

Sveučilište u Zagrebu
Prirodoslovno Matematički Fakultet
- Matematički odjel

Robert Manger, Miljenko Marušić

STRUKTURE PODATAKA I ALGORITMI

skripta

Drugo izdanje
(s primjerima u programskom jeziku C)

Zagreb, rujan 2003.

Sadržaj

1	UVOD	3
1.1	Osnovni pojmovi	3
1.2	Elementi od kojih se grade strukture podataka	5
1.3	Zapisivanje i analiziranje algoritma	6
2	LISTE	7
2.1	(Općenita) lista	7
2.1.1	Implementacija liste pomoću polja	8
2.1.2	Implementacija liste pomoću pointera	10
2.2	Stog (stack)	11
2.2.1	Implementacija stoga pomoću polja	13
2.2.2	Implementacija stoga pomoću pointera	14
2.3	Red (queue)	15
2.3.1	Implementacija reda pomoću cirkularnog polja	16
2.3.2	Implementacija reda pomoću pointera	17
3	STABLA	19
3.1	(Uređeno) stablo	19
3.1.1	Obilazak stabla	22
3.1.2	Implementacija stabla na osnovu veze “čvor \rightarrow roditelj”	23
3.1.3	Implementacija stabla na osnovu veze “čvor \rightarrow (prvo dijete, idući brat)”	24
3.2	Binarno stablo	26
3.2.1	Implementacija binarnog stabla pomoću pointera	28
3.2.2	Implementacija potpunog binarnog stabla pomoću polja	29
4	SKUPOVI	31
4.1	(Općeniti) skup	31
4.1.1	Implementacija skupa pomoću bit-vektora	32
4.1.2	Implementacija skupa pomoću sortirane vezane liste	32
4.2	Rječnik	33
4.2.1	Implementacija rječnika pomoću bit-vektora	33
4.2.2	Implementacija rječnika pomoću liste	33
4.2.3	Implementacija rječnika pomoću rasute (hash) tablice	35
4.2.4	Implementacija rječnika pomoću binarnog stabla traženja	38
4.3	Prioritetni red	42
4.3.1	Implementacija prioritetnog reda pomoću sortirane vezane liste	43
4.3.2	Implementacija prioritetnog reda pomoću binarnog stabla traženja	43
4.3.3	Implementacija prioritetnog reda pomoću hrpe	43
4.4	Preslikavanje i relacija	47
4.4.1	Implementacija preslikavanja pomoću hash tablice ili binarnog stabla	48
4.4.2	Implementacija relacije pomoću bit-matrice	50
4.4.3	Implementacija relacije pomoću multi-liste	50

5	OBLIKOVANJE ALGORITAMA	53
5.1	Metoda “podijeli pa vladaj”	53
5.1.1	Sortiranje sažimanjem (merge sort)	53
5.1.2	Traženje elementa u listi	54
5.1.3	Množenje dugačkih cijelih brojeva	54
5.2	Dinamičko programiranje	56
5.2.1	Problem određivanja šanse za pobjedu u sportskom nadmetanju	56
5.2.2	Rješavanje 0/1 problema ranca	57
5.3	“Pohlepni” pristup	58
5.3.1	Optimalni plan sažimanja sortiranih listi	59
5.3.2	Kontinuirani problem ranca	59
5.4	Backtracking	61
5.4.1	Problem n kraljica	62
5.4.2	Problem trgovačkog putnika	64

1

UVOD

1.1 Osnovni pojmovi

U računarstvu se susrećemo s dva osnovna pojma:

strukture podataka ... “statički” aspekt nekog programa. Ono sa čime se radi.

algoritmi ... “dinamički” aspekt programa. Ono što se radi.

Strukture podataka i algoritmi su u nerazlučivoj vezi: nemoguće je govoriti o jednom a da se ne spomene drugo. U ovom kolegiju proučavat ćemo baš tu vezu: 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 i pojmova, koji čine osnove računarstva.

Uz “strukture podataka” i “algoritme”, ovaj kolegij koristi još nekoliko naizgled sličnih pojmova. Objasnit ćemo njihovo značenje i međusobni odnos.

tip podataka ... skup vrijednosti koje neki podatak može poprimiti (npr. podatak tipa `int` može imati samo vrijednosti iz skupa cijelih brojeva prikazivih u stroju).

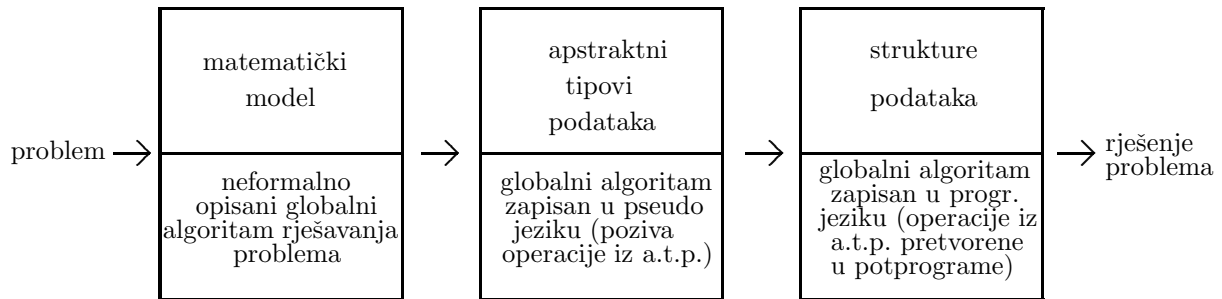
apstraktni tip podataka (a.t.p.) ... zadaje se navođenjem jednog ili više tipova podataka, te jedne ili više operacija (funkcija). Operandi i rezultati navedenih operacija su podaci navedenih tipova. Među tipovima postoji jedan istaknuti po kojem cijeli apstraktni tip podataka dobiva ime.

struktura podataka ... skupina varijabli u nekom programu i veza među tim varijablama.

algoritam ... konačni niz instrukcija, od kojih svaka ima jasno značenje i može biti izvršena u konačnom vremenu. Iste instrukcije mogu se izvršiti više puta, pod pretpostavkom da same instrukcije ukazuju na ponavljanje. Ipak, zahtijevamo da, za bilo koje vrijednosti ulaznih podataka, algoritam završava nakon konačnog broja ponavljanja.

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.

Svako od potpoglavlja 2.1–4.4 obrađuje jedan apstraktni tip podataka. Promatraju se razne implementacije za taj apstraktni tip podataka te njihove prednosti i mane. Na taj način upoznat ćemo mnogo apstraktnih tipova podataka, te još više struktura podataka i algoritama. Preostala potpoglavlja 5.1–5.4 posvećena su općenitim metodama (strategijama) koje mogu služiti za oblikovanje složenijih algoritama. Znanje iz ovog kolegija trebalo bi nam omogućiti da bolje programiramo. Naime, razvoj programa (metodom postepenog profinjavanja) može se promatrati u skladu sa sljedećim dijagramom.



Slika 1.1 Postupak rješavanja problema.

Iz ovog dijagrama vidi se uloga apstraktnih tipova podataka, struktura podataka i algoritama u postupku rješavanja problema. Također se vidi da se programiranje u podjednako mjeri sastoji od razvijanja struktura podataka kao i od razvijanja algoritama.

Slijedi primjer za apstraktni tip podataka. Definiramo apstraktni tip podataka koji odgovara matematičkom pojmu kompleksnih brojeva:

Apstraktni tip podataka COMPLEX

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

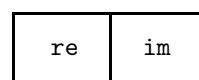
COMPLEX ... podaci ovog tipa su uređeni parovi podataka tipa **scalar**.

ADD(z1,z2,&z3) ... za zadane **z1,z2** tipa **COMPLEX** računa se njihov zbroj **z3**, također tipa **COMPLEX**. Dakle za **z1** oblika (x_1, y_1) , **z2** oblika (x_2, y_2) , dobiva se **z3** oblika (x_3, y_3) , tako da bude $x_3 = x_1 + x_2$, $y_3 = y_1 + y_2$.

MULT(z1,z2,&z3) ... za zadane **z1,z2** tipa **COMPLEX** računa se njihov umnožak **z3**, također tipa **COMPLEX**. Dakle za **z1** oblika (x_1, y_1) , **z2** oblika (x_2, y_2) , dobiva se **z3** oblika (x_3, y_3) , takav da je $x_3 = x_1 * x_2 - y_1 * y_2$, $y_3 = x_1 * y_2 + y_1 * x_2$.

Struktura podataka pogodna za prikaz kompleksnog broja bila bi tipa:

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



Implementacija apstraktnog tipa podataka **COMPLEX** se sastoji od prethodne definicije tipa, te od funkcija oblika:

```
void ADD (COMPLEX z1, COMPLEX z2, COMPLEX *z3) {...}
```

```
void MULT (COMPLEX z1, COMPLEX z2, COMPLEX *z3) {...}
```

(algoritmi upotrijebljeni u potprogramima su trivijalni).

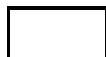
Za apstraktni tip podataka **COMPLEX** je važno da su prisutne operacije **ADD()** i **MULT()**. Bez tih operacija radilo bi se o tipu, i tada kompleksne brojeve ne bismo mogli razlikovati od uređenih parova skalara. Dodatna dimenzija "apstraktnosti" sastoji se u tome što nismo do kraja odredili što je tip **scalar**. To je naime nebitno za proučavanje struktura podataka i algoritama (važan je odnos među varijablama a ne njihov sadržaj).

Na vježbama će biti izložen cjelovit primjer za rješavanje problema metodom postepenog profinjavanja.

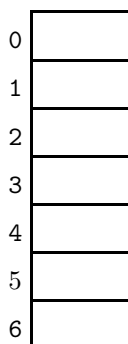
1.2 Elementi od kojih se grade strukture podataka

Struktura podataka se sastoji od manjih cjelina koje se udružuju u veće i međusobno povezuju vezama. Uvodimo posebne nazive za cjeline, načine udruživanja i načine povezivanja. Također, uvodimo pravila kako se strukture prikazuju dijagramima.

Ćelija ... varijabla koju promatramo kao zasebnu cjelinu. To je relativan pojam (nešto se u jednom trenutku može smatrati ćelijom, a kasnije se može gledati unutrašnja građa iste cjeline). Svaka ćelija ima svoj tip i adresu.



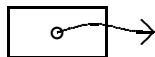
Polje ... (**array** u C-u). Mehanizam udruživanja manjih dijelova strukture u veće. Polje čini više ćelija istog tipa pohranjenih na uzastopnim adresama. Broj ćelija je unaprijed zadan i nepromjenljiv. Jedna ćelija se zove element polja i jednoznačno je određena pripadnom vrijednošću indeksa. Po ugledu na C, uzimamo da su indeksi 0, 1, 2, ..., N-1, gdje je N cjelobrojna konstanta.



Zapis ... (slog, **structure** u C-u). Također mehanizam udruživanja manjih dijelova strukture u veće. Zapis čini više ćelija, koje ne moraju biti istog tipa, no koje su pohranjene na uzastopnim adresama. Broj, redosljed i tip ćelija je unaprijed zadan i nepromjenljiv. Pojedina ćelija se zove komponenta zapisa. 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.



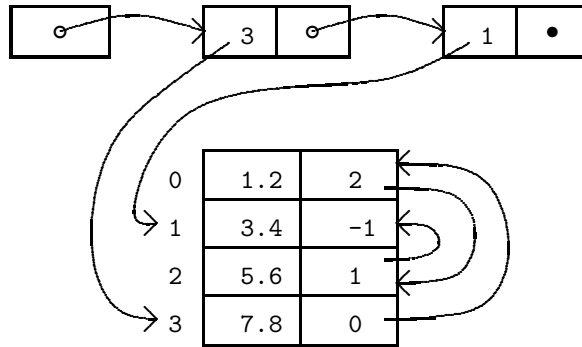
Pointer ... služi za uspostavljanje veze između dijelova strukture. Pointer je ćelija koja pokazuje neku drugu ćeliju. Sadržaj pointera je adresa ćelije koju treba pokazati.



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



Dijagram pokazuje primjer strukture podataka koja se sastoji od niza zapisa povezanih pomoću pointera, te od jednog polja zapisa. Zapisi su još međusobno povezani kursorima.



Slika 1.2 Primjer strukture podataka.

1.3 Zapisivanje i analiziranje algoritma

Kao jezik za zapisivanje algoritama služiti će nam programski jezik C. Dakle, umjesto algoritama pisati ćemo program (ili funkciju) u C-u koji radi po tom algoritmu. Obično će biti riječ o “skiciranom” nedovršenom kodu, gdje su neki nizovi naredbi zamijenjeni slobodnim tekstom.

Pod **analizom algoritma** podrazumijevamo procjenu vremena izvršavanja tog algoritma. Vrijeme poistovjećujemo s brojem operacija koje odgovarajući program treba obaviti, i izražavamo kao funkciju oblika $T(n)$, gdje je n neka mjera za veličinu skupa ulaznih podataka. Naprimjer, ako promatramo algoritam koji sortira niz realnih brojeva, tada njegovo vrijeme izvršavanja izražavamo u obliku $T(n)$ gdje je n duljina niza realnih brojeva. Slično, ako promatramo algoritam za invertiranje matrice, tada njegovo vrijeme izvršavanja izražavamo kao $T(n)$, gdje je n red matrice.

Često se dešava da $T(n)$ ne ovisi samo o veličini skupa ulaznih podataka n , nego također i o vrijednostima tih podataka. Npr. algoritam za sortiranje možda brže sortira niz brojeva koji je “skoro sortirani”, a sporije niz koji je “jako izmiješan”. Tada definiramo $T(n)$ kao vrijeme u najgorem slučaju, dakle kao maksimum za vrijeme izvršavanja po svim skupovima ulaznih podataka veličine n . Također se tada promatra vrijeme u prosječnom slučaju $T_{avg}(n)$, koje se dobiva kao matematičko očekivanje vremena izvršavanja. Da bi mogli govoriti o očekivanju, moramo pretpostaviti distribuciju za razne skupove ulaznih podataka veličine n . Obično se pretpostavlja uniformna distribucija, dakle smatra se da su svi skupovi ulaznih podataka jednako vjerojatni.

Funkciju $T(n)$ nema smisla precizno određivati, dovoljno je utvrditi njen red veličine. Npr. vrijeme izvršavanja Gaussovog algoritma za invertiranje matrice je $\mathcal{O}(n^3)$. Time smo zapravo rekli da $T(n) \leq \text{const} \cdot n^3$. Nismo precizno utvrdili koliko će sekundi trajati naš program. Jedino što znamo je sljedeće:

- ako se red matrice udvostruči, invertiranje bi moglo trajati i do 8 puta dulje;
- ako se red matrice utrostruči, invertiranje traje 27 puta dulje, itd.

2

LISTE

2.1 (Općenita) lista

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

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

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; identitet elementa određen je njegovom **pozicijom** (rednim brojem) a ne njegovom vrijednošću.

Važno svojstvo liste je da su njeni 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.

U nastavku navodimo nekoliko primjera za liste.

- Riječ u nekom tekstu je lista znakova. Npr. riječ ZNANOST se može interpretirati kao (Z, N, A, N, O, S, T). Primijetimo da dva različita elementa imaju istu vrijednost. Slično, redak u tekstu je lista znakova (uključujući i bjeline). Cijeli tekst je lista redaka.
- Polinom u matematici bilježimo na sljedeći način:

$$P(x) = a_1x^{e_1} + a_2x^{e_2} + \dots + a_nx^{e_n},$$

gdje je $0 \leq e_1 < e_2 < \dots < e_n$. Zapravo se radi o listi oblika :

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

- U nekim programskim jezicima lista je osnovni objekt od kojeg se grade svi ostali. Primjer: LISP (List Processing).
- Kao rezultat pretraživanja baze podataka obično se dobiva lista zapisa. Npr. postavljamo upit: ispisati prezimena i imena ljudi koji su rođeni 1960. godine a po zanimanju su matematičari, sortirano po abecedi. Rezultat je oblika: (Babić Mato, Marković Pero, Petrović Marija, ...).

Da bi matematički pojam liste pretvorili u apstraktni tip podataka, trebamo definirati operacije koje se obavljaju na listama. To se može učiniti na razne načine. Naš a.t.p. je samo jedna od mogućih varijanti.

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, \dots, a_n) definirane pozicije koje odgovaraju prvom, drugom, ..., n -tom elementu, a također i pozicija na kraju liste (neposredno iza n -tog elementa).

END(L) ... funkcija koja vraća poziciju na kraju liste L.

MAKE_NULL(&L) funkcija pretvara listu L u praznu listu, i vraća poziciju **END(L)**.

INSERT(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, \dots, a_n) , tada L postaje $(a_1, a_2, \dots, a_{p-1}, x, a_p, \dots, a_n)$. Ako je $p == \text{END}(L)$ tada L postaje $(a_1, a_2, \dots, a_n, x)$. Ako u L ne postoji pozicija p, rezultat je nedefiniran.

DELETE(p, &L) ... funkcija izbacuje element na poziciji p iz liste L. Dakle, ukoliko je lista L oblika (a_1, a_2, \dots, a_n) tada L postaje $(a_1, a_2, \dots, a_{p-1}, a_{p+1}, \dots, a_n)$. Rezultat nije definiran ako L nema pozicije p ili ako je $p == \text{END}(L)$.

FIRST(L) ... funkcija vraća prvu poziciju u listi L. Ako je L prazna, vraća se **END(L)**.

NEXT(p, L), **PREVIOUS(p, L)** ... funkcije koje vraćaju poziciju iza odnosno ispred p u listi L. Ako je p zadnja pozicija u L, tada je **NEXT(p, L) == END(L)**. **NEXT()** je nedefinirana za $p == \text{END}(L)$. **PREVIOUS()** je nedefinirana za $p == \text{FIRST}(L)$. Obje funkcije su nedefinirane ako L nema pozicije p.

RETRIEVE(p, L) ... funkcija vraća element na poziciji p u listi L. Rezultat je nedefiniran ako je $p == \text{END}(L)$ ili ako L nema pozicije p.

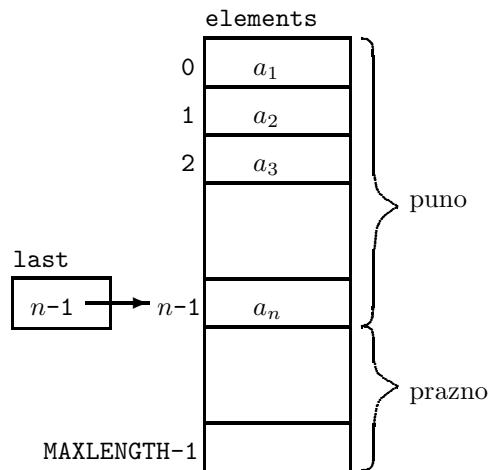
U nastavku ćemo opisati neke strukture podataka koje mogu služiti za prikaz liste. Postoje dva osnovna pristupa:

- “logički” redoslijed elemenata u listi poklapa se s “fizičkim” redoslijedom tih elemenata u memoriji. (Koristimo polje.)
- “logički” redoslijed se ne poklapa s “fizičkim”, pa se mora eksplicitno zapisati. (Služimo se pointerom ili kursorom.)

