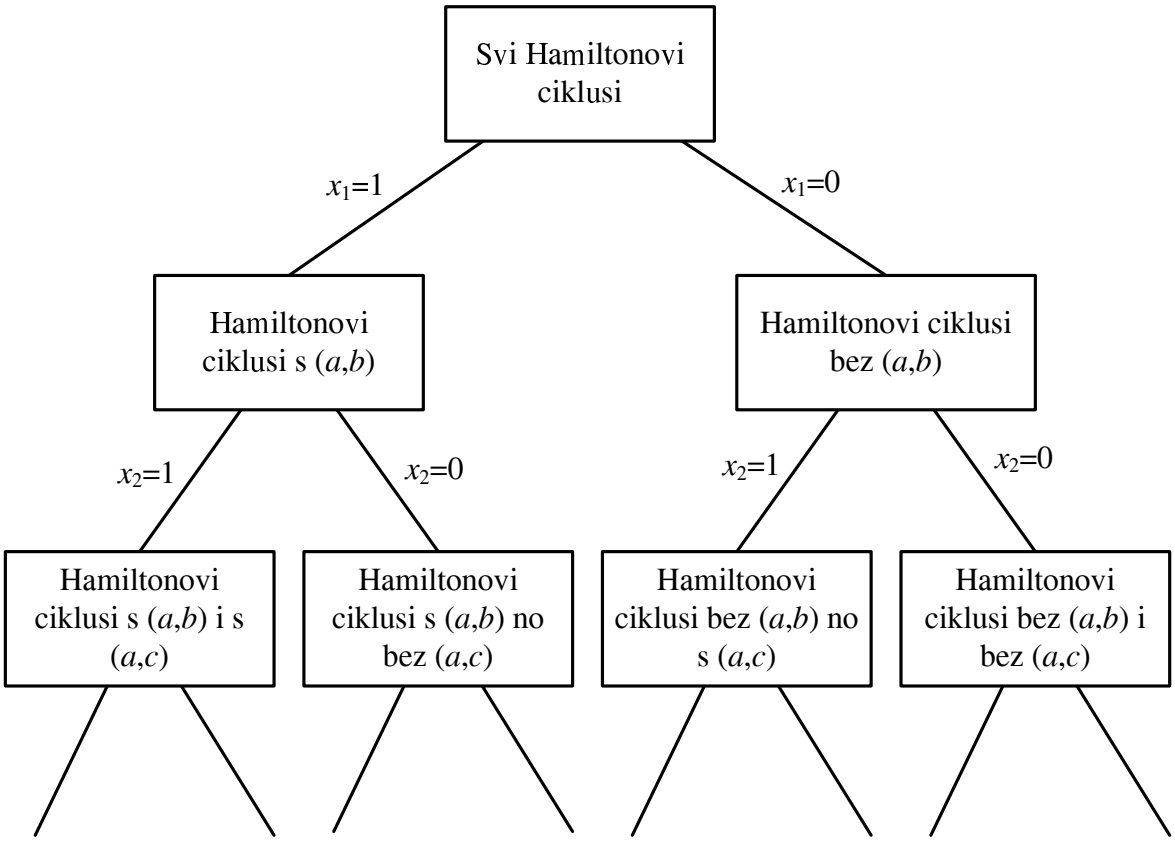
Ograničenja koja rješenje $(x_1, x_2, \ldots, x_m)$ mora zadovoljiti svode se na to da dotična konfiguracija bridova mora predstavljati Hamiltonov ciklus. To opet daje sljedeća pravila za provjeru prilikom generiranja čvorova stabla:

1. Ako bi isključenje brida (x, y) učinilo nemogućim da x (odnosno y) ima bar dva incidentna (dodirujuća) brida u konfiguraciji, tada (x, y) mora biti uključen.

2. Ako bi uključenje brida (x, y) uzrokovalo da x (ili y) ima više od dva incidentna brida u konfiguraciji, tada (x, y) mora biti isključen.

3. Ako bi uključenje brida (x, y) zatvorilo ne-Hamiltonov ciklus u konfiguraciji, tada (x, y) mora biti isključen.

Primijetimo da cijenu zadanog Hamiltonovog ciklusa možemo računati kao zbroj, tako da za svaki vrh grafa pribrojimo cijenu dvaju njemu incidentnih bridova iz ciklusa.



Slika 6.12: Stablo rješenja za problem trgovačkog putnika.