Oba pristupa dozvoljavaju razne varijante. Mi ćemo od svakog pristupa opisati samo jednu (najtípičniju) varijantu.

2.1.1 Implementacija liste pomoću polja

Elementi liste su spremljeni u uzastopnim ćelijama jednog polja. Također imamo jedan kursor koji pokazuje gdje se zadnji element liste nalazi u polju. 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. Duljina liste je ograničena.



Slika 2.1 Implementacija liste pomoću polja.

Implementacija liste pomoću polja može se precizno izraziti u C-u.

```

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

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

typedef int position;

position END (LIST L) {
    return (L.last + 1);
}

position MAKE_NULL (LIST *L_ptr) {
    L_ptr->last = -1;
    return 0;
}

void INSERT (elementtype x, position p, LIST *L_ptr) {
    position q;
    if (L_ptr->last >= MAXLENGTH-1)
        error("list is full"); /* zaustavlja rad i ispisuje poruku */
    else if ( (p > L_ptr->last+1) || (p < 0) )
        error("position does not exist");
    else {
        for (q = L_ptr->last; q >= p; q--)
            /* pomakni elemente na pozicijama p, p+1,...jedno mjesto dalje */
            L_ptr->elements[q+1] = L_ptr->elements[q];
        L_ptr->last++;
        L_ptr->elements[p] = x;
    }
}

void DELETE (position p, LIST *L_ptr) {
    position q;
    if ( (p > L_ptr->last) || (p < 0) )
        error("position does not exist");
    else {
        L_ptr->last--;
        for (q = p; q <= L_ptr->last; q++)
            /* pomakni elemente na pozicijama p+1, p+2,...jedno mjesto natrag */
            L_ptr->elements[q] = L_ptr->elements[q+1];
    }
}

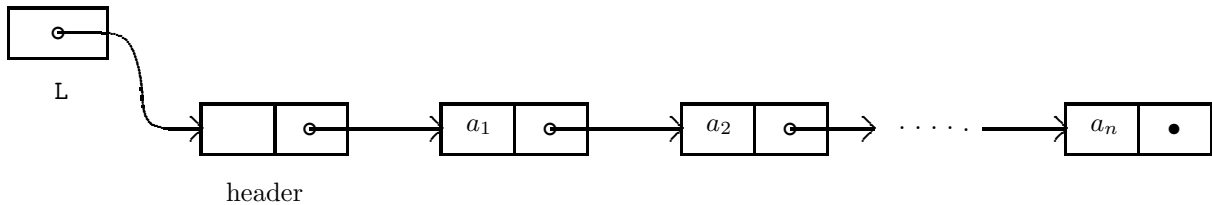
```

Ostali potprogrami su vrlo jednostavni. `FIRST(L)` uvijek vraća 0, `NEXT(p,L)` odnosno `PREVIOUS(p,L)` vraćaju `p+1` odnosno `p-1`, `RETRIEVE(p,L)` vraća `L.elements[p]`. Kontrolira se da li su parametri u dozvoljenom rasponu. Vrijeme izvršavanja za `INSERT()` i `DELETE()` je $\mathcal{O}(n)$, a za ostale operacije je $\mathcal{O}(1)$. Implementacija liste pomoću polja se smatra pogodnom pod sljedećim uvjetima:

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

2.1.2 Implementacija liste pomoću pointera

Lista se prikazuje nizom ćelija. Svaka ćelija sadrži jedan element liste i pointer na istu takvu ćeliju koja sadrži idući element liste. Postoji i polazna ćelija (header) koja označava početak liste i ne sadrži nikakav element. Ovakva struktura se obično zove vezana lista.



Slika 2.2 Implementacija liste pomoću pointera.

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 je teže odrediti kraj liste ili prethodni element. Listu poistovjećujemo s pointerom na header.

Pozicija elementa a_i je pointer na ćeliju koja sadrži pointer na a_i . Znači, pozicija od a_1 je pointer na header, pozicija od a_2 je pointer na ćeliju u kojoj je zapisan a_1 , itd. Pozicija END(L) je pointer na zadnju ćeliju u vezanoj listi. Ova pomalo neprirodna definicija se uvodi zato da bi se efikasnije obavljale operacije INSERT() i DELETE(). Implementacija se ovako može opisati u C-u:

```

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

typedef celltype *LIST;

typedef celltype *position;

position END (LIST L) {
/* vraća pointer na zadnju ćeliju u L */
    position q;
    q = L;
    while (q->next != NULL)
        q = q->next;
    return q;
}

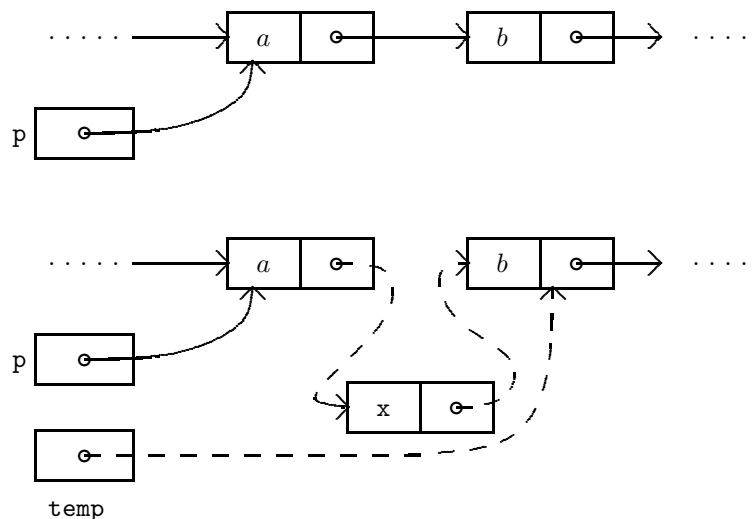
position MAKE_NULL (LIST *Lptr) {
    *Lptr = (celltype*) malloc(sizeof(celltype));
    (*Lptr)->next = NULL;
    return(*Lptr);
}

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

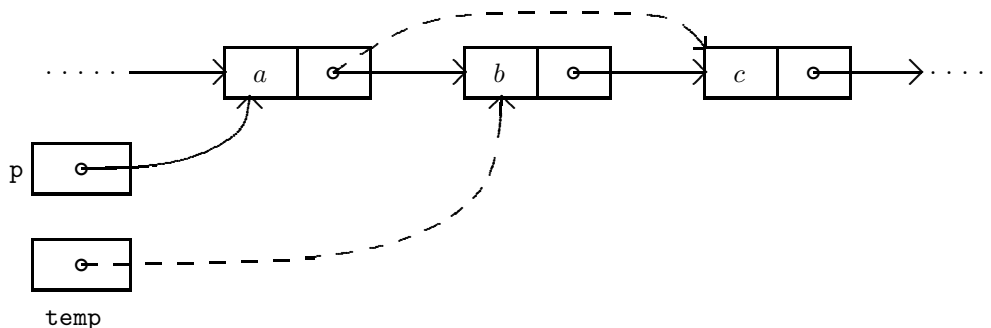
```

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

```



Slika 2.3 Dijagram za INSERT().



Slika 2.4 Dijagram za DELETE().

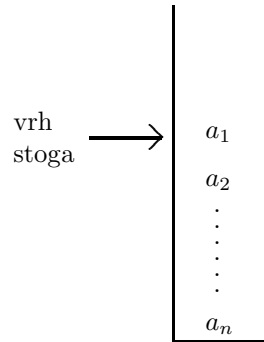
Potprogrami `FIRST()`, `NEXT()`, i `RETRIEVE()` su vrlo jednostavni (svode se na jednu naredbu za pridruživanje). `PREVIOUS()` se ne da lijepo implementirati: jedini način je da prođemo cijelom listom od početka do zadane pozicije. Lako se vidi da je vrijeme izvršavanja za `END()` i `PREVIOUS()` $\mathcal{O}(n)$, a za sve ostale operacije $\mathcal{O}(1)$.

Implementacija liste pomoću pointera smatra se pogodnom pod sljedećim uvjetima:

- ima mnogo ubacivanja/izbacivanja u listu
- duljina liste jako varira
- nije nam potrebna operacija `PREVIOUS()`.

2.2 Stog (stack)

Stog je specijalna vrsta liste u kojoj se sva ubacivanja i izbacivanja obavljaju na jednom kraju koji se zove **vrh**. Stog se također naziva i stack, LIFO lista (last-in-first-out) te “pushdown” lista.



Slika 2.5 Organizacija stoga.

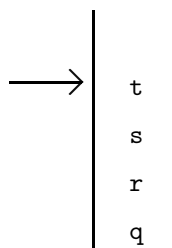
Slijede primjeri:

- Hrpa tanjura u restoranu; hrpa knjiga na podu.
- Glavni program poziva potprograme, a potprogrami pozivaju svoje potprograme.

prog MAIN	proc A1	proc A2	proc A3
...
...
...
call A1	call A2	call A3	...
r: ...	s: ...	t: ...	end;
...	
...	
end.	end;	end;	

Slika 2.6 Pozivi potprograma unutar programa.

Potrebno je da program pamti adrese povratka u nadređenu proceduru. Mnogi jezici (kao npr. Pascal, C, PL/I) u tu svrhu koriste stog. Kod ulaska u novi potprogram na stog se stavlja adresa povratka u nadređeni potprogram. Kad potprogram završi rad uzima se adresa s vrha stoga te se na nju prenosi kontrola (na slici: q je adresa povratka u operacijski sustav). Na stog se mogu stavljati i adrese iz istog potprograma (rekurzivni potprogrami).



Slika 2.7 Sadržaj stoga prilikom poziva potprograma.

- Interpreter višeg programskog jezika (npr. BASIC-a) računa vrijednost aritmetičkog izraza. Jednostavnije je ako se umjesto konvencionalne notacije koristi tzv. postfix notacija, dakle umjesto $(A/(B \uparrow C)*D+E)$ pišemo $ABC \uparrow /D * E +$. Računanje se provodi čitanjem postfix izraza s lijeva na desno. Kad susretnemo operand, stavimo ga na stog. Kad susretnemo operator, skidamo sa stoga onoliko operandi koliko taj operator traži, obavimo operaciju i rezultat vraćamo na stog.

Stog se, osim kao specijalni slučaj liste, može smatrati i posebnim apstraktnim tipom podataka. Obično ga se tada definira ovako.

Apstraktni tip podataka STACK

`elementtype` ... bilo koji tip.

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

`MAKE_NULL(&S)` ... funkcija pretvara stog `S` u prazni stog.

`EMPTY(S)` ... funkcija koja vraća “istinu” ako je `S` prazan stog. Inače vraća “laž”.

`PUSH(x,&S)` ... funkcija ubacuje element `x` na vrh stoga `S`. U terminima operacija s listama, to je ekvivalentno s `INSERT(x,FIRST(S),&S)`.

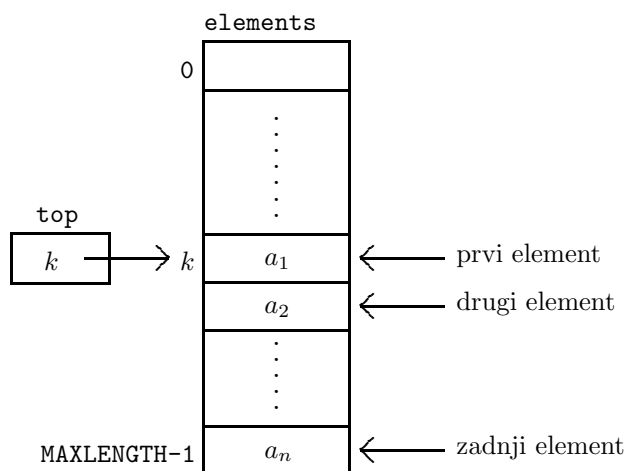
`POP(&S)` ... funkcija izbacuje element s vrha stoga `S`. To je ekvivalentno s `DELETE(FIRST(S),&S)`.

`TOP(S)` ... funkcija vraća element koji je na vrhu stoga `S` (stog ostaje nepromijenjen). To je ekvivalentno s `RETRIEVE(FIRST(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 se mogu i pojednostaviti.

2.2.1 Implementacija stoga pomoću polja

Ova se implementacija zasniva na strukturi podataka koju smo opisali za općenitu listu, s jednom malom modifikacijom. Da prilikom ubacivanja/izbacivanja ne bi morali prepisivati ostale elemente, listu umjesto u “gornji” smještamo u “donji” dio polja. Stog raste “prema gore”, dakle prema manjim indeksima polja.



Slika 2.8 Implementacija stoga pomoću polja.

Slijedi implementacija u C-u. Kao što vidimo, potprogrami `MAKE_NULL()`, `EMPTY()`, `PUSH()`, `POP()`, `TOP()` se svi svode na po jednu naredbu.

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

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

void MAKE_NULL (STACK *S_ptr) {
    S_ptr->top = MAXLENGTH;
}
```

```

int EMPTY (STACK S) {
    if (S.top >= MAXLENGTH)
        return 1;
    else
        return 0;
}

void PUSH (elementtype x, STACK *S_ptr) {
    if (S_ptr->top == 0)
        error("stack is full");
    else {
        S_ptr->top--;
        S_ptr->elements[S_ptr->top] = x;
    }
}

void POP (STACK *S_ptr) {
    if (EMPTY(*S_ptr))
        error("stack is empty");
    else
        S_ptr->top++;
}

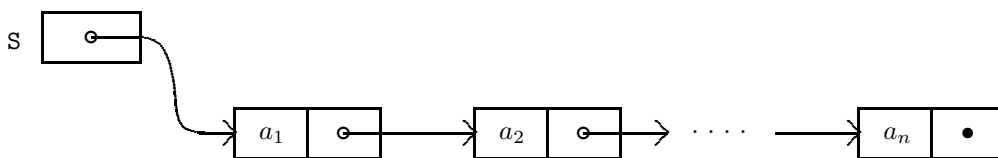
elementtype TOP (STACK S) {
    if (EMPTY(S))
        error("stack is empty");
    else
        return (S.elements[S.top]);
}

```

Implementacija stoga pomoću polja je vrlo pogodna, zato jer je vrijeme izvršavanja bilo koje operacije konstantno, tj. $\mathcal{O}(1)$. Mana je da se stog može prepuniti ako je **MAXLENGTH** premali.

2.2.2 Implementacija stoga pomoću pointera

Ova se implementacija zasniva na vezanoj listi (vidi potpoglavlje 2.1). Budući da kod stoga ne postoji pojam “pozicije”, nije nam više potrebna polazna ćelija (header), već je dovoljan pointer na prvu ćeliju. Time se struktura pojednostavljuje. Ćelija je isto građena kao u 2.1. Vrh stoga je na početku vezane liste. Stog se poistovjećuje s pointerom na početak vezane liste.



Slika 2.9 Implementacija stoga pomoću pointera.

Potprogrami **PUSH()** i **POP()** liče na **INSERT()** i **DELETE()** iz implementacije liste pomoću pointera, no nešto su jednostavniji budući da rade samo na početku liste. Ostala tri potprograma su gotovo trivijalni:

MAKE_NULL(&S) pridružuje $S = \text{NULL}$,

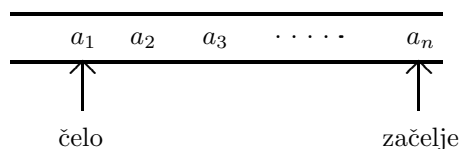
EMPTY(S) se svodi na provjeru da li je $S == \text{NULL}$,

TOP(S) vraća $S \rightarrow \text{element}$ (ako je S neprazan).

Implementacija pomoću pointera naizgled zaobilazi problem prepunjenja. Vrijeme izvršavanja bilo koje operacije je $\mathcal{O}(1)$.

2.3 Red (queue)

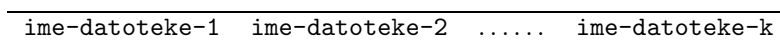
Red (queue) je specijalna vrsta liste. Elementi se ubacuju na jednom kraju liste (začelje), a izbacuju na suprotnom kraju (čelo). Drugi naziv: FIFO lista (first-in-first-out).



Slika 2.10 Red.

Navodimo nekoliko primjera za red.

- Ljudi koji čekaju na šalteru banke čine red.
- 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 print-serveru se formira red:



Slika 2.11 Niz datoteka u redu za štampanje.

Štampač štampa datoteku sa čela reda. Korisnik šalje datoteku na začelje reda.

- Slični primjeri se pojavljuju: kod korištenja jedne jedinice magnetske trake, ili kod izvođenja programa u batch obradi.

Kao i stog, red se može definirati kao posebni apstraktni tip podataka.

Apstraktni tip podataka QUEUE

`elementtype` ... bilo koji tip.

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

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

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

`ENQUEUE(x,&Q)` ... funkcija ubacuje element `x` na začelje reda `Q`. U terminima a.t.p.-a `LIST`, to je ekvivalentno s `INSERT(x,END(Q),&Q)`.

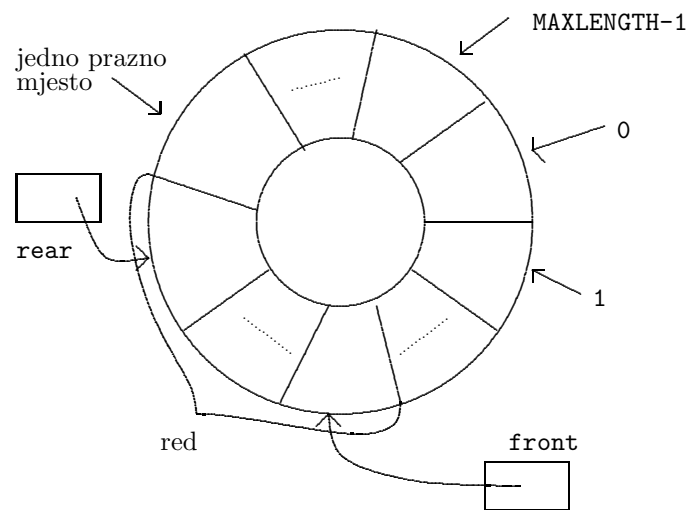
`DEQUEUE(&Q)` ... funkcija izbacuje element na čelu reda `Q`. Ekvivalentno s `DELETE(FIRST(Q),&Q)`.

`FRONT(Q)` ... funkcija vraća element na čelu reda `Q` (red ostaje nepromjenjen). To je ekvivalentno s `RETRIEVE(FIRST(Q),Q)`.

Implementacije reda se opet mogu dobiti iz implementacija liste, uz odgovarajuća pojednostavnjenja ili modifikacije.

2.3.1 Implementacija reda pomoću cirkularnog polja

Mogli bismo doslovno preuzeti implementaciju liste pomoću polja (poglavlje 2.1 - a_1 je čelo). Operacija `ENQUEUE()` se tada obavlja u konstantnom vremenu, budući da ne zahtijeva pomicanje dotadašnjih elemenata liste. No, `DEQUEUE()` nije jednako efikasna, jer zahtijeva da se cijeli ostatak reda prepíše za jedno mjesto "gore". Tome možemo doskočiti tako da uvedemo još jedan kursor na početak reda. Tada više nema prepisivanja, no pojavila se nova nepogodnost: ubacivanjem/izbacivanjem red "putuje" prema "donjem kraju" polja. Dosegnut ćemo donji kraj, makar u početnom dijelu polja ima mjesta. Još bolje rješenje je cirkularno polje. Zamišljamo da nakon zadnjeg indeksa ponovo slijedi početni indeks. Red zauzima niz uzastopnih ćelija polja. Postoji kursor na čelo i na začelje. Uslijed ubacivanja/izbacivanja elemenata, red "putuje" poljem u smjeru kazaljke na satu. Da bismo razlikovali situaciju praznog reda i posve punog reda (onog koji zauzima cijelo polje), dogovorit ćemo se da između čela i začelja mora biti bar jedno prazno mjesto. Znači, red ne može imati više od `MAXLENGTH - 1` elemenata. Cirkularnost se postiže tako da s indeksima računamo modulo `MAXLENGTH`.



Slika 2.12 Implementacija reda pomoću cirkularnog polja.

Slijedi zapis implementacije u C-u:

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

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

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

void MAKE_NULL (QUEUE *Q_ptr) {
    Q_ptr->front = 0;
    Q_ptr->rear = MAXLENGTH-1;
}

int EMPTY (QUEUE Q) {
    if (addone(Q.rear) == Q.front) return 1;
    else return 0;
}
```

```

void ENQUEUE (elementtype x, QUEUE *Q_ptr) {
    if (addone(addone(Q_ptr->rear)) == (Q_ptr->front))
        error("queue is full");
    else {
        Q_ptr->rear = addone(Q_ptr->rear);
        Q_ptr->elements[Q_ptr->rear] = x;
    }
}

void DEQUEUE (QUEUE *Q_ptr) {
    if (EMPTY(*Q_ptr)) error("queue is empty");
    else Q_ptr->front = addone(Q_ptr->front);
}

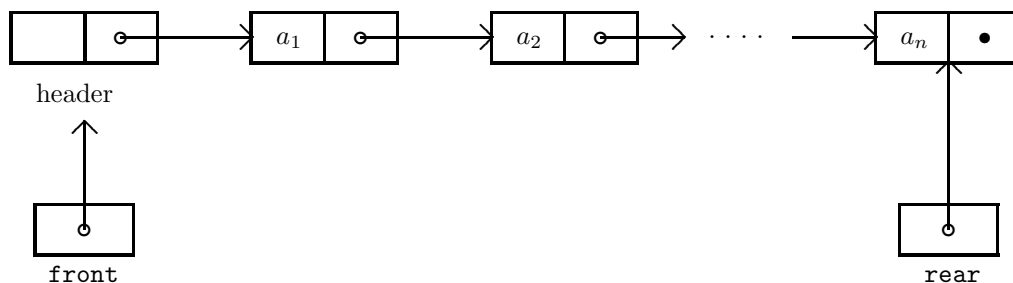
elementtype FRONT (QUEUE Q) {
    if (EMPTY(Q)) error("queue is empty");
    else return (Q.elements[Q.front]);
}

```

Vrijeme izvršavanja bilo koje operacije je ograničeno konstantom, dakle $\mathcal{O}(1)$.

2.3.2 Implementacija reda pomoću pointera

Isto kao u poglavlju 2.1. Početak vezane liste je čelo reda. Dodajemo još i pointer na kraj vezane liste. Header olakšava prikaz praznog reda. Vrijeme izvršavanja bilo koje operacije je opet $\mathcal{O}(1)$.



Slika 2.13 Implementacija reda pomoću pointera.

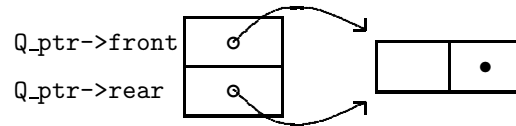
```

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

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

void MAKE_NULL (QUEUE *Q_ptr) {
    Q_ptr->front = (celltype*)malloc(sizeof(celltype));
    Q_ptr->front->next = NULL;
    Q_ptr->rear = Q_ptr->front;
}

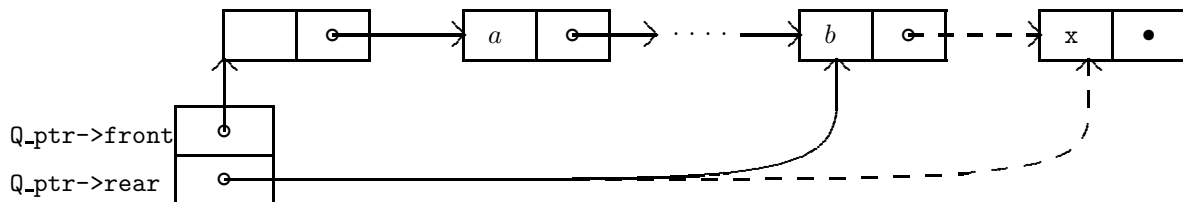
```



Slika 2.14 Dijagram za MAKE_NULL() - prazni red.

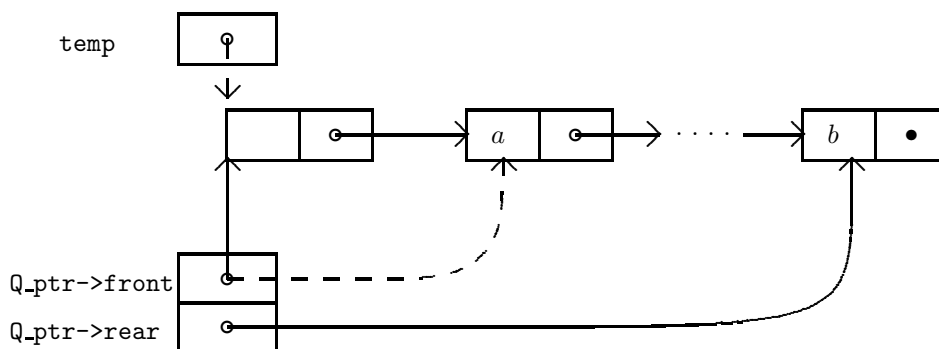
```
int EMPTY (QUEUE Q) {
    if (Q.front == Q.rear) return 1;
    else return 0;
}

void ENQUEUE (elementtype x, QUEUE *Q_ptr) {
    Q_ptr->rear->next = (celltype*)malloc(sizeof(celltype));
    Q_ptr->rear = Q_ptr->rear->next;
    Q_ptr->rear->element = x;
    Q_ptr->rear->next = NULL;
}
```



Slika 2.15 Dijagram za ENQUEUE().

```
void DEQUEUE (QUEUE *Q_ptr) {
    celltype *temp;
    if (EMPTY(*Q_ptr)) error("queue is empty");
    else {
        temp = Q_ptr->front;
        Q_ptr->front = Q_ptr->front->next;
        free(temp);
    }
}
```



Slika 2.16 Dijagram za DEQUEUE().

```
elementtype FRONT (QUEUE Q) {
    if (EMPTY(Q)) error("queue is empty");
    else return (Q.front->next->element);
}
```

3

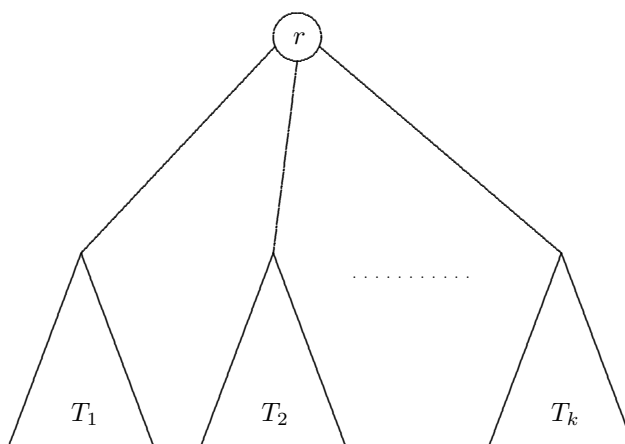
STABLA

3.1 (Uređeno) stablo

Za razliku od liste, zasnovane na linearnom uređaju podataka, stablo se zasniva na hijerarhijskom uređaju. Dakle među podacima postoji odnos “podređeni-nadređeni” ili “dijete-roditelj”.

(Uređeno) **stablo** T je neprazni konačni skup podataka istog tipa koje zovemo **čvorovi**. Pritom:

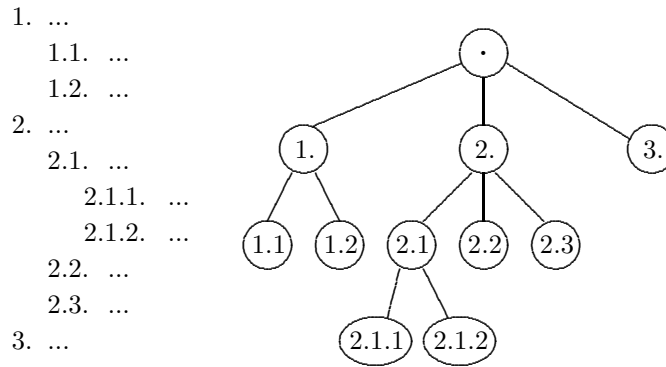
- postoji jedan istaknuti čvor r koji se zove **korijen** od T ;
- ostali čvorovi grade konačni niz (T_1, T_2, \dots, T_k) od 0 ili više disjunktnih (manjih) stabala.



Slika 3.1 Stablo T .

Ovo je bila rekurzivna definicija. Manja stabla T_1, T_2, \dots, T_k se zovu **pod-stabla** korijena r . Korijeni r_1, r_2, \dots, r_k od T_1, T_2, \dots, T_k su **djeca** od r , a r je njihov **roditelj**. Primjetimo da korijen nema roditelja, a svaki od preostalih čvorova ima točno jednog roditelja. Uređenost stabla se očituje u tome što među pod-stablama postoji linearan uređaj (zna se koje je prvo, koje drugo, ...). Slijede primjeri uređenih stabala.

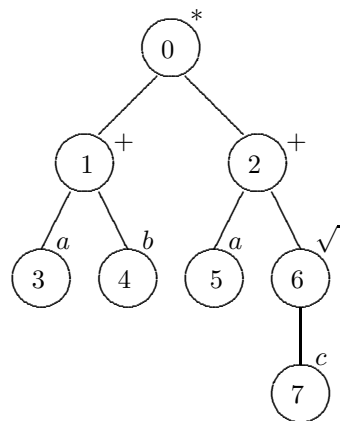
- Sadržaj neke knjige može se shvatiti kao stablo (i to zaista uređeno):



Slika 3.2 Sadržaj knjige prikazan kao stablo.

Slični primjeri su: organizacija poduzeća, organizacija oružanih snaga, struktura države, porodično stablo.

- Građa aritmetičkog izraza $(a + b) * (a + \sqrt{c})$ se može prikazati stablom. Čvorovi bez djece predstavljaju operande, a ostali čvorovi računске operacije. Uređenost stabla je važna ako su operacije ne-komutativne.



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

Stablo iz zadnjeg primjera je **označeno** stablo, tj. svakom čvoru je pridružen dodatni podatak koji zovemo **oznaka**. Razlikujemo čvor (tj. njegovo ime) od oznake. Ime čvora služi za identifikaciju (u istom stablu ne mogu postojati dva čvora s istim imenom). Oznaka čvora služi za informaciju (dva čvora mogu imati istu oznaku). Uočimo da su pojmovi: stablo, oznaka, čvor (u kontekstu stabla) redom analogni pojmovima: lista, element, pozicija (u kontekstu listi).

Niz čvorova i_1, i_2, \dots, i_m takvih da je i_p roditelj od i_{p+1} ($p = 1, \dots, m - 1$) 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. **Nivo** s je skup čvorova stabla sa svojstvom da jedinstveni put od korijena do tog čvora ima duljinu s . Nivo 0 čini sam korijen (po dogovoru). Nivo 1 čine djeca korijena, nivo 2 njihova djeca, itd. **Visina** stabla je maksimalni neprazni nivo. **List** je čvor bez djece. **Unutrašnji čvor** je čvor koji nije list. Djeca istog čvora zovu se **braća**.

Da bismo matematički pojam stabla pretvorili u apstraktni tip podataka, potrebno je definirati i operacije koje se obavljaju sa stablima. To se opet može učiniti na razne načine. Naš a.t.p. je jedna od mogućih varijanti.

Apstraktni tip podataka TREE

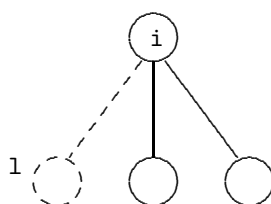
node ... bilo koji tip (imena čvorova). U skupu **node** uočavamo jedan poseban element **LAMBDA** (koji služi kao ime nepostojećeg čvora).

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**.

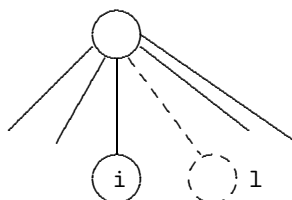
MAKE_ROOT(1,&T) ... funkcija pretvara stablo **T** u stablo koje se sastoji samo od korijena s oznakom **1**. Vraća čvor koji služi kao korijen (tj. njegovo ime).

INSERT_CHILD(1,i,&T) ... funkcija u stablo **T** ubacuje novi čvor s oznakom **1**, tako da on bude prvo po redu dijete čvora **i**. Funkcija vraća novi čvor. Nije definirana ako **i** ne pripada **T**.



Slika 3.4 Dijagram za **INSERT_CHILD()**.

INSERT_SIBLING(1,i,&T) ... funkcija u stablo **T** ubacuje novi čvor s oznakom **1**, tako da on bude idući po redu brat čvora **i**. Funkcija vraća novi čvor. Nije definirana ako je **i** korijen ili ako **i** ne pripada **T**.



Slika 3.5 Dijagram za **INSERT_SIBLING()**.

DELETE(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.

ROOT(T) ... funkcija vraća korijen od stabla **T**.

FIRST_CHILD(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**.

NEXT_SIBLING(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**.

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

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

CHANGE_LABEL(1,i,&T) ... funkcija mijenja oznaku čvora **i** u stablu **T**, tako da ta oznaka postane **1**. Nije definirana ako **i** ne pripada **T**.

3.1.1 Obilazak stabla

Obilazak stabla je algoritam kojim “posjećujemo” čvorove stabla, tako da svaki čvor posjetimo točno jednom. To je potrebno ako želimo obaviti neku obradu nad svim čvorovima (npr. ispisati oznake). Primjetimo da svaki obilazak uspostavlja jedan linearni uređaj među čvorovima. Najpoznatiji obilasci su: `PREORDER()`, `INORDER()`, `POSTORDER()`. Ova tri algoritma zadajemo rekursivno. Neka je T stablo sastavljeno od korijena r i pod-stabala T_1, T_2, \dots, T_k od korijena (vidi sliku uz definiciju stabla). Tada:

`PREORDER()` ... najprije posjećuje r , zatim obilazi T_1 , zatim obilazi T_2, \dots , na kraju obilazi T_k .

`INORDER()` ... najprije obilazi T_1 , zatim posjećuje r , zatim obilazi T_2, \dots , na kraju obilazi T_k .

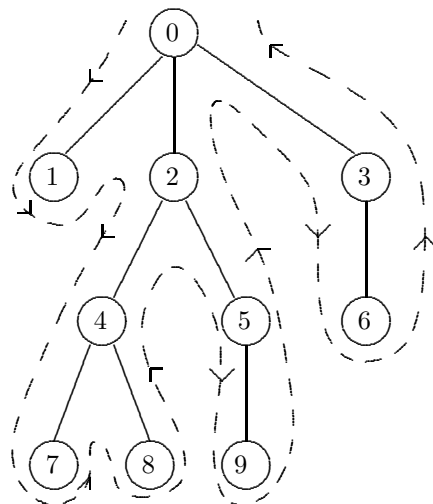
`POSTORDER()` ... najprije obilazi T_1 , zatim obilazi T_2, \dots , zatim obilazi T_k , na kraju posjećuje r .

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

`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



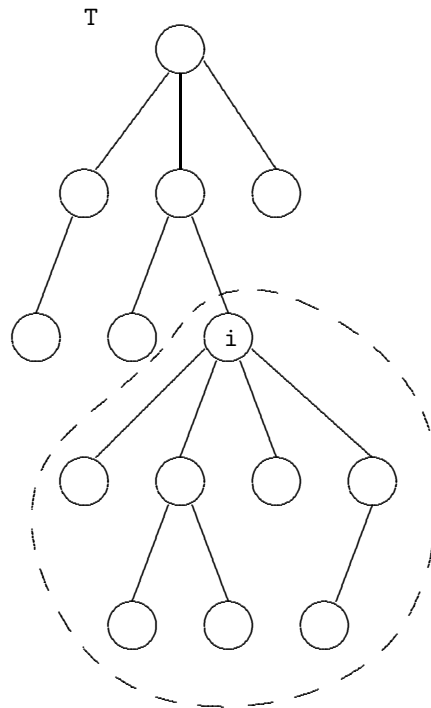
Slika 3.6 Obilazak stabla.

U apstraktnom tipu podataka `TREE` algoritmi obilaska mogu se zapisati kao potprogrami. U sljedećem potprogramu operacija posjećivanja čvora pretvorena je u ispis oznake čvora. Mogle bi se, naravno, obavljati i razne druge vrste obrade čvora.