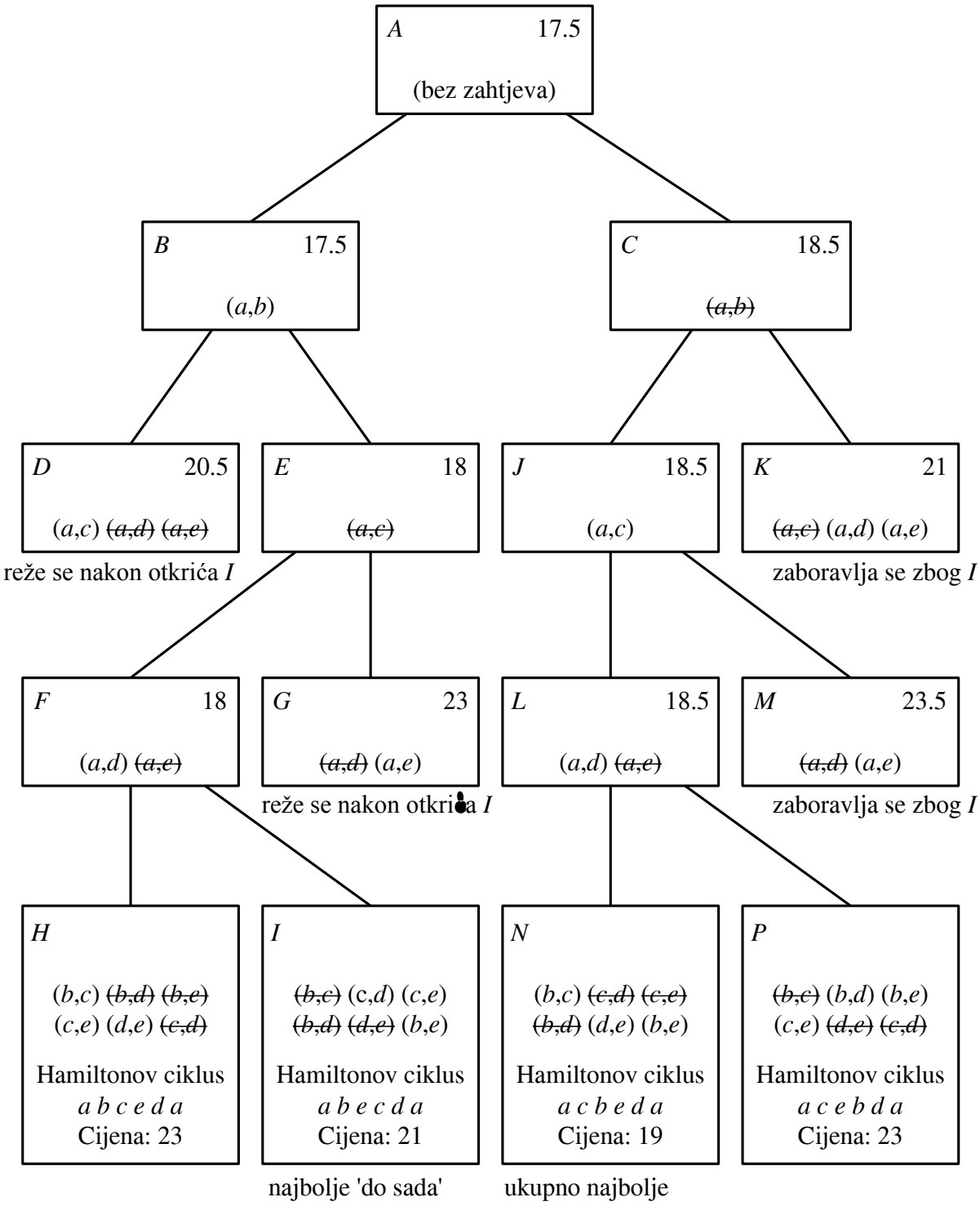
Time je svaki brid iz ciklusa uračunat dvaput, pa rezultat dobivamo dijeljenjem ukupnog zbroja s 2. Na osnovu ove primjedbe dobivamo sljedeći postupak računanja donje ograde za cijenu bilo kojeg Hamiltonovog ciklusa:

- Za svaki vrh x nađemo dva najjeftinija brida u grafu koji su incidentni s x te zbrojimo cijene tih bridova.
- "Male" zbrojeve, dobivene za pojedine vrhove, zbrojimo zajedno u "veliki" zbroj.
- "Veliki" zbroj podijelimo s 2.

Npr. za graf sa slike 6.11 donja ograda bi izašla $(5 + 6 + 8 + 7 + 9)/2 = 17.5$, što zbog cjelobrojnosti možemo zaokružiti na 18.

Da bi dobili precizniju donju ogradu za cijenu Hamiltonovog ciklusa koji pripada određenom podstablu iz stabla rješenja, postupamo slično kao gore, no uzimamo u obzir već fiksiranu uključenost odnosno isključenost nekih bridova:

- Ako je brid (x, y) uključen, tada njegovu cijenu treba uračunati u "male" zbrojeve vrhova x i y, bez obzira da li je on jeftin ili nije;
- Ako je brid (x, y) isključen, tada njegovu cijenu ne smijemo uračunati u "mali" zbroj vrha x odnosno y.



Slika 6.13: Stablo rješenja generirano *branch-and-bound* algoritmom za primjerak problema trgovačkog putnika sa slike 6.11.

Npr. za podstablo gdje je (a, b) isključen i (a, c) uključen, donja ograda bi izašla $(6 + 7 + 8 + 7 + 9)/2 = 18.5$, dakle 19 nakon zaokruživanja.