```

void PREORDER (node i, TREE T) {
    /* obilazi se pod-stablo od T kojeg čini čvor i s potomcima */
    node c;
    printf ("...", LABEL(i,T));
    c = FIRST_CHILD(i,T);
    while (c != LAMBDA) {
        PREORDER (c,T);
        c = NEXT_SIBLING(c,T);
    }
}

```

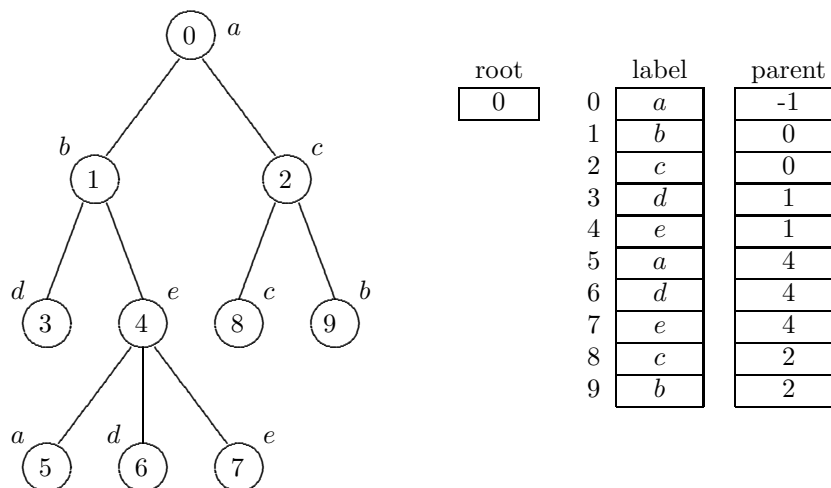


Slika 3.7 Podstablo od T s korijenom i.

3.1.2 Implementacija stabla na osnovu veze “čvor \rightarrow roditelj”

Zasniva se na tome da svakom čvoru eksplicitno zapišemo njegovog roditelja. Moguće su razne varijante, s obzirom na razne prikaze skupa čvorova. Mi uzimamo (bez velikog gubitka općenitosti) da su imena čvorova cijeli brojevi $0, 1, 2, \dots, n-1$, gdje je n broj čvorova. Stablo prikazujemo poljima. i -te ćelije polja opisuju i -ti čvor i u njima piše oznaka tog čvora odnosno kursor na roditelja. Dakle:

```
#define MAXNODES ... /* dovoljno velika konstanta */
#define LAMBDA -1
typedef int node;
typedef struct {
    node root;
    labeltype label[MAXNODES];
    node parent[MAXNODES];
} TREE;
```

Slika 3.8 Implementacija stabla na osnovu veze “čvor \rightarrow roditelj”.

Kursor `root` pokazuje gdje se nalazi korijen stabla. Ako je `MAXNODES` veći od stvarnog broja čvorova, neke od ćelija su slobodne. Možemo ih označiti i prepoznati tako da im upišemo neku nemoguću vrijednost (npr. `T[i].parent==i`).

Opisana struktura dobro podržava operacije `PARENT()` i `LABEL()`. Ostale operacije zahtijevaju pretraživanje cijelog polja. Daljnja mana je da se ne pamti redosljed braće - stablo je zapravo neuređeno. Ipak, možemo uvesti umjetno pravilo da su braća poredana po svojim imenima (indeksima). Tada vrijedi:

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

Opisana implementacija je dobra ako: nema mnogo ubacivanja/izbacivanja čvorova, nije potrebna uređenost stabla, pretežno se koriste operacije `PARENT()`, `LABEL()`.

3.1.3 Implementacija stabla na osnovu veze “čvor \rightarrow (prvo dijete, idući brat)”

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. Razmatrat ćemo varijantu s kursorima. Imena čvorova su cijeli brojevi.

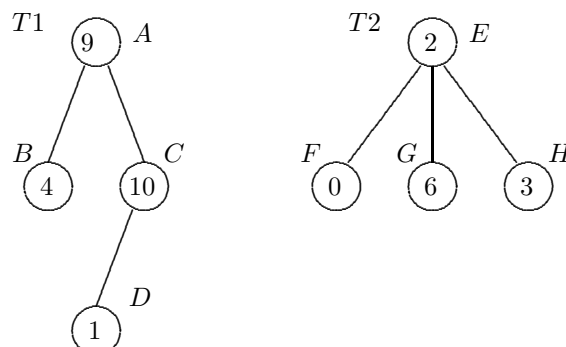
```
#define MAXNODES ... /* dovoljno velika konstanta */

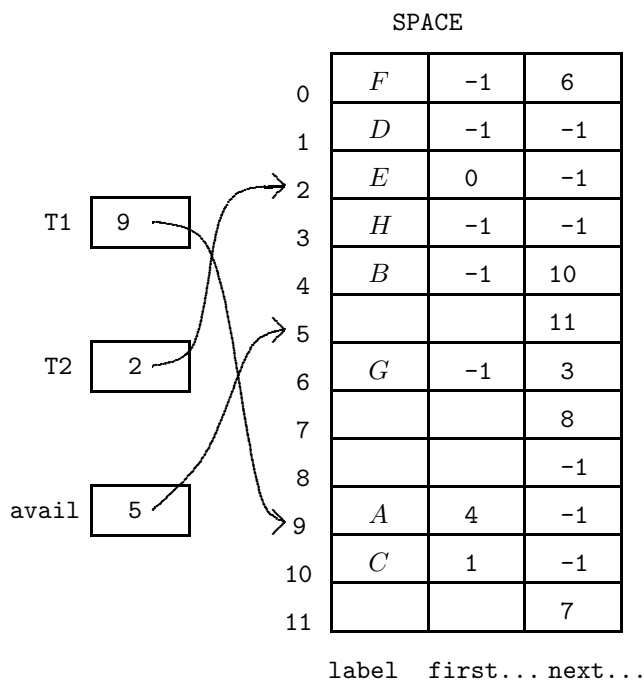
typedef int node;

typedef struct {
    labeltype label;
    node first_child, next_sibling;
} node_struct;

node_struct SPACE[MAXNODES];
```

Slobodnu memoriju zauzmemo globalnim poljem `SPACE[]`. Polje predstavlja “zalihu” ćelija od kojih će se graditi stabla. i -ta ćelija opisuje i -ti čvor. Stablo je prikazano kao vezana struktura ćelija. Stablo se poistovjećuje s kursorom na korijen, dakle: `typedef int TREE; .` Razna stabla s kojima radimo troše ćelije iz istog (jedinstvenog) polja `SPACE[]`. Sve slobodne ćelije (koje ne pripadaju ni jednom stablu) povezane su u vezanu listu, čiji poredak pokazuje globalni kursor `avail`. Slobodne ćelije se vežu kursorima smještenim npr. u komponenti `next_sibling`.





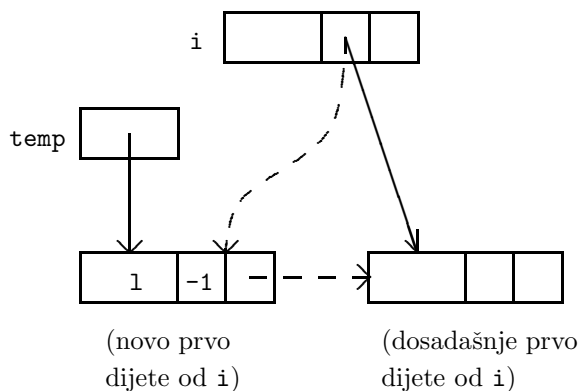
Slika 3.9 Stabla T1 i T2 koriste ćelije iz istog polja.

Sve operacije osim PARENT() mogu se efikasno implementirati. Npr.:

```

node INSERT_CHILD (labeltype l, node i) {
    node temp;
    if (avail == -1) error("no space");
    else {
        temp = avail;
        avail = SPACE[avail].next_sibling;
        SPACE[temp].label = l;
        SPACE[temp].first_child = -1;
        SPACE[temp].next_sibling = SPACE[i].first_child;
        SPACE[i].first_child = temp;
        return temp;
    }
}

```



Slika 3.10 Dijagram za INSERT_CHILD().

Opisana implementacija je pogodna onda kada ima puno ubacivanja/izbacivanja čvorova, ili kad se radi s više stabala koja se spajaju u veća ili kada se intenzivno koriste veze od roditelja prema djeci.

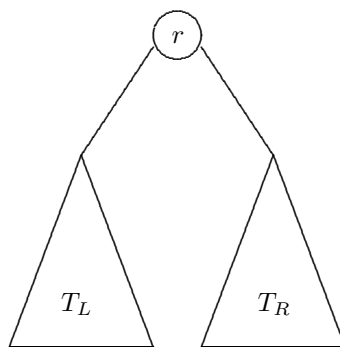
Ako nam je također potrebna i operacija `PARENT()`, tada možemo u ćeliju polja `SPACE[]` dodati i kursor na roditelja.

3.2 Binarno stablo

Umjesto stabala (koja se često razmatraju u matematici) u računarstvu se još češće pojavljuju binarna stabla. Radi se o nešto jednostavnijem i pravilnije građenom objektu kojeg je lakše prikazati u računalu.

Binarno stablo T je konačan skup podataka istog tipa koje zovemo **čvorovi**. Pri tome vrijedi:

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

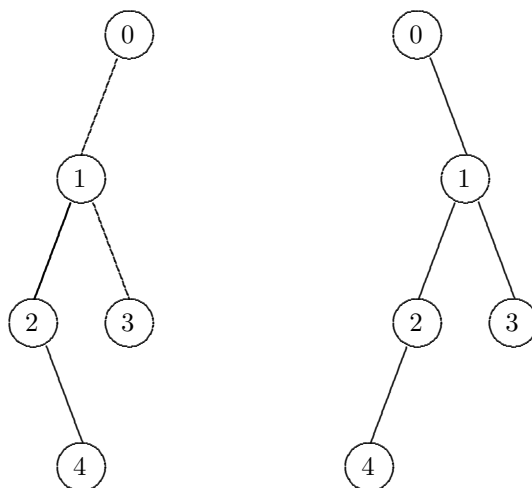


Slika 3.11 Binarno stablo.

Ako T sadrži korijen r , tada se binarna stabla T_L i T_R zovu lijevo i desno pod-stablo. 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**. Ostala terminologija je ista kao kod stabla. Primjenjuju se isti algoritmi obilaska.

Primjetimo da binarno stablo nije specijalni slučaj uređenog stabla. 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.

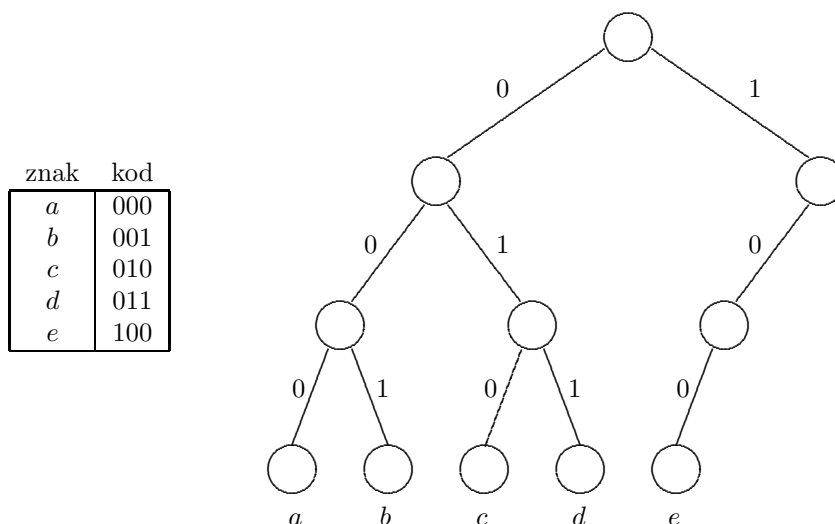


Slika 3.12 Dva različita binarna stabla.

Na dijagramima se vide dva različita binarna stabla. Isti dijagrami bi se mogli shvatiti i kao uređena stabla, no tada bi to bilo jedno te isto stablo.

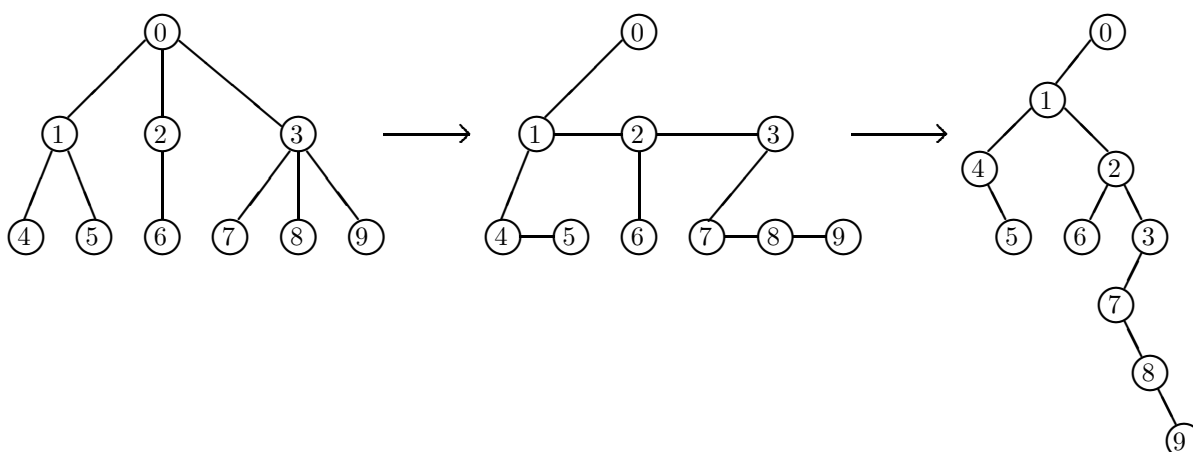
Slijede primjeri binarnih stabala:

- Ako je aritmetički izraz sasavljen od binarnih operacija tada se njegova građa može prikazati binarnim stablom (vidi primjer iz poglavlja 3.1)
- Znakovi su kodirani nizovima bitova. Tada se postupak dekodiranja može prikazati binarnim stablom:



Slika 3.13 Prikaz binarnog koda pomoću binarnog stabla.

- Bilo koje uređeno stablo može se interpretirati kao binarno stablo, na osnovu veza čvor \rightarrow prvo dijete i čvor \rightarrow idući brat. Npr.:



Slika 3.14 Interpretacija stabla kao binarnog stabla.

Ovu pretvorbu smo već nehotice koristili u drugoj implementaciji stabla. Zapravo se radilo o implementaciji binarnog stabla.

- Daljnji važni primjeri slijede u poglavlju 4. Binarno stablo se može upotrijebiti za prikaz skupova.

Binarno stablo se na razne načine može definirati kao apstraktni tip podataka. Osim operacija koje rade na nivou čvorova, lijepo se mogu definirati i operacije na nivou pod-stabala. Izložit ćemo prilično opširni popis operacija. Neke od njih se mogu izvesti pomoću drugih, no ipak ih navodimo zbog spretnijeg rada.

Apstraktni tip podataka BTREE

node ... bilo koji tip (imena čvorova). U skupu **node** uočavamo jedan poseban element **LAMBDA** (koji služi kao ime nepostojećeg čvora).

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

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

MAKE_NULL(&T) ... funkcija pretvara binarno stablo **T** u prazno binarno stablo.

EMPTY(T) ... funkcija vraća "istinu" ako i samo ako je **T** prazno binarno stablo.

CREATE(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**.

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

INSERT_LEFT_CHILD(l,i,&T), **INSERT_RIGHT_CHILD(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. Nije definirana ako **i** ne pripada **T** ili ako **i** već ima dotično dijete.

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

ROOT(T) ... funkcija vraća korijen binarnog stabla **T**. Ako je **T** prazno, vraća **LAMBDA**.

LEFT_CHILD(i,T), **RIGHT_CHILD(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**.

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

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

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

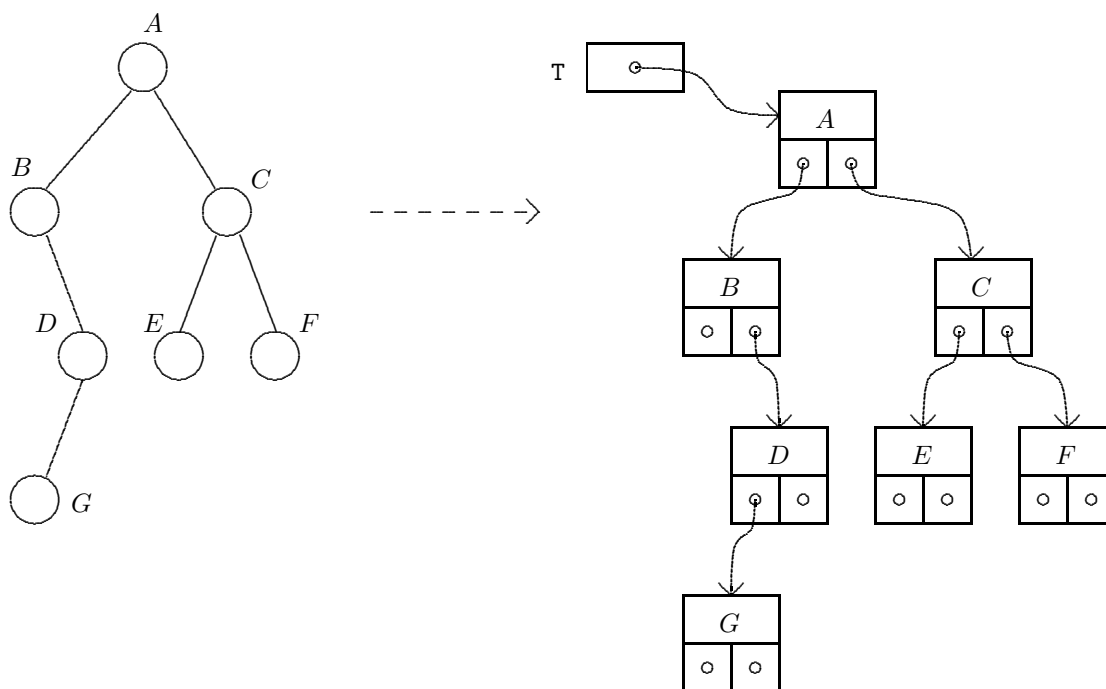
3.2.1 Implementacija binarnog stabla pomoću pointera

Svakom čvoru eksplicitno zapišemo njegovo lijevo i desno dijete. Veza se može zapisati pomoću kursora ili pomoću pointera. Varijanta pomoću kursora bi bila gotovo identična implementaciji opisanoj na kraju poglavlja 3.1. Zato promatramo varijantu pomoću pointera. Imamo definicije:

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

typedef celltype *node;
typedef celltype *BTREE;
```

Svaki čvor prikazujemo jednom ćelijom. Čvor je zato jednoznačno određen pointerom na tu ćeliju. Binarno stablo se gradi kao struktura ćelija povezanih pointerima. Binarno stablo poistovjećujemo s pointerom na korijen. Prazno stablo je prikazano pointerom NULL. Ukoliko koristimo funkciju `PARENT()`, tada je zgodno u ćeliju dodati još jedan pointer.



Slika 3.15 Implementacija binarnog stabla pomoću pointera.