Na osnovi svih ovih ideja moguće je konstruirati *backtracking* algoritam za rješavanje problema trgovačkog putnika, štoviše njegovu *branch-and-bound* varijantu. Za konkretni graf sa slike 6.11, algoritam generira stablo prikazano slikom 6.13. Pronalazi se optimalni Hamiltonov ciklus (a, c, b, e, d, a) sa cijenom 19.

U svaki čvor na slici 6.13 upisano je njegovo ime (veliko slovo), pripadna donja ograda te zahtjevi na uključenost odnosno isključenost nekih bridova. Zahtjevi iz nadređenog čvora se prenose i na sve podređene čvorove. Na svakoj razini stabla uvodi se zapravo samo jedan novi zahtjev, ostali upisani zahtjevi su posljedica naših pravila za provjeru ograničenja. Primjerice, u čvoru D originalno se zahtijeva da bridovi (a, b) i (a, c) budu uključeni, no onda nužno (a, d) i (a, e) moraju biti isključeni zbog pravila 2. Čvorovi su imenovani abecedno u redoslijedu svojeg nastajanja. Taj redoslijed je posljedica naše heuristike da se prije razgrađuje onaj brat koji ima manju donju ogradu.

6.5 Lokalno traženje

Lokalno traženje još je jedna metoda oblikovanja algoritama koja je primjenjiva isključivo na probleme optimizacije. Slično kao kod pohlepnog pristupa, dobivaju se brzi no u pravilu približni algoritmi. Za razliku od pohlepnog pristupa, rješenje se ne konstruira od početka. Umjesto toga, krene se od već poznatog rješenja pa ga se poboljšava.

Ovaj odjeljak započinje općenitim opisom lokalnog traženja. Zatim slijede dva konkretna algoritma tog tipa: prvi od njih rješava već prije razmatrani problem trgovačkog putnika, a drugi se bavi traženjem optimalnog linearnog razmještaja. Na kraju se raspravlja o manjkavostima lokalnog traženja te o njegovim poboljšanim varijantama.

6.5.1 Općenito o lokalnom traženju

Lokalno traženje (*local search*) zahtijeva da u svakom trenutku raspolažemo jednim dopustivim rješenjem zadanog primjerka razmatranog problema optimizacije. To tekuće rješenje nastoji se poboljšati. Promatramo okolinu (susjedstvo) tekućeg rješenja, dakle skup dopustivih rješenja koja su "blizu" tekućem, u smislu da se mogu dobiti nekom jednostavnom transformacijom tekućeg. Biramo najbolje rješenje iz okoline, ili bar rješenje koje je bolje od tekućeg. Izabrano rješenje proglašava se novim tekućim. Postupak se nastavlja dok god je moguće, to jest sve dok ne dobijemo tekuće rješenje u čijoj okolini nema boljeg rješenja.

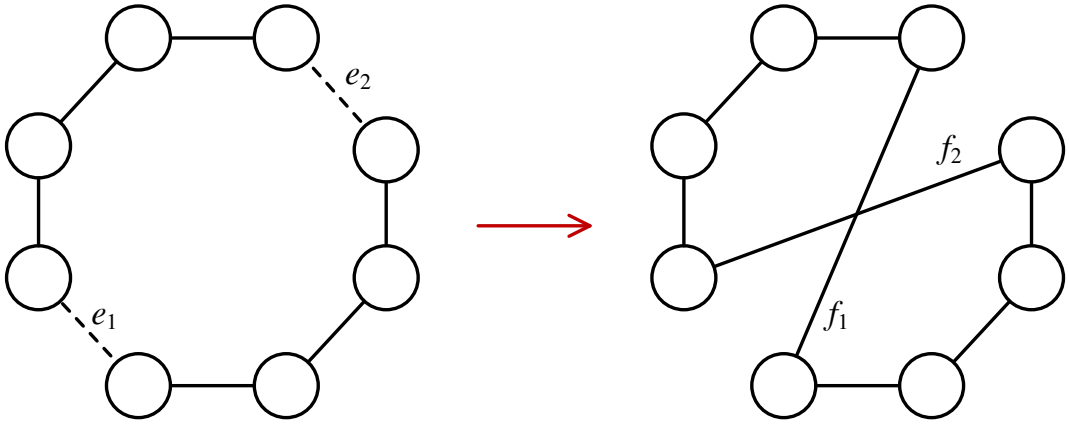
Algoritmi dobiveni strategijom lokalnog traženja relativno su su brzi, no oni samo približno rješavaju postavljeni problem optimizacije. Rješenje koje oni daju obično nije optimalno, ali je bolje od polaznog. Lokalno traženje primjenjuje se za teške kombinatoričke probleme, gdje računanje stvarno optimalnog rješenja iziskuje previše vremena, pa se zbog toga moramo zadovoljiti približnim rješenjem.

6.5.2 2-opt algoritam za problem trgovačkog putnika

Kao prvi primjer algoritma zasnovanog na lokalnom traženju promatramo takozvani **2-opt algoritam** za rješavanje problema trgovačkog putnika. Znači opet promatramo

potpuni neusmjereni graf čijim bridovima su zadane cijene. Cijenu puta u grafu računamo kao zbroj cijena pripadnih bridova. Tražimo Hamiltonov ciklus s minimalnom cijenom.

2-opt algoritam pamti bilo koji Hamiltonov ciklus kao tekuće rješenje. Promatra se takozvana 2-okolina tekućeg ciklusa: nju čine svi Hamiltonovi ciklusi koji se dobivaju iz tekućeg izbacivanjem dvaju ne-susjednih bridova i umetanjem odgovarajućih "unakrsnih" bridova - vidi sliku 6.14. U 2-okolini pronalazi se ciklus s najmanjom cijenom. Ako taj ciklus ima manju cijenu od tekućeg, on postaje novi tekući i algoritam se nastavlja. Inače je algoritam završen i on kao rješenje daje zadnji tekući ciklus (za njega kažemo da je 2-optimalan).



Slika 6.14: Generiranje 2-okoline zadanog Hamiltonovog ciklusa.

Primijetimo da u 2-okolini tekućeg ciklusa postoji $n(n-1)/2 - n = n(n-3)/2$ ciklusa, gdje je n broj vrhova u grafu. Naime, dva od n bridova duž ciklusa možemo izabrati na $n(n-1)/2$ načina, no n takvih izbora ne vrijedi jer su bridovi susjedni. Dakle, jedna iteracija 2-opt algoritma izvršava se u vremenu $\mathcal{O}(n^2)$. Broj iteracija i točnost rješenja teško je predvidjeti, no očito je da obje stvari bitno ovise o izabranom polaznom rješenju.

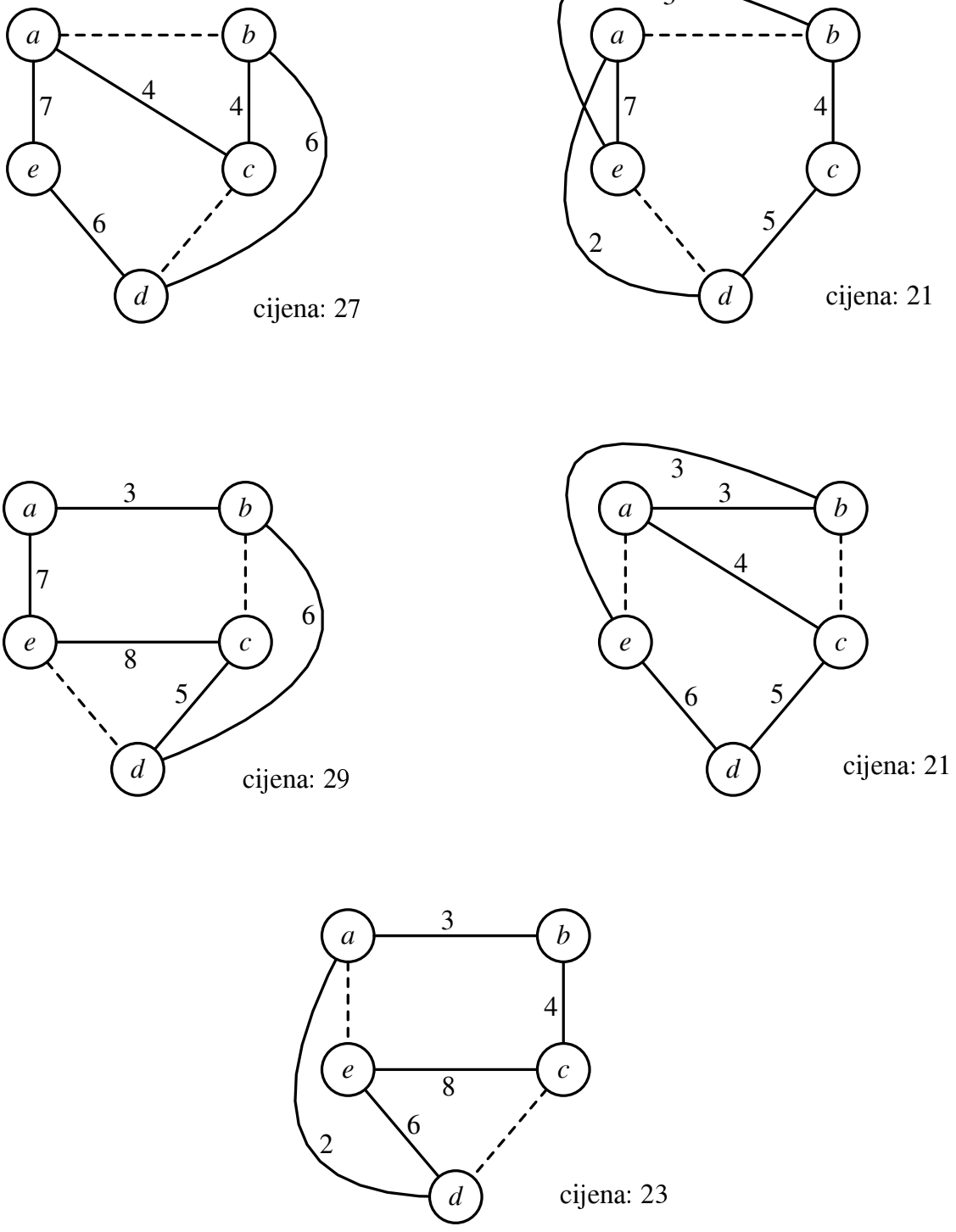
Kao konkretni primjer rada 2-opt algoritma, ponovo promatramo primjerak problema trgovačkog putnika sa slike 6.11.