Sve operacije iz a.t.p. BTREE se mogu efikasno implementirati, tako da vrijeme izvršavanja bude $\mathcal{O}(1)$. Npr.:

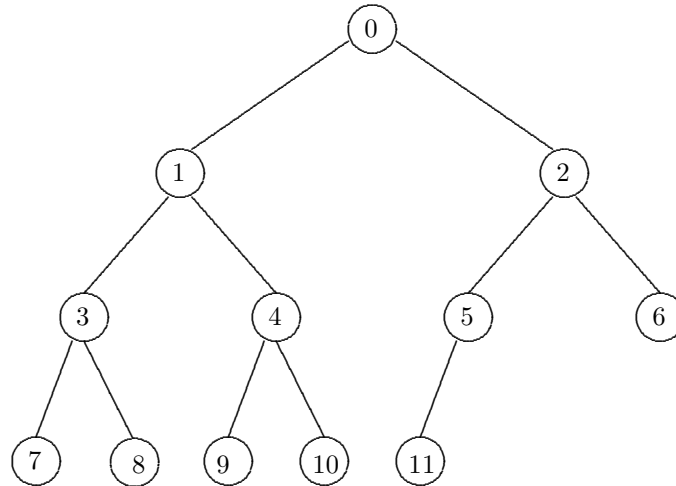
```
void CREATE (labeltype l, BTREE TL, BTREE TR, BTREE *Tptr) {
    *Tptr = (celltype*)malloc(sizeof(celltype));
    (*Tptr)->label = l;
    (*Tptr)->leftchild = TL;
    (*Tptr)->rightchild = TR;
}
```

3.2.2 Implementacija potpunog binarnog stabla pomoću polja

Potpuno binarno stablo je građeno od n čvorova, s imenima $0, 1, 2, \dots, n-1$. 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 čvor $2i+2$ (ako je $2i+2 > n-1$ tada i nema desno dijete).

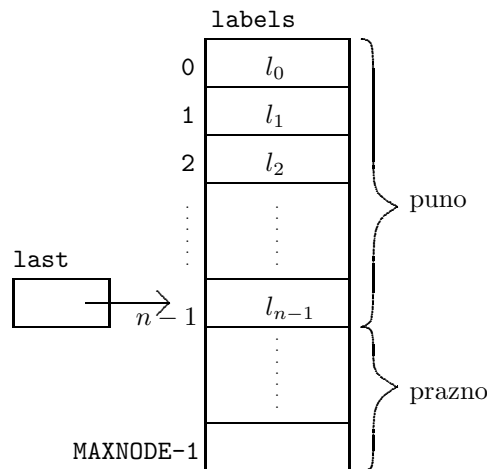
Npr. potpuno binarno stablo s $n = 12$ čvorova izgleda kao na slici.



Slika 3.16 Potpuno binarno stablo.

Na svim nivoima osim zadnjeg postoje svi mogući čvorovi. Čvorovi na zadnjem nivou su “gurnuti” na lijevu stranu. Numeriranje ide s nivoa 0 na nivo 1, nivo 2, itd., s lijeva na desno.

Potpuno binarno stablo treba zamišljati kao objekt sa statičkom građom. Na njega se ne namjeravaju primjenjivati operacije poput `CREATE()`, `LEFT_SUBTREE()`, `RIGHT_SUBTREE()`, jer rezultat više ne bi bio potpuno binarno stablo. Umjesto toga potrebno je “manevrirati” po već zadanom stablu, te čitati ili mijenjati oznake čvorova. Eventualno se dopušta ubacivanje/izbacivanje čvorova na desnom kraju zadnjeg nivoa.



Slika 3.17 Implementacija potpunog binarnog stabla pomoću polja.

Potpuno binarno stablo se može elegantno prikazati na računalu pomoću polja. i -ta ćelija polja sadrži oznaku čvora i . Također imamo kursor koji pokazuje zadnji čvor $n - 1$. Dakle imamo definicije:

```
#define MAXNODE .../* dovoljno velika konstanta */
typedef int node;
typedef struct {
    int last;
    labeltype labels[MAXNODE];
} BTREE;
```

Manevriranje po stablu se obavlja na očigledni način. Korijen je predstavljen 0-tom ćelijom. Lijevo i desno dijete čvora iz i -te ćelije nalaze se u $(2i + 1)$ -oj i $(2i + 2)$ -oj ćeliji (ako postoje). Roditelj čvora iz i -te ćelije je u $\lfloor (i - 1)/2 \rfloor$ -toj ćeliji.

4

SKUPOVI

4.1 (Općeniti) skup

U mnogim algoritmima kao matematički model pojavljuje se **skup**. Znači, pohranjuje se 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, možemo govoriti o tome koji je element veći, a koji je manji.

Slijedi primjer korištenja skupova. Pretraživanjem baze podataka dobivamo skupove imena osoba, $A = \{\text{osobe s barem 10 godina staža}\}$, $B = \{\text{osobe s višom stručnom spremom}\}$. Tada osobe s barem deset godina staža ili VSS računamo kao $A \cup B$, a osobe s barem 10 godina staža i VSS kao $A \cap B$. Slično, $B \setminus A$ čine osobe s VSS koje još nemaju 10 godina staža.

Skup ćemo definirati kao apstraktni tip podataka, s operacijama koje su uobičajene u matematici.

Apstraktni tip podataka SET

`elementtype` ... bilo koji tip s totalnim uređajem \leq .

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

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

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

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

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

`MIN(A)`, `MAX(A)` ... funkcija vraća najmanji odnosno najveći element skupa `A`, u smislu uređaja \leq . Nije definirana ako je `A` prazan skup.

`SUBSET(A,B)` ... funkcija vraća "istinu" ako je $A \subseteq B$, inače vraća "laž".

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

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

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

Ovako zadani apstraktni tip podataka `SET` dosta je teško implementirati. Ako postignemo efikasno obavljanje jednih operacija, sporo će se obavljati neke druge operacije. U ovom potpoglavlju gledamo implementacije napravljene sa ciljem da se spretno obavljaju operacije s više skupova: `UNION()`, `INTERSECTION()`, `DIFFERENCE()`, `SUBSET()`. Implementacije koje pogoduju ostalim operacijama obrađuju se u poglavljima 4.2. i 4.3.

4.1.1 Implementacija skupa pomoću bit-vektora

Uzimamo bez velikog gubitka općenitosti da je `elementtype = {0, 1, 2, ..., 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:

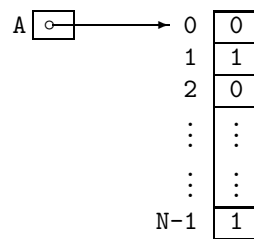
```
#define N ... /* dovoljno velika konstanta */
typedef char *SET; /* početna adresa char polja duljine N */
```

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.

Operacije iz apstraktnog tipa podataka `SET` se mogu isprogramirati na očigledni način. `INSERT()`, `DELETE()` i `MEMBER()` zahtijevaju konstantno vrijeme, budući da se svode na direktan pristup i -tom bitu. `UNION()`, `INTERSECTION()`, `DIFFERENCE()`, `SUBSET()` zahtijevaju vrijeme proporcionalno `N`. Ispisujemo `UNION()`, a ostale funkcije su slične:

```
void UNION (SET A, SET B, SET *C_ptr) {
    int i;
    for (i=0; i<N; i++)
        (*C_ptr)[i] = (A[i]==1) || (B[i]==1);
}
```

Ova implementacija postaje neupotrebljiva ako je `N` velik.

4.1.2 Implementacija skupa pomoću sortirane vezane liste

Prikazujemo skup kao listu. 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 efikasnije obavljale, dobro je da lista bude sortirana u skladu s \leq . Pretpostavimo da su definirani tipovi `celltype`, `LIST` iz poglavlja 2.1. Dalje ispisujemo `INTERSECTION()`. Pretpostavljamo sljedeću definiciju tipa `SET`.

```
typedef celltype *SET;

void INTERSECTION (SET ah, SET bh, SET *chp) {
    /* Računa presjek skupova A i B, prikazanih sortiranim vezanim listama
       čije headere pokazuju pointeri ah i bh. Rezultat je skup C, prikazan
       sortiranom vezanom listom čiji pointer na header pokazuje chp */
```

```

celltype *ac, *bc, *cc;
/* tekuće ćelije u listi za A i B, te zadnja ćelija u listi za C */
*chp = (celltype*)malloc(sizeof(celltype));
/* stvori header liste za 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;
}

```

Funkcije `UNION()`, `DIFFERENCE()` i `SUBSET()` su vrlo slične. Vrijeme izvršavanja je proporcionalno duljini jedne od listi.

4.2 Rječnik

Često nije potrebno obavljati složene operacije kao što su \cup , \cap , \setminus , \subseteq . Umjesto toga, pamti se jedan skup, te se obavljaju povremena ubacivanja i izbacivanja elemenata. Također, povremeno treba ustanoviti nalazi li se zadani element u skupu ili ne. Takav skup nazivamo **rječnik** (dictionary). Apstraktni tip podataka `DICTIONARY` definiramo slično kao apstraktni tip podataka `SET`, s time da se popis operacija reducira na `MAKE_NULL()`, `INSERT()`, `DELETE()` i `MEMBER()`. Slijede primjeri:

- Pravopis je popis ispravno napisanih riječi nekog jezika. Da bismo ustanovili da li je neka riječ ispravno zapisana, gledamo postoji li ona u pravopisu. Povremeno u pravopis ubacujemo nove izraze ili izbacujemo zastarjele.
- Neki tekst procesori imaju u sebi tzv. spelling checker, dakle popis ispravno napisanih engleskih riječi i program koji uspoređuje naš tekst s popisom.
- Višekorisničko računalo prepoznaje korisnike na osnovu njihovih imena. Kod prijavljivanja, korisnik upisuje ime, a stroj provjerava postoji li ime na popisu. Ovlaštena osoba ubacuje imena novih korisnika, ili briše imena odsutnih.

4.2.1 Implementacija rječnika pomoću bit-vektora

Sasvim isto kao u poglavlju 4.1. Operacije `INSERT()`, `DELETE()` i `MEMBER()` se obavljaju u vremenu $\mathcal{O}(1)$. Implementacija postaje neupotrebljiva ako je `elementtype` velik.

4.2.2 Implementacija rječnika pomoću liste

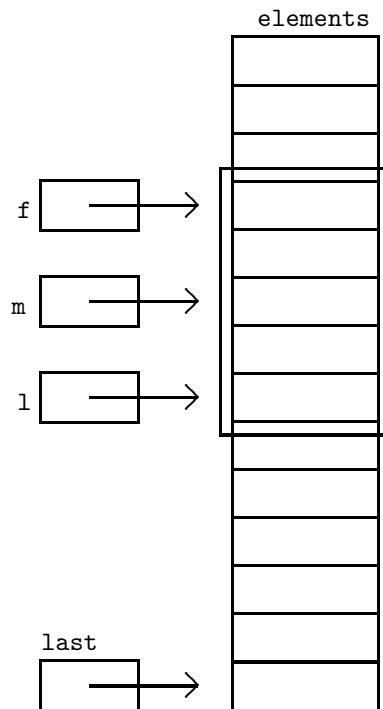
Skup shvatimo kao listu, koja može biti sortirana (u skladu s \leq) ili nesortirana, te prikazana pomoću polja ili pointera (vidi potpoglavlje 2.1). Od četiri moguće varijante detaljnije promatramo **sortiranu**

listu prikazanu kao **polje**. Ta varijanta je pogodna za “statične” skupove, dakle one za koje se često obavlja funkcija `MEMBER()`, a rjeđe `INSERT()` ili `DELETE()`. Funkcija `MEMBER()` obavlja se u vremenu $\mathcal{O}(\log_2 n)$, gdje je n duljina liste. Služimo se algoritmom **binarnog traženja**. Uz pretpostavku da je tip `LIST` definiran kao u potpoglavlju 2.1, te uz

```
typedef LIST SET;
```

imamo:

```
int MEMBER (elementtype x, SET A) {
    int f, l; /* indeks prvog i zadnjeg elementa razmatrane pod-liste */
    int m; /* indeks srednjeg elementa razmatrane pod-liste */
    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.

Da bi se čuvala sortiranost, `INSERT()` mora ubaciti novi element na “pravo” mjesto, što zahtjeva u najgorem slučaju $\mathcal{O}(n)$ prepisivanja elemenata. Slično, `DELETE()` zahtijeva $\mathcal{O}(n)$ prepisivanja.

Ako se za naš rječnik često obavljaju `INSERT()` i `DELETE()`, a rjeđe `MEMBER()`, tada su pogodnije ostale tri varijante implementacije pomoću liste. Kod sve tri varijante može se izbjeći prepisivanje

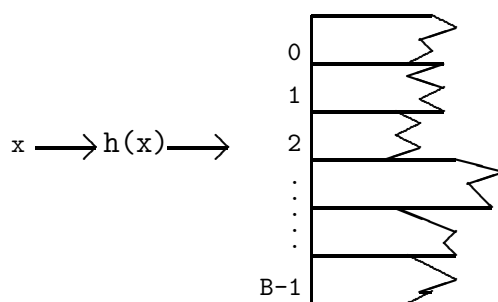
elemenata prilikom ubacivanja i izbacivanja. Ipak, vrijeme za `INSERT()`, `DELETE()` i `MEMBER()` je u najgorem slučaju $\mathcal{O}(n)$ (zbog traženja elemenata odnosno njegovog mjesta u listi).

4.2.3 Implementacija rječnika pomoću rasute (hash) tablice

Implementaciju pomoću bit-vektora možemo gledati kao bijekciju oblika: `elementtype` \rightarrow memorija. Naime, svakoj mogućoj vrijednosti tipa `elementtype` pridružuje se njen zasebni bit memorije. Takva implementacija je u jednu ruku idealna, jer se funkcije `INSERT()`, `DELETE()`, `MEMBER()` izvršavaju u vremenu $\mathcal{O}(1)$. S druge strane, ta implementacija je 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 ćelija koje zovemo pretinci. Indeksi pretinaca su $0, 1, 2, \dots, B-1$.

Implementacija rječnika pomoću hash tablice bazira se na tome da element x spremimo u pretinac s indeksom $h(x)$. Kasnije ga u istom pretincu i tražimo.



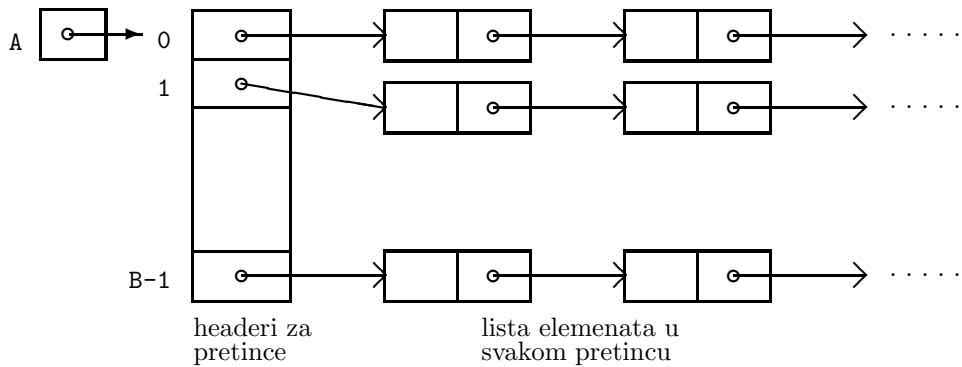
Slika 4.3 Hash funkcija i hash tablica.

Za funkcioniranje ove ideje važno je da `h()` jednoliko raspoređuje vrijednosti iz skupa `elementtype` na pretince $0, 1, 2, \dots, B-1$. Npr., ako uzmemo da je `elementtype` skup svih nizova znakova duljine 10, tada bismo mogli definirati ovakav `h()`:

```
int h (elementtype x) {
    int i, sum=0;
    for (i=0; i<10; i++) sum += x[i];
    return sum % B;
}
```

Postoje razne varijante implementacije pomoću hash tablice. One se razlikuju po građi pretinca tablice.

Otvoreno “haširanje”. Pretinac je građen kao vezana lista. 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.



Slika 4.4 Organizacija podataka u otvorenom haširanju.

Potrebne definicije u C-u su:

```
#define B ... /* pogodna konstanta */
typedef struct cell_tag {
    elementtype element;
    struct cell_tag *next;
} celltype;

typedef celltype **DICTIONARY; /* početna adresa polja headera */

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

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

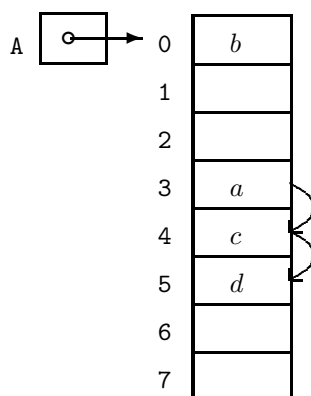
void INSERT (elementtype x, DICTIONARY *Ap) {
    int bucket;
    celltype *oldheader;
    if (MEMBER(x,*Ap) == 0) {
        bucket = h(x);
        oldheader = (*Ap)[bucket];
        (*Ap)[bucket] = (celltype*)malloc(sizeof(celltype));
        (*Ap)[bucket]->element = x;
        (*Ap)[bucket]->next =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

```

void DELETE (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 ćeliji */
            temp = (*Ap)[bucket];
            (*Ap)[bucket] = (*Ap)[bucket]->next;
            free(temp);
        }
        else { /* x, ukoliko postoji, nije u prvoj ćeliji */
            current = (*Ap)[bucket]; /* current pokazuje prethodnu ćeliju */
            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;
        }
    }
}

```

Zatvoreno “haširanje”. Pretinac ima fiksni kapacitet. Zbog jednostavnosti uzimamo da u jedan pretinac stane jedan element. Dakle, hash tablica je polje ćelija tipa `elementtype`. Ako trebamo ubaciti element x , a ćelija $h(x)$ je već puna, tada pogledamo na alternativne lokacije: $hh(1,x)$, $hh(2,x)$, ... sve dok ne nađemo jednu praznu. Najčešće se $hh(i,x)$ задаје kao $hh(i,x) = (h(x)+i) \% B$; to se onda zove “linearno” haširanje. Na slici je nacrtana 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.

Prazne pretince prepoznavamo 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 je ispravan 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 ćelija se kasnije može upotrijebiti kod novog ubacivanja, kao da je prazna. Slijede definicije u C-u.

```

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

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

void MAKE_NULL (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
       prazni pretinac, ili dok se ne vrati na mjesto polaska. Funkcija
       vraća indeks pretinca na kojem je stala iz bilo kojeg od ovih 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 samo također stane i onda kad naiđe na DELETED */
    ...
}

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

void INSERT (elementtype x, DICTIONARY *Ap) {
    int bucket;
    if (MEMBER(x,*Ap)) return; /* x je već u A */
    bucket = locate1(x,*Ap);
    if ( ((*Ap)[bucket]==EMPTY) || ((*Ap)[bucket]==DELETED) )
        (*Ap)[bucket] = x;
    else
        error("table is full");
}

void DELETE (elementtype x; DICTIONARY *Ap) {
    int bucket;
    bucket = locate(x,*Ap);
    if ( (*Ap)[bucket] == x ) (*Ap)[bucket] = DELETED;
}

```

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 kod otvorenog haširanja bude $n \leq 2 \cdot B$, a kod zatvorenog haširanja $n \leq 0.9 \cdot B$. Ako je to ispunjeno, tada možemo očekivati da bilo koja od operacija INSERT(), DELETE(), MEMBER() zahtijeva svega nekoliko čitanja iz tablice. Ako se tablica previše napuni, treba je prepisati u novu, veću.

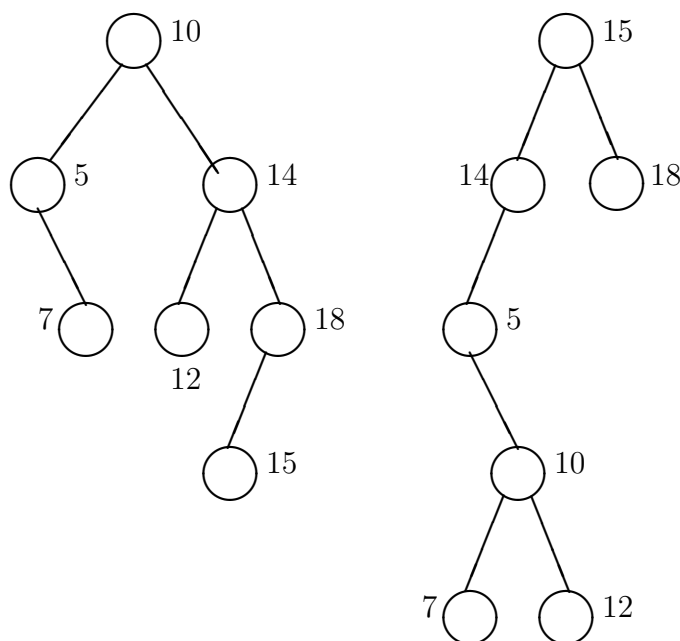
4.2.4 Implementacija rječnika pomoću binarnog stabla traženja

Binarno stablo T je **binarno stablo traženja** 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 pod-stablu od i manje od oznake od i . Također, oznake svih čvorova u desnom pod-stablu 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 stabla i obratno. Element rječnika služi kao oznaka odgovarajućeg čvora stabla - kažemo da je element "spremljen" u tom čvoru. Primjetimo da će svi čvorovi našeg stabla imati različite oznake, makar se to ne zahtjeva u definiciji binarnog stabla traženja.

Na slici se vidi prikaz skupa $A = \{5, 7, 10, 12, 14, 15, 18\}$ pomoću binarnog stabla traženja. Prikaz nije jedinstven. 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 (vidi potpoglavlje 3.2). Tada su nam potrebne sljedeće definicije:

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

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

Operacija `MAKE_NULL()` je trivijalna, svodi se na pridruživanje vrijednosti `NULL` pointeru. Operacija `MEMBER()` lako se implementira zahvaljujući svojstvima binarnog stabla traženja:

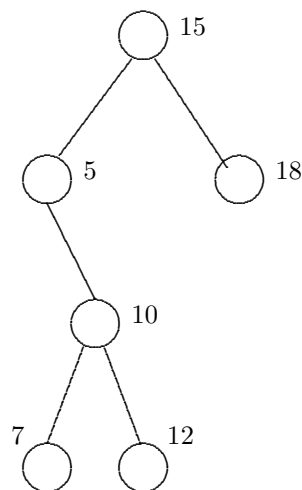
```
int MEMBER (elementtype x, DICTIONARY A) {
    if (A == NULL) return 0;
    else if (x == A->element) return 1;
    else if (x < A->element) return(MEMBER(x,A->leftchild));
    else /* x > A->element */ return(MEMBER(x,A->rightchild));
}
```


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

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

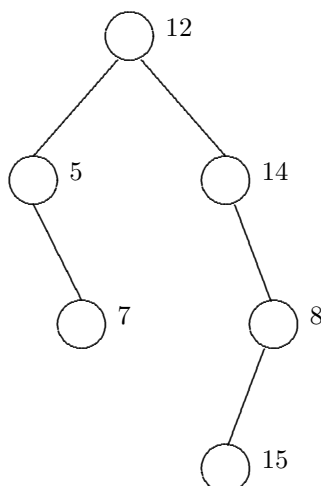
Nešto je složenija operacija `DELETE(x,&A)`. Imamo tri slučaja:

- x je u listu; tada jednostavno izbacimo list iz stabla. (npr. za x jednak 15 u prvom binarnom stablu sa slike 4.6).
- x je u čvoru koji ima samo jedno dijete. Nadomjestimo čvor od x s njegovim djetetom. (npr. za x jednak 14 u drugom binarnom stablu sa slike 4.6. Čvor koji pohranjuje 14 se izbacuje, a umjesto njega čvor od 5 postaje lijevo dijete čvora od 15).



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

- x je u čvoru koji ima oba djeteta. Tada nađemo najmanji element y u desnom pod-stablu čvora od x . Izbacimo čvor od y (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).



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

Sve se ovo spretno može zapisati ako uvedemo pomoćnu funkciju `DELETE_MIN(&A)`. Ta funkcija iz nepraznog binarnog stabla `A` izbacuje čvor s najmanjim elementom, te vraća taj najmanji element.

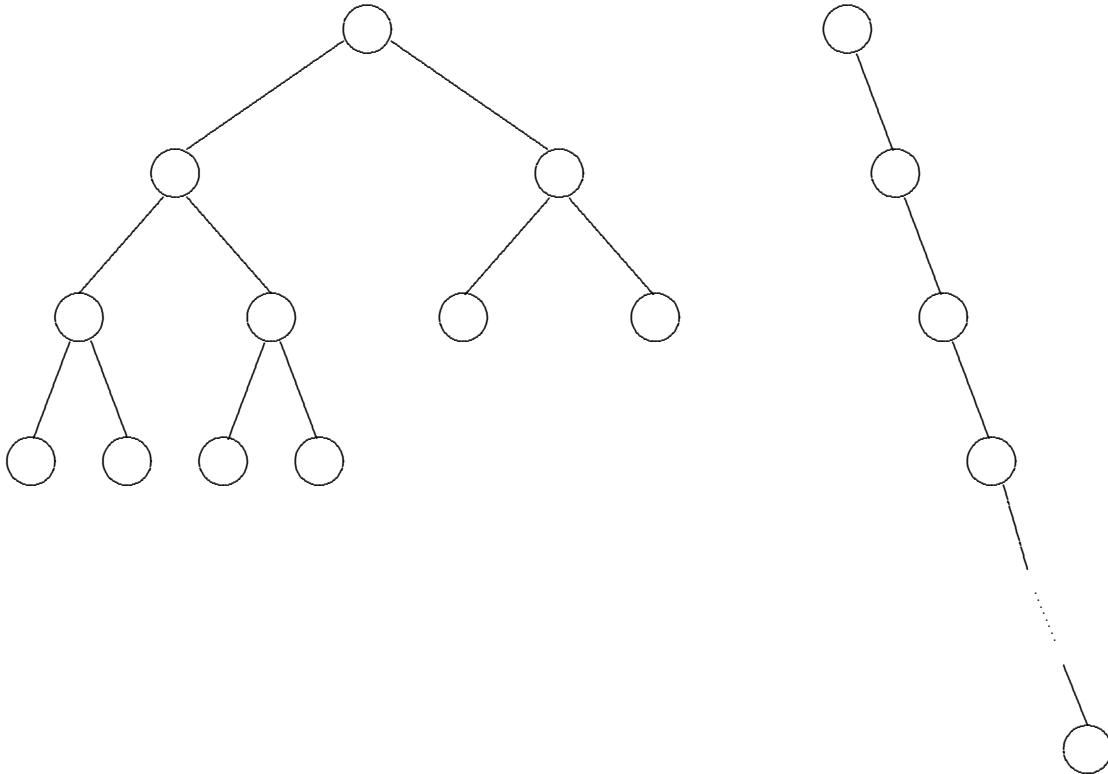
```

elementtype DELETE_MIN (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 djetete */
        minel = DELETE_MIN( &((*Ap)->leftchild) );
    return minel;
}

void DELETE (elementtype x, DICTIONARY *Ap) {
    celltype *temp;
    if ( *Ap != NULL )
        if ( x < (*Ap)->element )
            DELETE( x, &((*Ap)->leftchild) );
        else if ( x > (*Ap)->element )
            DELETE( 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 lijevom djetetom */
            temp = *Ap; *Ap = (*Ap)->leftchild; free(temp);
        }
        else
            /* postoje oba djeteta */
            (*Ap)->element = DELETE_MIN ( &((*Ap)->rightchild) );
}

```

Svaka od funkcija `MEMBER()`, `INSERT()`, `DELETE()` prolazi jednim putom, od korijena binarnog stabla do nekog čvora. Zato je vrijeme izvršavanja svih operacija ograničeno visinom stabla. Neka riječnik ima n elemenata. Visina stabla tada varira između $\lceil \log_2(n+1) \rceil - 1$ i $n - 1$. Ekstremni slučajevi su potpuno binarno stablo i “ispruženo” stablo - lanac.



Slika 4.9 Ekstremni slučajevi visine binarnog stabla: potpuno i ispruženo binarno stablo.

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

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

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

Opisana implementacija je samo 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. `INSERT()` i `DELETE()` su znatno kompliciranije nego kod nas. Naime, te operacije preuzimaju zadatak “pregradnje” stabla kad god se ono suviše “iskrivi”.

4.3 Prioritetni red

U nekim algoritmima pojavljuje se skup čijim elementima su pridruženi cijeli ili realni brojevi - prioriteti. Operacije su: ubacivanje novog elementa, te “izbacivanje” elementa s najmanjim prioritetom. Takav skup nazivamo **prioritetni red**. Slijede primjeri.

- Pacijenti ulaze u čekaonicu, te iz nje odlaze liječniku na pregled. Prvi na redu za liječnika nije onaj koji je prvi ušao, već onaj čije je stanje najteže.

- U računalu se formira prioritetni red programa (procesa) koji čekaju na izvođenje. Svakom programu je pridružen prioritet. Procesor uzima iz reda program s najmanjim prioritetom, te ga izvodi.

Spomenute probleme svodimo na jednostavniji problem ubacivanja elemenata u skup, te izbacivanja najmanjeg elementa. Naime, umjesto originalnih elemenata x , promatramo uređene parove $(\text{prioritet}(x), x)$. Za uređene parove definiramo leksikografski uređaj: $(\text{prioritet}(x_1), x_1)$ je manji-ili-jednak od $(\text{prioritet}(x_2), x_2)$ ako je $(\text{prioritet}(x_1) \text{ manji od } \text{prioritet}(x_2))$ ili $(\text{prioritet}(x_1) \text{ jednak } \text{prioritet}(x_2) \text{ i } x_1 \text{ manji-ili-jednak od } x_2)$. Ova dosjetka opravdava sljedeću definiciju apstraktnog tipa podataka.

Apstraktni tip podataka **PRIORITY_QUEUE**

elementtype ... bilo koji tip s totalnim uređajem \leq .

PRIORITY_QUEUE ... podatak ovog tipa je konačni skup čiji elementi su (međusobno različiti) podaci tipa **elementtype**.

MAKE_NULL(&A) ... funkcija pretvara skup **A** u prazni skup.

EMPTY(A) ... funkcija vraća "istinu" ako je **A** prazan skup, inače vraća "laž".

INSERT(x, &A) ... funkcija ubacuje element **x** u skup **A**, tj. mijenja **A** u $A \cup \{x\}$.

DELETE_MIN(&A) ... funkcija iz skupa **A** izbacuje najmanji element, te vraća taj izbačeni element. Nije definirana ako je **A** prazan skup.

Primjetimo da se a.t.p. **PRIORITY_QUEUE** može smatrati restrikcijom a.t.p. **SET**. Naime funkcija **DELETE_MIN()** zapravo je kombinacija od **MIN()** i **DELETE()**.

4.3.1 Implementacija prioritetnog reda pomoću sortirane vezane liste

Prikazujemo red kao listu. Od raznih varijanti najbolja se čini sortirana vezana lista. **DELETE_MIN()** pronalazi i izbacuje prvi element u listi, pa ima vrijeme $\mathcal{O}(1)$. **INSERT()** mora ubaciti novi element na "pravo mjesto", što znači da mora pročitati u prosjeku pola liste. Zato **INSERT()** ima vrijeme $\mathcal{O}(n)$, gdje je n broj elemenata u redu.

4.3.2 Implementacija prioritetnog reda pomoću binarnog stabla traženja

Sasvim isto kao za rječnik (vidi potpoglavlje 4.2). Mogu se upotrijebiti isti potprogrami **INSERT()**, **DELETE_MIN()**, **MAKE_NULL()**. Funkcija **EMPTY()** je trivijalna.

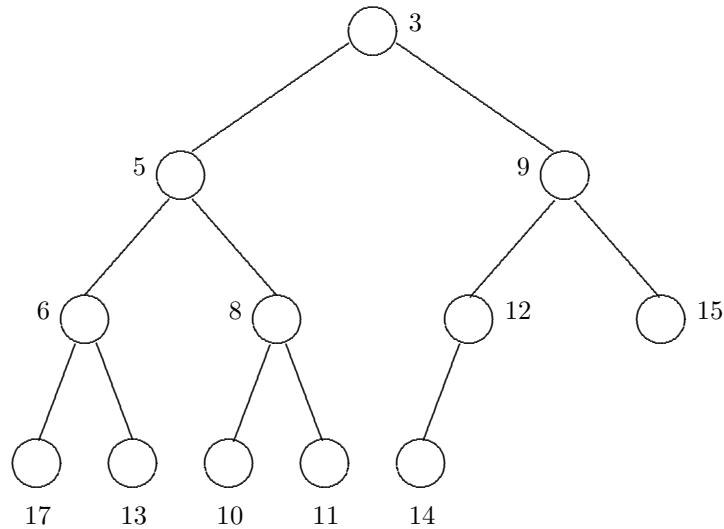
4.3.3 Implementacija prioritetnog reda pomoću hrpe

Potpuno binarno stablo T je **hrpa** (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 .

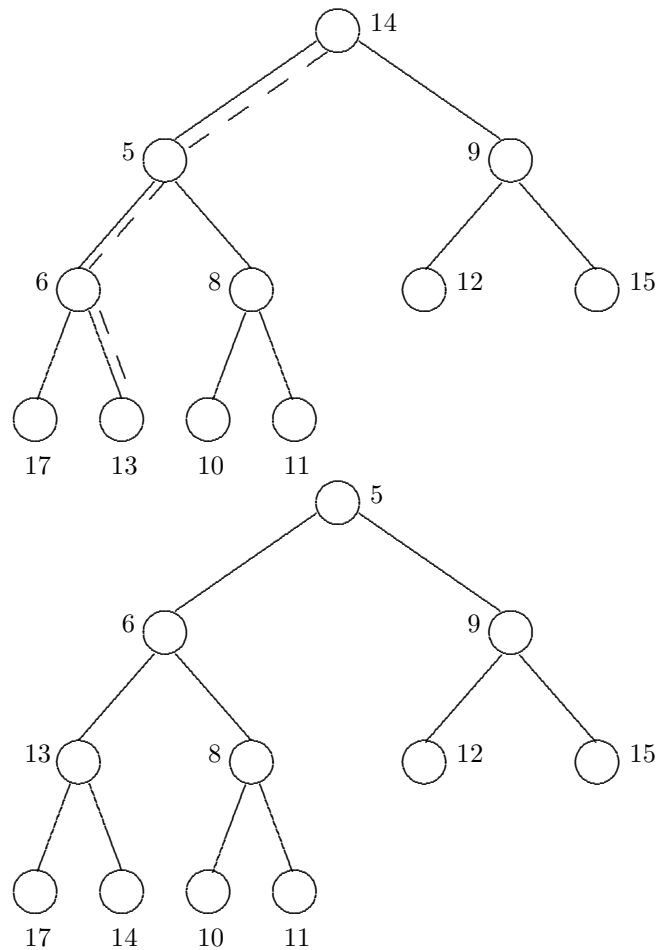
Ideja implementacije je sljedeća: prioritetni red prikazujemo hrpom. Svakom elementu reda odgovara točno jedan čvor hrpe i obratno. Element reda služi kao oznaka odgovarajućeg čvora hrpe - kažemo da je element "spremljen" u tom čvoru. Primjetimo da će svi čvorovi naše hrpe imati različite oznake, makar se to ne zahtijeva u definiciji hrpe.

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



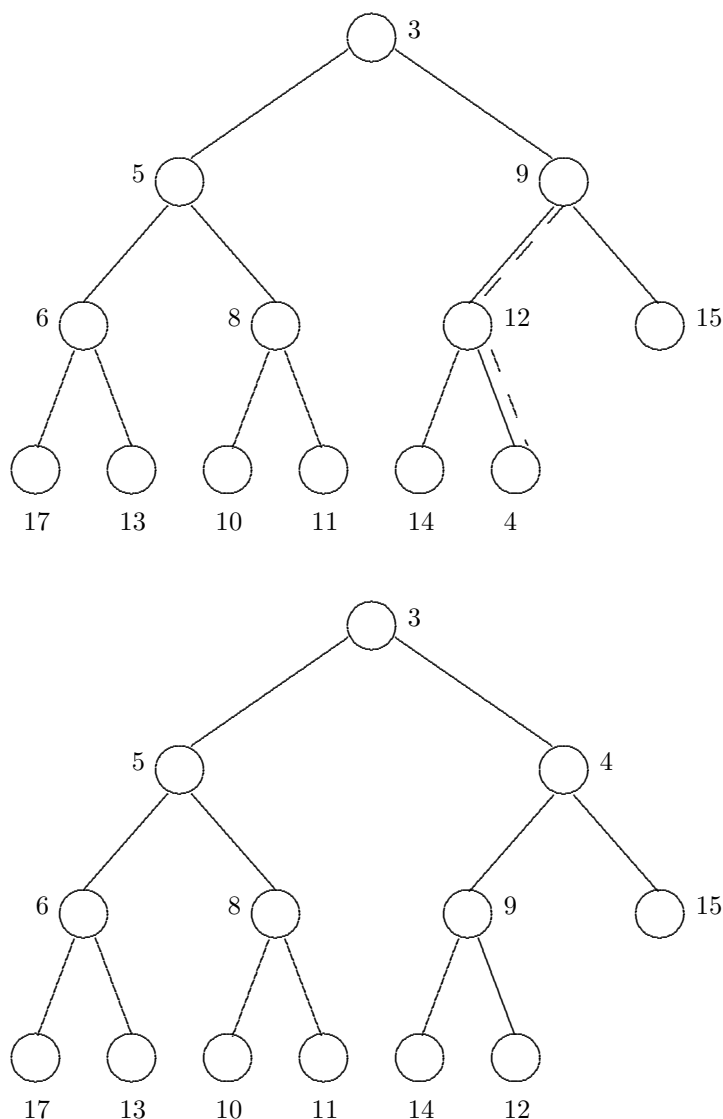
Slika 4.10 Prikaz prioritetnog reda pomoću hrpe.