- Za polazno rješenje biramo Hamiltonov ciklus (a, b, c, d, e, a) sa cijenom $3 + 4 + 5 + 6 + 7 = 25$. 2-okolina tog rješenja sadrži $n(n-3)/2 = 5$ ciklusa čiji izgled i cijene se vide na slici 6.15.

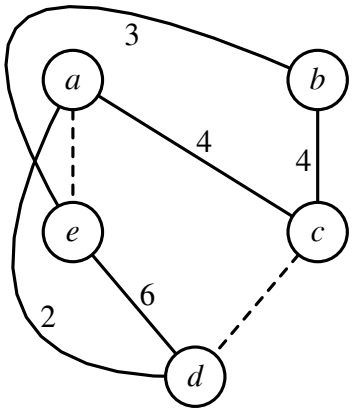
- Novo tekuće rješenje je npr. (a, d, c, b, e, a) sa cijenom 21. Algoritam kreće u drugu iteraciju. U okolini novog tekućeg ciklusa nalazi se optimalno rješenje (a, c, b, e, d, a) sa cijenom 19, prikazano na slici 6.16.

- Dakle, u drugoj iteraciji dobivamo optimalno rješenje kao treće tekuće. Algoritam će završiti rad nakon treće iteracije, jer u toj trećoj iteraciji neće naći bolje rješenje.

Algoritam 2-opt može se realizirati sljedećom C funkcijom `TwoOpt( )`, gdje je `SIZE` pogodna globalna konstanta. Funkcija je samo skicirana, dakle neki njezini dijelovi su opisani riječima te bi ih trebalo razraditi do kraja.



Slika 6.15: 2-okolina Hamiltonovog ciklusa (a, b, c, d, e, a) za graf sa slike 6.11.



Slika 6.16: Član 2-okoline Hamiltonovog ciklusa (a, d, c, b, e, a) za graf sa slike 6.11.

```
void TwoOpt (float cost[ ][SIZE], int n, int u[ ]) {
  /* Promatramo potpuni graf s vrhovima 0, 1, ...  , n-1.  */
  /* Element cost[i][j] sadrži cijenu brida između vrhova i, j.  */
  /* Pronalazi se 2-optimalni Hamiltonov ciklus te se on zapisuje */
  /* kao niz vrhova u elemente u[0],u[1],...,u[n-1].  Polje u[] je */
  /* inicijalizirano tako da prikazuje početni Hamiltonov ciklus.  */
  /* Tekući ciklus sastoji se od bridova (u[0],u[1]), (u[1],u[2]), */
  /* ..., (u[n-2],u[n-1]), (u[n-1],u[0]).  Svi u[i] su različiti.  */
  float d, dmax;
  int imax, jmax, i, j;
  do {
    dmax = 0;
    for (i=0; i<=n-3; i++)
      for (j=(i+2); j<=(i==0?n-2:n-1); j++) {
        d = cost[u[i]][u[(i+1)%n]] + cost[u[j]][u[(j+1)%n]]
           - cost[u[i]][u[j]] - cost[u[(i+1)%n]][u[(j+1)%n]];
        if (d > dmax ) {
          dmax = d;
          imax = i;
          jmax = j;
        }
      }
    if (dmax > 0)
      ...  u tekućem ciklusu zamijeni bridove
      (u[imax],u[(imax+1)%n]), (u[jmax],u[(jmax+1)%n]) s bridovima
      (u[imax]],u[jmax]), (u[(imax+1)%n],u[(jmax+1)%n]].  Dakle
      preuredi polje u[] na odgovarajući način ....
  } while (dmax > 0);
}
```

Algoritam 2-opt najjednostavniji je član porodice r-opt algoritama za $r \geq 2$. Općenito, r-opt algoritam pretražuje r-okolinu tekućeg Hamiltonovog ciklusa. r-okolina se sastoji od svih Hamiltonovih ciklusa koji se u r od n bridova razlikuju od tekućeg. Jedna iteracija r-opt algoritma zahtijeva vrijeme $\mathcal{O}(n^r)$. Znači, povećanjem r dobivamo sve točniji no vremenski sve zahtjevniji algoritam. U praksi se najčešće koriste: brzi 2-opt i precizniji 3-opt algoritam. Eksperimenti pokazuju da se 4-opt algoritam već ne isplati: naime on je znatno sporiji no ne bitno točniji od 3-opt.

6.5.3 Traženje optimalnog linearnog razmještaja

Kao drugi primjer algoritma zasnovanog na lokalnom traženju izložit ćemo algoritam za traženje **optimalnog linearnog razmještaja** (*optimal one-dimensional placement*). Problem glasi ovako. Zadan je neusmjereni graf G, čiji skup vrhova je označen s V, a skup bridova s E. Vrhovi iz V zovu se "komponente". Svakom bridu $(u, v) \in E$ pridružena je težina $w(u, v)$ koja se tumači kao "broj žica" između komponenti u i v. Htjeli bi pronaći pogodnu numeraciju $v_1, v_2, \ldots, v_n$ vrhova takvu da $\sum_{(v_i,v_j)\in E} |i - j| \cdot w(v_i, v_j)$ bude minimalna. Drugim riječima, htjeli bi pronaći razmještaj komponenti u linearno organiziranom ormaru (sastavljenom od ladica 1, 2, ..., n) tako da ukupna duljina potrebnih žica bude minimalna.

Algoritam lokalnog traženja za problem linearnog razmještaja kreće od proizvoljnog linearnog razmještaja kao od polaznog rješenja. Okolina tekućeg rješenja istražuje se tako da se na tekući razmještaj primjenjuju transformacije sljedećeg oblika:

1. Zamijeni susjedne komponente v_i i v_{i+1}. Neka je $L(j)$ zbroj težina bridova koji idu nalijevo od v_j, to jest $L(j) = \sum_{k=1}^{j-1} w(v_k, v_j)$. Slično, neka je $R(j) = \sum_{k=j+1}^{n} w(v_k, v_j)$. Zamjena susjednih komponenti donosi smanjenje težine ukoliko je $L(i) - R(i) + R(i + 1) - L(i + 1) + 2 \cdot w(v_i, v_{i+1})$ negativno.

2. Izvadi komponentu v_i i umetni je između v_j i v_{j+1} za neke i i j.

3. Zamijeni bilo koje dvije komponente v_i i v_j.

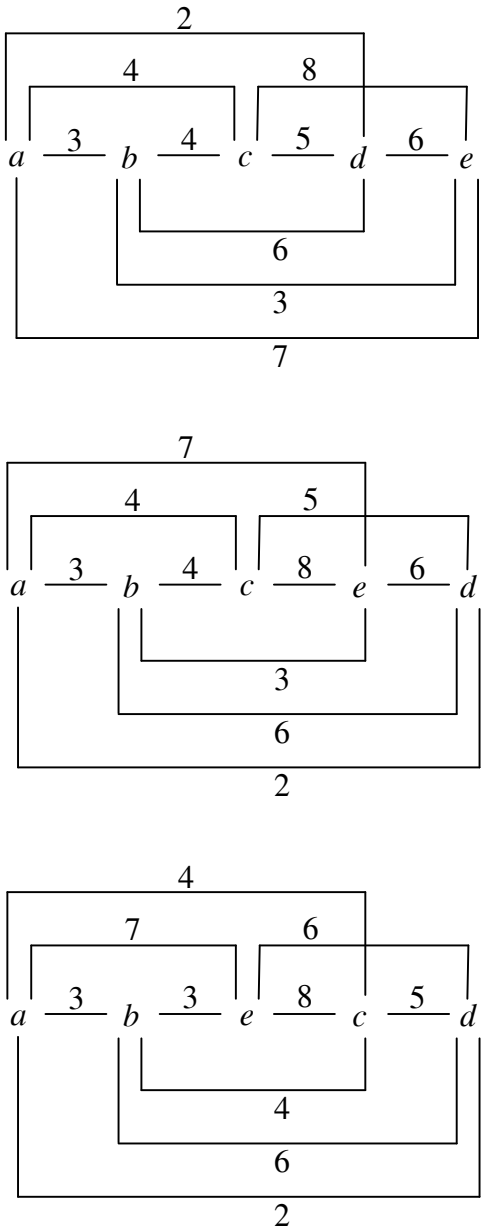
U svrhu konkretnog primjera, pretpostavimo da graf sa slike 6.11, koji je prije predstavljao primjerak problema trgovačkog putnika, sada predstavlja primjerak problema linearnog razmještaja. Primjenjujemo algoritam lokalnog traženja, s time da se ograničavamo na jednostavne transformacije oblika 1.

- Polazni razmještaj (a, b, c, d, e) ima težinu 97 i prikazan je na slici 6.17 gore.