Da bismo obavili operaciju `DELETE_MIN()`, vraćamo element iz korijena. Budući da korijen ne možemo samo tako izbaciti (stablo bi se raspalo), izbacujemo zadnji čvor na zadnjem nivou, a njegov element stavimo u korijen. Time se sigurno pokvarilo svojstvo hrpe. Popravlak 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 `DELETE_MIN()` na hrpu sa slike 4.10 vidi se na slici 4.11.



Slika 4.11 Prikaz operacije `DELETE_MIN()`.

Da bismo obavili operaciju `INSERT()`, stvaramo novi čvor na prvom slobodnom mjestu zadnjeg nivoa, 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.



Slika 4.12 Prikaz operacije `INSERT()`.

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 poglavlje 3.2). Dakle, potrebne su nam sljedeće definicije:

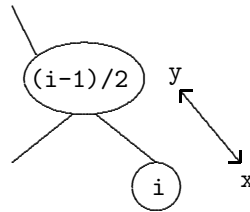
```
# define MAXSIZE ... /* dovoljno velika konstanta */
typedef struct {
    elementtype elements[MAXSIZE];
    int last;
} PRIORITY_QUEUE;
```

Funkcije `INSERT()` i `DELETE_MIN()` tada izgledaju ovako:

```

void INSERT (elementtype x, PRIORITY_QUEUE *Ap) {
    int i;
    elementtype temp;
    if ( Ap->last >= MAXSIZE-1 )
        error("priority queue is full");
    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;
        }
    }
}

```

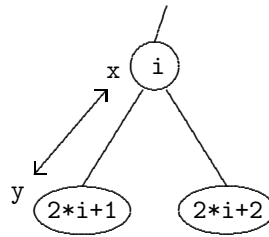


Slika 4.13 Dijagram za INSERT().

```

elementtype DELETE_MIN (PRIORITY_QUEUE *Ap) {
    int i, j;
    elementtype minel, temp;
    if ( Ap->last < 0 )
        error ("priority queue is empty");
    else {
        minel = Ap->elements[0]; /* najmanji element je u korijenu */
        Ap->elements[0] = Ap->elements[Ap->last]; /* zadnji čvor ide u korijen */
        (Ap->last)--; /* izbacujemo zadnji čvor */
        i = 0; /* i je indeks čvora u kojemu se nalazi element iz izbačenog čvora */
        while ( i <= (Ap->last-1)/2 ) { /* pomičemo 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;
            }
        }
        return(minel); /* ne treba dalje pomicati */
    }
    return(minel); /* pomicanje je došlo sve do lista */
}

```



Slika 4.14 Dijagram za DELETE_MIN().

Funkcije MAKE_NULL() i EMPTY() su trivijalne, pa ih nećemo pisati. Primjetimo da funkcija INSERT() ima jednu manjkavost: ona stvara novi čvor s elementom x čak i onda kad x već jeste u prioritetnom redu A . U primjenama obično ubacujemo samo one elemente za koje smo sigurni da se ne nalaze u prioritetnom redu. Zato ova manjkavost naše funkcije obično ne smeta.

Potprogrami INSERT() i DELETE_MIN() 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 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 operacija: MEMBER(), DELETE(), MIN(), MAX().

4.4 Preslikavanje i relacija

Često je potrebno pamtit i pridruživanja među podacima, koja se mogu opisati matematičkim pojmom “funkcije” (“preslikavanja”). Naša definicija će biti malo općenitija nego što je uobičajeno u matematici.

Preslikavanje 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, a zatim definicija apstraktnog tipa koji odgovara preslikavanju.

- Telefonski imenik je preslikavanje čiju domenu čine imena ljudi, a kodomenu telefonski brojevi. Imena se mogu ubacivati i izbacivati, a brojevi se mogu mijenjati. Slični primjeri: rječnik stranih riječi, indeks pojmova, sadržaj knjige.
- Svaki suradnik neke tvrtke opisan je zapisom u kojem se nalaze podaci poput imena i prezimena, datuma rođenja, zanimanja i sl. Zapisi (suradnici) se razlikuju na osnovu matičnog broja suradnika. Riječ je o preslikavanju čiju domenu čine matični brojevi, a kodomenu ostali podaci o suradniku.
- Matematički pojam “vektor duljine n ” može se interpretirati kao preslikavanje čija domena je skup $\{1, 2, \dots, n\}$, a kodomena (npr.) skup realnih brojeva. Slični primjeri: matrica, polje.

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.

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

`ASSIGN(&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.

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

`COMPUTE(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 osnovu d , tj. onako kao kad bi se pohranjivao sam d . To omogućuje da kasnije pronađemo par $(d, M(d))$ na osnovu zadanog d . Operacije `MAKE_NULL()`, `ASSIGN()`, `DEASSIGN()`, `COMPUTE()` iz a.t.p. MAPPING implementiraju se analogno kao `MAKE_NULL()`, `INSERT()`, `DELETE()`, `MEMBER()` iz a.t.p. DICTIONARY. Ove ideje ćemo ilustrirati sljedećim primjerima.

4.4.1 Implementacija preslikavanja pomoću hash tablice ili binarnog stabla

Promatramo preslikavanje M čija domena je tip `int`, a kodomena tip `float`. M je zadano sljedećom tablicom.

d	$M(d)$
3	2.18
8	1.12
12	8.26
15	9.31
19	4.28
25	6.64

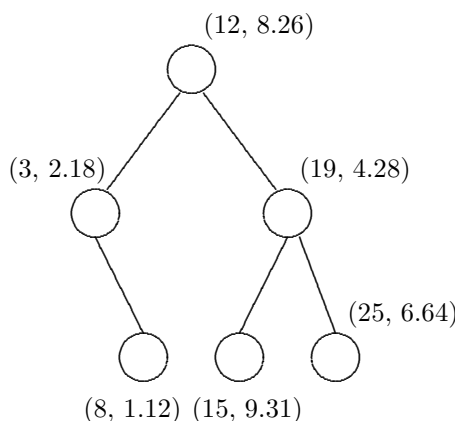
Slika 4.15 Primjer za preslikavanje M .

Za cijele brojeve d koji se ne pojavljuju u tablici $M(d)$ nije definirano. M tada shvaćamo kao skup $M = \{(3, 2.18), (8, 1.12), \dots, (25, 6.64)\}$. Taj skup shvaćamo kao rječnik i prikazujemo ga na načine opisane u poglavlju 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 $\{0, 1, 2, \dots, B-1\}$. Uređeni par $(d, M(d))$ sprema se u pretinac $h(d)$, te se traži u istom pretincu. Konkretno, uzimamo $B = 8$, $h(x) = x \% 8$. U slučaju kolizije primjenjujemo linearno haširanje.

0	8	1.12
1	25	6.64
2		
3	3	2.18
4	12	8.26
5	19	4.28
6		
7	15	9.31

Slika 4.16 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 osnovu 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 .



Slika 4.17 Preslikavanje M prikazano pomoću binarnog stabla traženja.

Često je potrebno pamtit i pridruživanja među podacima koja su općenitija od onih obuhvaćenih pojmom “preslikavanje”. Tada govorimo o relaciji.

Binarna relacija 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 $d_1 \not R d_2$.

Evo nekoliko primjera, te odgovarajuća definicija apstraktnog tipa.

- Studenti upisuju izborne kolegije. Time se uspostavlja relacija čije domene su skup studenata i skup kolegija. Možemo se pitati koje sve kolegije je upisao zadani student, te koji sve studenti su upisali zadani kolegij. Slični primjeri: ljudi i časopisi, filmovi i kina, odredišta i avionske kompanije.
- Proizvodi se sastoje od dijelova (vijaka, ploča, ležajeva, ...). Pitamo se koji su sve dijelovi potrebni za zadani proizvod, te u kojim sve proizvodima se pojavljuje zadani dio. Slični primjeri: programi sastavljeni od potprograma, cocktaili sastavljeni od pića.

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 konačni skup podataka tipa `domain2`.

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

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

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

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

`COMPUTE2(R,d1,&S2)` ... za zadani `d1` funkcija skupu `S2` pridružuje kao vrijednost $\{d2 \mid d1 R d2\}$.

`COMPUTE1(R,&S1,d2)` ... za zadani `d2` funkcija skupu `S1` pridružuje kao vrijednost $\{d1 \mid d1 R d2\}$.

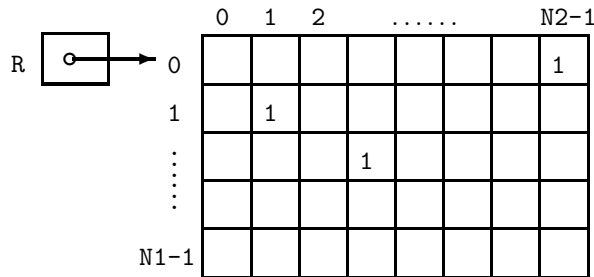
Za implementaciju relacije koriste se iste strukture podataka kao za rječnik. U skladu s našom definicijom, relaciju R pohranjujemo kao skup uređenih parova oblika (d_1, d_2) takvih da je $d_1 R d_2$. Operacije `MAKE_NULL()`, `RELATE()`, `UNRELATE()` iz a.t.p. `RELATION` implementiraju se analogno kao `MAKE_NULL()`, `INSERT()` odnosno `DELETE()` iz a.t.p. `DICTIONARY`. Da bi se mogle efikasno obavljati i operacije `COMPUTE1()`, `COMPUTE2()`, potrebne su male modifikacije i nadopune strukture. Pokazat ćemo dva primjera za takve modificirane strukture.

4.4.2 Implementacija relacije pomoću bit-matrice

Nastaje na osnovu implementacije skupa pomoću bit-vektora (vidi poglavlje 4.1). Jednodimenzionalno polje (vektor) presložimo u dvodimenzionalno (matricu). Uzimamo da je $\text{domain1} = \{0, 1, \dots, N1 - 1\}$, a $\text{domain2} = \{0, 1, \dots, N2 - 1\}$, 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 N1 x N2 */
```

(i, j) -ti bit je 1 (odnosno 0) ako i samo ako je i -ti podatak u relaciji s j -tim podatkom. Operacije `COMPUTE2(R,i,&S2)` odnosno `COMPUTE1(R,&S1,j)` svode se na čitanje i -tog retka odnosno j -tog stupca polja.



Slika 4.18 Implementacija relacije pomoću bit-matrice.

4.4.3 Implementacija relacije pomoću multi-liste

Podsijeca na implementaciju skupa pomoću vezane liste (vidi poglavlje 4.1). Umjesto jedne vezane liste sa svim uređenim parovima (d_1, d_2) , imamo mnogo malih listi koje odgovaraju rezultatima operacija `COMPUTE2()`, `COMPUTE1()`. Jedan uređeni par (d_1, d_2) prikazan je zapisom tipa:

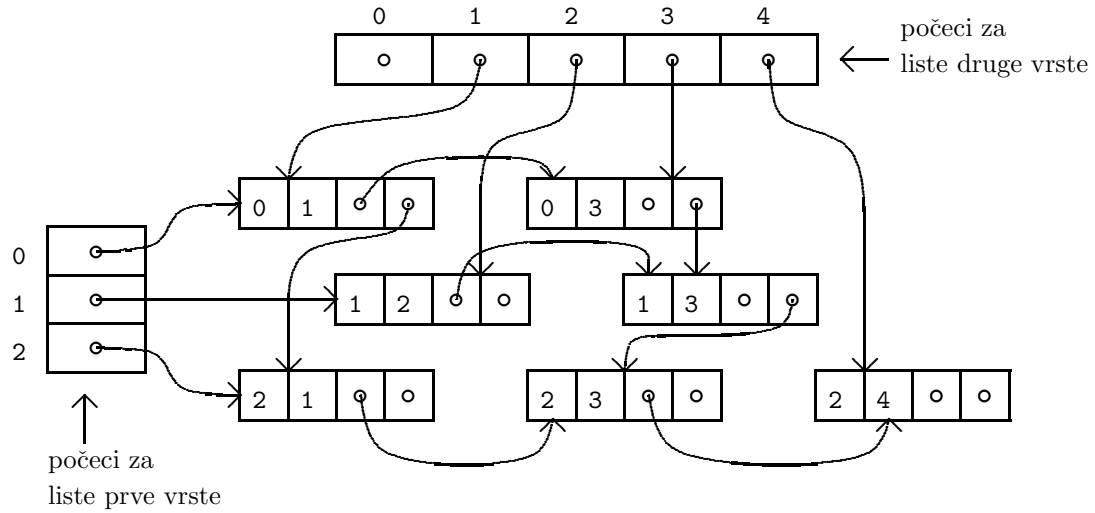
```
typedef struct cell_tag {
    domain1 element1;
    domain2 element2;
    struct cell_tag *next1;
    struct cell_tag *next2;
} celltype;
```

Jedan zapis je istovremeno uključen u:

- vezanu listu prve vrste: povezuje sve zapise s istim d_1 ;
- vezanu listu druge vrste: 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 poglavljima 4.1 i 4.2. Operacija `COMPUTE2()` (odnosno `COMPUTE1()`) 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 je prikazana relacija $R = \{(0, 1), (0, 3), (1, 2), (1, 3), (2, 1), (2, 3), (2, 4)\}$. Dva preslikavanja prikazana su kao polja pointera.



Slika 4.19 Implementacija relacije pomoću multi-liste.

5

OBLIKOVANJE ALGORITAMA

5.1 Metoda “podijeli pa vladaj”

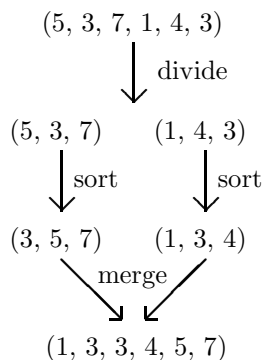
Na osnovu dugogodišnjeg iskustva, ljudi su identificirali nekoliko općenitih “tehnika” (metoda, strategija, smjernica) za oblikovanje algoritama. U ovom dijelu kolegija razmotrit ćemo najvažnije tehnike: podijeli-pa-vladaj, dinamičko programiranje, pohlepni pristup i backtracking. Kad želimo sami oblikovati 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.

“Podijeli-pa-vladaj” je možda najprimjenjivija strategija za oblikovanje algoritama. Ona se sastoji u tome da zadani problem razbijemo u nekoliko manjih (istovrsnih) problema, tako da se rješenje polaznog problema može relativno lako konstruirati iz rješenja manjih problema. Dobiveni algoritam je rekurzivan - naime svaki od manjih problema se dalje rješava na isti način, tj. razbije se u još manje probleme, itd. Mora postojati način da se problemi dovoljno male veličine riješe direktno (bez daljnjeg razbijanja). U ostatku ovog potpoglavlja slijede tri primjera algoritama oblikovanih u skladu s metodom “podijeli pa vladaj”.

5.1.1 Sortiranje sažimanjem (merge sort)

Algoritam merge sort za sortiranje liste (vidi vježbe) može se ovako tumačiti:

- što je lista dulja, to ju je teže sortirati. Zato ćemo polaznu listu razbiti na dvije manje i svaku od tih manjih listi ćemo sortirati zasebno
- velika sortirana lista se dobiva relativno jednostavnim postupkom sažimanja malih sortiranih listi.

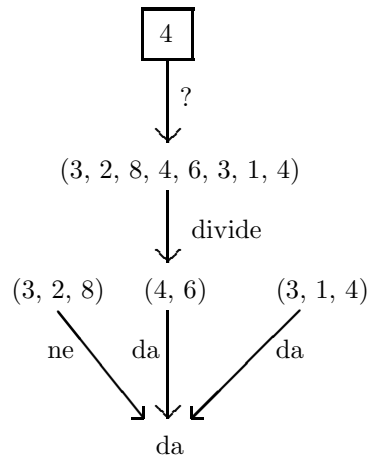


Slika 5.1 Sortiranje sažimanjem.

5.1.2 Traženje elementa u listi

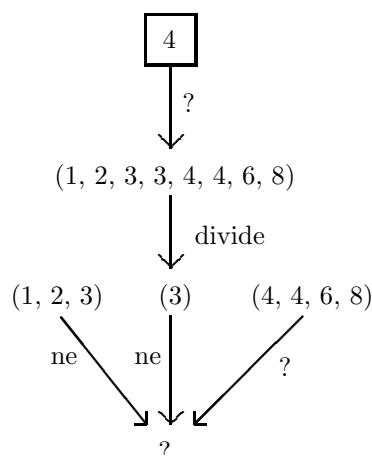
Želimo utvrditi da li u zadanoj listi postoji element sa zadanom vrijednošću. Metoda “podijeli pa vladaj” sugerira sljedeće rješenje:

- 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 se pojavljuje u velikoj listi ako i samo ako se pojavljuje u bar jednoj maloj listi. Znači, konačni odgovor se dobije tako da se parcijalni odgovori kombiniraju logičkom operacijom “ili”.



Slika 5.2 Traženje elementa u listi.

- ako je polazna lista sortirana, tada se gornji algoritam može bitno pojednostaviti. Podjelu na manje liste provodimo tako da srednja mala lista ima duljinu 1, a ostale dvije su podjednake duljine.
- jedna operacija uspoređivanja (zadana vrijednost s elementom u sredini) omogućuje da se vrijednost odmah pronađe ili da se dvije liste eliminiraju iz daljnjeg razmatranja. Dobivamo algoritam binarnog traženja (vidi pogl. 4.2).



Slika 5.3 Traženje elementa u sortiranoj listi.

5.1.3 Množenje dugačkih cijelih brojeva

Promatramo problem množenja dvaju n -bitnih cijelih brojeva X i Y . Klasični algoritam (srednja škola) zahtijeva računanje n parcijalnih produkata veličine n ; dakle njegova složenost je $\mathcal{O}(n^2)$ ukoliko

operacije sa 1 bitom smatramo osnovnima. Pristup “podijeli pa vladaaj” sugerira ovakav algoritam:

- svaki od brojeva X odnosno Y treba podijeliti na dva dijela koji su duljine $n/2$ bitova (zbog jednostavnosti uzimamo da je n potencija od 2):

$$\begin{array}{lcl} X \dots & \boxed{\begin{array}{|c|c|} \hline A & B \\ \hline \end{array}} & X = A2^{n/2} + B \\ Y \dots & \boxed{\begin{array}{|c|c|} \hline C & D \\ \hline \end{array}} & Y = C2^{n/2} + D \end{array}$$

Slika 5.4 Prikaz brojeva duljine n bitova.

- produkt od X i Y se tada može ovako izraziti:

$$XY = AC2^n + (AD + BC)2^{\frac{n}{2}} + BD.$$

Znači, jedno množenje “velikih” n -bitnih brojeva reducirali smo 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 zahtijevaju $\mathcal{O}(n)$ koraka, možemo pisati sljedeću rekurziju za ukupan broj koraka $T(n)$ koje implicira gornja formula:

$$\begin{aligned} T(1) &= 1, \\ T(n) &= 4T(n/2) + cn \quad (\text{ovdje je } c \text{ neka konstanta}). \end{aligned}$$

Lako se vidi da je rješenje rekurzije $T(n) = \mathcal{O}(n^2)$. Znači, naš algoritam dobiven metodom “podijeli-pa-vladaaj” 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 manjom složenošću. 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-vladaaj” je asimptotski brža 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 cijelih brojeva: npr svaku cjelobrojnu varijablu mogli bi zamijeniti dugačkim poljem bitova. Također bi sami morali 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 najviše n bitova. */
    /* Funkcija vraća X*Y. Pretpostavljamo da je n potencija od 2. */
    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 */
}
}

```

Primjetimo na kraju da je za algoritme tipa “podijeli-pa-vladaj” važno da potproblemi budu dobro izbalansirani (sličnih veličina) - to će naime osigurati efikasnost. Npr., insertion sort se može smatrati algoritmom tipa “podijeli-pa-vladaj”, gdje se polazna lista duljine n (koju treba sortirati) podijeli na dvije nejednake liste duljina $n-1$ odnosno 1. Rezultirajuće vrijeme sortiranja je $\mathcal{O}(n^2)$. S druge strane, merge sort je također algoritam tipa “podijeli-pa-vladaj”, ali se polazna lista duljine n dijeli na dvije liste duljine $\approx n/2$. Vrijeme sortiranja za merge sort je $\mathcal{O}(n \log n)$.