- Razmatramo zamjenu d i e. Imamo $L(d) = 13$, $R(d) = 6$, $L(e) = 24$, $R(e) = 0$. Dakle $L(d) - R(d) + R(e) - L(e) + 2 \cdot w(d, e) = -5$. Mijenjamo d i e pa dobivamo bolji razmještaj (a, b, c, e, d) s težinom 92 prikazan na sredini slike 6.17.

- Razmještaj sa sredine slike 6.17 možemo još poboljšati zamjenom c s e, čime odobivamo razmještaj (a, b, e, c, d) s težinom 91 prikazan na slici 6.17 dolje.

- Pokazuje se da je taj zadnji razmještaj lokalno optimalan za skup transformacija oblika 1. Ipak, on nije globalno optimalan jer npr. razmještaj (a, c, e, d, b) ima težinu 84.

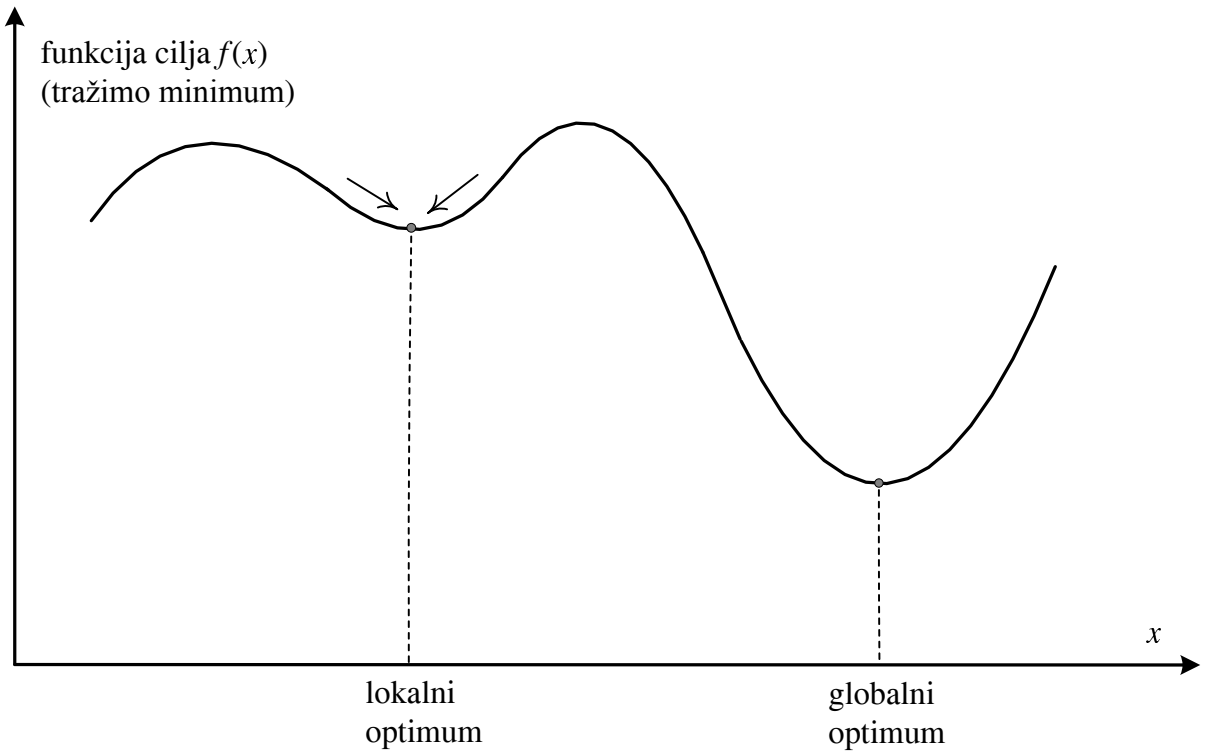


Slika 6.17: Tri dopustiva rješenja istog primjerka problema linearnog razmještaja.

Analiziramo vremensku složenost jedne iteracije opisanog algoritma. Ako se ograničimo na transformacije oblika 1, tada postoji samo $n-1$ transformacija koje treba razmotriti. Već izračunate $L(i)$ i $R(i)$ treba ažurirati samo onda kad se v_i zamijeni s v_{i-1} ili v_{i+1}. Ažuriranje je jednostavno: na primjer kod zamjene v_i s v_{i+1} novi $L(i)$ odnosno $R(i)$ je $L(i+1) - w(v_i, v_{i+1})$ odnosno $R(i+1) + w(v_i, v_{i+1})$. Dakle testiranje svih $n-1$ transformacija te ažuriranje $L(i)$-ova i $R(i)$-ova za odabranu transformaciju zahtijeva vrijeme $\mathcal{O}(n)$. To je i sve što moramo raditi u jednoj iteraciji. Inicijalizacija $L(i)$-ova i $R(i)$-ova na početku algoritma može se obaviti u vremenu $\mathcal{O}(n^2)$.

6.5.4 Složeniji oblici lokalnog traženja

Na kraju ovog odjeljka, primijetimo da je osnovna manjkavost lokalnog traženja u tome što ono može "zaglaviti" u lokalnom optimumu koji se po vrijednosti funkcije cilja jako razlikuje od pravog (globalnog) optimuma. Manjkavost je ilustrirana slikom 6.18.



Slika 6.18: Manjkavost lokalnog traženja - zaglavljivanje u lokalnom optimumu.

Novije varijante strategije lokalnog traženja, poput ***tabu search*** ili ***simulated annealing***, sadrže dodatna pravila kojima se nastoji pobjeći iz lokalnog optimuma. **Evolucijski (genetički) algoritmi** mogu se interpretirati kao multiplicirana varijanta lokalnog traženja, gdje se umjesto jednog rješenja pamti cijela "populacija" rješenja, a kao okolina te populacije promatraju se rješenja koja se mogu dobiti kombiniranjem ili mutiranjem rješenja iz populacije. Velik broj tekućih rješenja kod evolucijskog algoritma smanjuje opasnost zaglavljivanja u lokalnom optimumu.

Literatura

[1] Aho A.V., Hopcroft J.E., Ulman J.D., *Data Structures and Algorithms*, 2nd edition. Addison-Wesley, Reading MA, 1987. ISBN-13: 978-0201000238.

[2] Drozdek A., *Data Structures and Algorithms in C++*, 4th edition. Cengage Learning, Boston MA, 2012. ISBN-13: 978-1133608424

[3] Gilberg R.F., Forouzan B.A., *Data Structures: A Pseudocode Approach with C*, 2nd edition. Cengage Learning, Boston MA, 2004. ISBN-13: 978-0534390808.

[4] Goodrich M.T., Tamassia R., *Algorithm Design - Foundations, Analysis, and Internet Examples*. John Wiley & Sons, New York NY, 2002. ISBN-13: 978-0471383659.

[5] Goodrich M.T., Tamassia R., Mount D., *Data Structures and Algorithms in C++*, 2nd edition. John Wiley & Sons, New York NY, 2011. ISBN-13: 978-0470383278.

[6] Horowitz E., Sahni S., Mehta D., *Fundamentals of Data Structures in C++*, 2nd edition. Silicon Press, Summit NJ, 2006. ISBN-13: 978-0929306377.

[7] Horowitz E., Sahni S., Rajasekaran S., *Computer Algorithms / C++*, 2nd edition. Silicon Press, Summit NJ, 2008. ISBN-13: 978-0929306421.

[8] Hubbard J.R., *Schaum's Outline of Data Structures with C++*. McGraw-Hill, New York NY, 2000. ISBN-13: 978-0071353458.

[9] Kernighan B.W., Ritchie D.M., *C Programming Language*, 2nd edition. Prentice Hall, Englewood Cliffs NJ, 1988. ISBN-13: 978-0131103627.

[10] King K.N., *C Programming: A Modern Approach*, 2nd edition. W.W. Norton & Company, New York NY, 2008. ISBN-13: 978-0393979503.

[11] Kruse R.L., Leung B.P., Tondo, C.L., *Data Structures and Program Design in C*, 2nd edition. Prentice Hall, Englewood Cliffs NJ, 1996. ISBN-13: 978-0132883665.

[12] Loudon K., *Mastering Algorithms with C*. O'Reilly, Sebastopol CA, 1999. ISBN-13: 978-1565924536.

[13] McAllister W., *Data Structures and Algorithms Using Java*. Jones & Bartlett Publishers, Sudbury MA, 2008. ISBN-13: 978-0763757564.

[14] McMillan M., *Data Structures and Algorithms Using C#*. Cambridge University Press, Cambridge, 2007. ISBN-13: 978-0521670159.

[15] Mehlhorn K., Sanders P., *Algorithms and Data Structures: The Basic Toolbox*. Springer Verlag, Berlin Heidelberg, 2010. ISBN-13: 978-3642096822.

[16] Prata S., *C Primer Plus*, 5th edition. Sams Publishing, Indianapolis IN, 2005. ISBN-13: 978-0672326967.

[17] Preiss B.R., *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, New York NY, 1999. ISBN-13: 978-0471241348.

[18] Radhakrishnan S., Wise L., Sekharan C.N., *Data Structures Featuring C++: A Programmer's Perspective*. SRR LLC, 2013. ISBN-13: 978-0989095907

[19] Sahni S., *Data Structures, Algorithms, and Applications in C++*, 2nd edition. Silicon Press, Summit NJ, 2004. ISBN-13: 978-0929306322.

[20] Sedgewick R., *Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching nad Graph Algorithms*, 3rd edition. Addison-Wesley, Reading MA, 2001. ISBN-13: 978-0201756081.

[21] Shaffer C.A., *Data Structures and Algorithm Analysis in C++*, 3rd edition. Dover Publications, Mineola NY, 2011. ISBN-13: 978-0486485829.

[22] Weiss M.A., *Data Structures and Algorithm Analysis in C++*, 4th edition. Prentice Hall, Englewood Cliffs NJ, 2013. ISBN-13: 978-0132847377.