5.2 Dinamičko programiranje

Metoda “podijeli pa vladaj” koji put rezultira neefikasnim algoritmom. Naime, dešava se da broj potproblema koje treba riješiti raste eksponencijalno s veličinom zadanog problema. Pritom ukupan broj različitih potproblema možda i nije tako velik, no jedan te isti potproblem se pojavljuje na mnogo mjesta u rekurziji (pa ga uvijek iznova rješavamo).

U ovakvim situacijama preporuča se **dinamičko programiranje**. Metoda zahtijeva da se svi potproblemi redom riješe, te da se rješenja spremaju 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 problema prema manjima), algoritmi dinamičkog programiranja idu s “dna prema gore” (od manjih prema većem). Znači, prvo se u tabelu unose rješenja za probleme najmanje veličine, zatim rješenja za malo veće probleme, itd., sve dok se ne dosegne veličina zadanog problema. Važan je redoslijed ispunjavanja tabele.

Postupak dinamičkog programiranja koji put zahtijeva da riješimo i neke potprobleme koji nam na kraju neće biti potrebni za rješenje zadanog problema. To se još uvijek više isplati nego rješavanje istih potproblema mnogo puta.

5.2.1 Problem određivanja šanse za pobjedu u sportskom nadmetanju

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 (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 su očigledna. 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 rekursivno 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. Donji red je ispunjen nulama, a desni stupac jedinicama. Prethodna rekursivna formula kaže da se bilo koji od preostalih elemenata tabele može dobiti kao aritmetička sredina elemenata ispod i elemenata udesno. Znači, dobar način za popunjavanje tabele je da se ide po dijagonalama počevši od donjeg desnog ugla.

	1/2	21/32	13/16	15/16	1	4
	11/32	1/2	11/16	7/8	1	3
	3/16	5/16	1/2	3/4	1	2
	1/16	1/8	1/4	1/2	1	1
	0	0	0	0		0
	4	3	2	1	0	

← i

j ↑

Slika 5.5 Šanse za pobjedu u sportskom nadmetanju.

Algoritam se može zapisati sljedećom C funkcijom, koji radi nad globalnim 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];
}
```

Unutrašnja petlja u gornjem potprogramu traje $\mathcal{O}(s)$, a vanjska petlja se ponavlja za $s = 1, 2, \dots, (i+j)$. Dakle, računanje $P(i, j)$ sve skupa traje $\mathcal{O}\left(\sum_{s=1}^{i+j} s\right) = \mathcal{O}((i+j)^2)$.

5.2.2 Rješavanje 0/1 problema ranca

Zadano je n predmeta O_1, O_2, \dots, O_n i ranac. Predmet O_i ima težinu w_i i vrijednost (cijenu) p_i . Kapacitet ranca je c težinskih jedinica. Pitamo se: koje predmete treba staviti u ranac tako da ukupna težina ne prijeđe c , te da ukupna vrijednost bude maksimalna. Uzimamo da su w_i ($i = 1, 2, \dots, n$) i c pozitivni cijeli brojevi.

Ovaj problem bi se mogao riješiti algoritmom tipa “podijeli pa vladaj” koji bi generirao sve moguće podskupove skupa $\{O_1, O_2, \dots, O_n\}$ i izabrao onaj s najvećom vrijednošću uz dopustivu težinu. No, takav algoritam bi očito 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 $\{O_1, O_2, \dots, O_i\}$ uz kapacitet j . Prilikom postizavanja

M_{ij} , predmet O_i je stavljen u ranac ili nije. Ukoliko O_i nije stavljen, tada je $M_{ij} = M_{i-1,j}$. Ukoliko je O_i stavljen, tada prije izabrani predmeti predstavljaju optimalni izbor iz skupa $\{O_1, O_2, \dots, O_{i-1}\}$ 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 $(n+1) \times (c+1)$. Redoslijed računanja je redak-po-redak (j se brže mijenja nego i). Vrijeme izvršavanja algoritma je $\mathcal{O}(nc)$.

Algoritam ispunjavanja tabele zapisat ćemo sljedećom funkcijom. Koriste se globalna polja sljedećeg oblika s očiglednim značenjem.

```
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 zero_one_knapsack (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 sljedeću tabelu

$i \backslash j$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	7	7	8	8
3	0	0	1	7	8	8	9

Znači, maksimalna vrijednost koja se može nositi u rancu je $M_{3,6} = 9$. Ona se postiže izborom prvog i trećeg predmeta. Ukupna težina je $2 + 4 = 6$.

Zapisana verzija algoritma (tabele) zapravo daje samo maksimalnu vrijednost koja se može ponijeti u rancu, a ne i optimalni izbor predmeta. Da bi 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.

5.3 “Pohlepni” pristup

Zamislamo 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 je bio da izaberemo najveći apoen koji ne prelazi 62 kune (to je 50 kuna), stavljamo ga u listu za vraćanje, te 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. Rješenje 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.

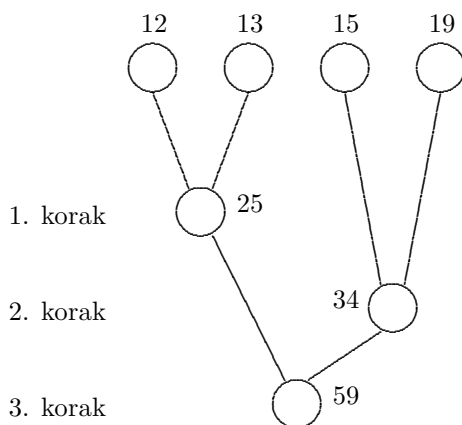
Pohlepni pristup ne mora uvijek dovesti do optimalnog rješenja. Algoritam vraćanja ostatka je funkcionirao samo zahvaljujući specijalnim svojstvima raspoloživih apoena. Ako bi apoeni bili 11, 5 i 1 kuna, a trebali bismo vratiti 15 kuna, tada bi pohlepni algoritam najprije izabrao 11 kuna, a zatim bi 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 (to onda treba dokazati). Postoje također problemi kod kojih pohlepni algoritam nije sasvim korektan, ali predstavlja dobru heuristiku (to onda treba eksperimentalno provjeriti).

5.3.1 Optimalni plan sažimanja sortiranih listi

Imamo n sortiranih listi s duljinama w_1, w_2, \dots, w_n . Sažimljemo ih 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 je bio 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 treba izvesti tako da imamo najmanje posla u tom koraku.

Nije sasvim očigledno da ćemo ovom strategijom zaista postići najmanji ukupni broj operacija. No, može se dokazati (metodom binarnih stabala) da ipak hoćemo.



Slika 5.6 Optimalno sažimanje sortiranih listi.

5.3.2 Kontinuirani problem ranca

Riječ je o problemu koji je sličan 0/1 rancu (vidi pogl. 5.2). No sada se predmeti koje stavljamo u ranac mogu "rezati", tj. ako ne stane cijeli predmet tada možemo staviti samo njegov dio. Stroga formulacija problema glasi:

Zadan je prirodni broj n i pozitivni realni brojevi c, w_i ($i = 1, 2, \dots, n$), p_i ($i = 1, 2, \dots, n$).

Traže se realni brojevi x_i ($i = 1, 2, \dots, n$) takvi da $\sum_{i=1}^n p_i x_i \rightarrow \max$, uz ograničenja $\sum_{i=1}^n w_i x_i \leq c, 0 \leq x_i \leq 1$ ($i = 1, 2, \dots, n$).

Interpretacija zadanih podataka je ista kao u potpoglavlju 5.2. 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 spremati što veći dio predmeta. Dobivamo funkciju u jeziku C koji radi nad sljedećim globalnim poljima s očiglednim značenjem.

```
#define MAXLENGTH ... /* dovoljno velika konstanta */
float w[MAXLENGTH], p[MAXLENGTH], x[MAXLENGTH];
```

Pretpostavljamo da su podaci u poljima $w[]$ i $p[]$ već uređeni tako da je $p[i]/w[i] \geq p[i+1]/w[i+1]$.

```
void cont_knapsack (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;
}
```

Promatrajmo rad našeg pohlepnog algoritma na podacima iz potpoglavlja 5.2, 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 2. predmet, a idući po profitabilnosti je 3. predmet. Zato algoritam smješta u ranac cijeli 2. predmet, a ostatak kapaciteta popunjava s $3/4$ trećeg predmeta. Znači rješenje je $(x_1, x_2, x_3) = (0, 1, 3/4)$, a maksimalna vrijednost ranca je 13.

Tvrđnja: opisani pohlepni algoritam zaista daje korektno (optimalno) rješenje kontinuiranog problema ranca.

Dokaz: Možemo uzeti da je $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \geq \frac{p_n}{w_n}$. Neka je $X = (x_1, x_2, \dots, x_n)$ 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 $Y = (y_1, y_2, \dots, y_n)$ 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 $x_k \neq y_k$. Tvrđimo 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}, \dots, y_n) koliko treba da ukupna težina ostane ista. Time dobivamo novo rješenje $Z = (z_1, z_2, \dots, z_n)$ 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:

$$\begin{aligned} \sum_{i=1}^n p_i z_i &= \sum_{i=1}^n p_i y_i + (z_k - y_k) p_k - \sum_{i=k+1}^n (y_i - z_i) p_i \\ &= \sum_{i=1}^n p_i y_i + (z_k - y_k) w_k \frac{p_k}{w_k} - \sum_{i=k+1}^n (y_i - z_i) w_i \frac{p_i}{w_i} \end{aligned}$$

$$\begin{aligned}
&= \dots \text{zbog svojstva } \frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \dots \\
&\geq \sum_{i=1}^n p_i y_i + \left[(z_k - y_k) w_k - \sum_{i=k+1}^n (y_i - z_i) w_i \right] \frac{p_k}{w_k} \\
&\geq \sum_{i=1}^n p_i y_i + 0 \cdot \frac{p_k}{w_k} \\
&= \sum_{i=1}^n p_i y_i.
\end{aligned}$$

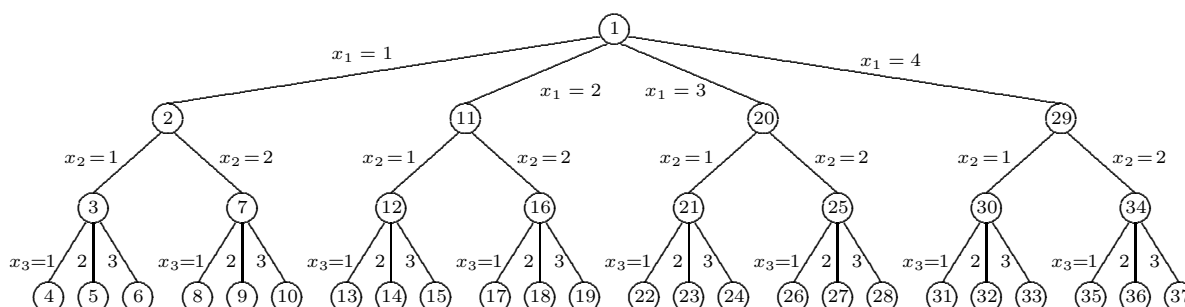
Znači, našli smo novo optimalno rješenje Z koje se “bolje poklapa” s X nego što se Y poklapa s X . Iteriranjem ove konstrukcije prije ili kasnije ćemo doći do optimalnog rješenja koje se u potpunosti poklapa s X . Znači, X je optimalan. Time je dokaz teorema završen.

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 poglavlja 5.2, 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 bi zahtijevao da se u ranac najprije stavi “najprofitabilniji” 2. predmet. Nakon toga 3. predmet (idući po profitabilnosti) više ne bi stao, pa bi (zbog nemogućnosti rezanja) mogli dodati još samo 1. predmet. Znači, dobili bi suboptimalno rješenje s vrijednošću 8. U poglavlju 5.2 smo pokazali da optimalno rješenje ima vrijednost 9.

5.4 Backtracking

Backtracking je vrlo općenita metoda koja se primjenjuje za teške kombinatorne probleme. Rješenje problema se traži sistematskim ispitivanjem svih mogućnosti za konstrukciju tog rješenja. Metoda zahtijeva da se traženo rješenje izrazi kao n -torka oblika (x_1, x_2, \dots, x_n) , gdje je x_i element nekog konačnog skupa S_i . Kartezijev produkt $S_1 \times S_2 \times \dots \times S_n$ 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 $n = 3$, $S_1 = \{1, 2, 3, 4\}$, $S_2 = \{1, 2\}$, $S_3 = \{1, 2, 3\}$.



Slika 5.7 Stablo rješenja.

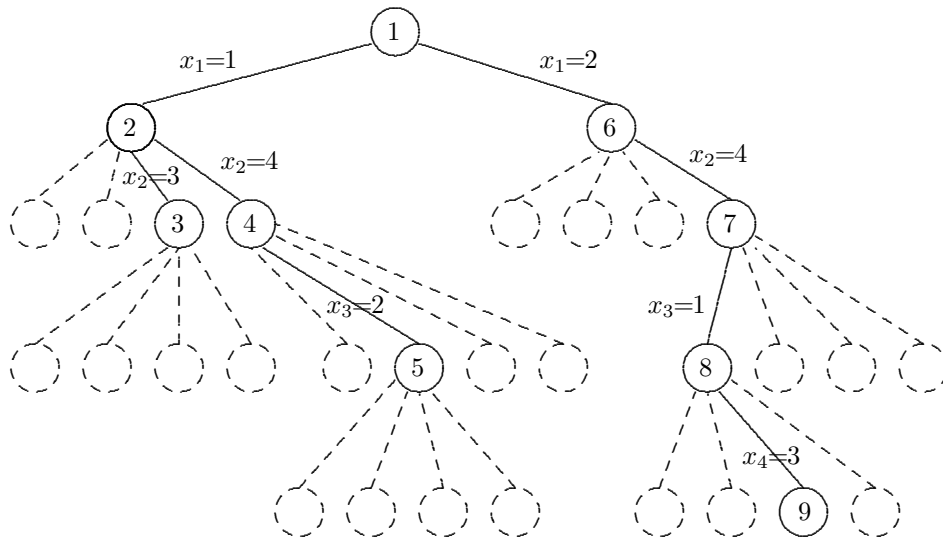
Backtracking je u osnovi rekurzivni algoritam koji se sastoji od simultanog generiranja i ispitivanja čvorova u stablu rješenja. Čvorovi se stavljaju na stog. Jedan korak algoritma sastoji se od toga da se uzme čvor s vrha stoga, te da se provjeri da li taj čvor predstavlja rješenje problema. Ukoliko čvor predstavlja rješenje, tada se poduzme odgovarajuća akcija. 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. Redoslijed obrađivanja čvorova (skidanja sa stoga) vidi se na prethodnoj slici.

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, \dots, 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, \dots, x_i fiksirano na određeni način. Ukoliko se već na osnovu 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 prethodne slike 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.

5.4.1 Problem n kraljica

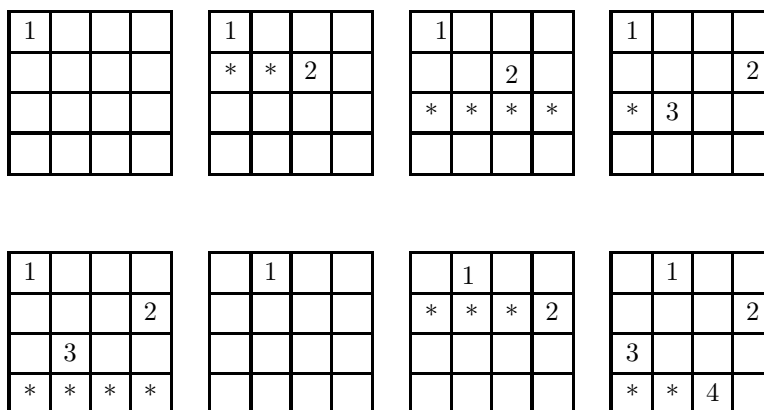
Na šahovsku ploču dimenzije $n \times n$ treba postaviti n kraljica tako da se one međusobno ne napadaju. Očito svaka kraljica mora biti u posebnom retku ploče. Zbog toga možemo uzeti da je i -ta kraljica u i -tom retku. Rješenje problema se može predložiti kao n -torka (x_1, x_2, \dots, x_n) , gdje je x_i indeks stupca u kojem se nalazi i -ta kraljica. Znači $S_i = \{1, 2, \dots, n\}$ za sve i . Broj n -torki u prostoru rješenja je n^n . Ograničenja koja rješenje (x_1, x_2, \dots, 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 generira sljedeće stablo. Rad je ovdje prekinut čim je pronađeno prvo rješenje. Crtkano su označeni čvorovi koje je algoritam u postupku generiranja odmah poništio (jer krše ograničenja).



Slika 5.8 Stablo rješenja za problem n kraljica.

Sljedeća slika ilustrira ove iste korake algoritma kao pomake figura na šahovskoj ploči. Zvezdice označavaju mjesta gdje je algoritam pokušao smjestiti kraljicu, ali je odmah odustao zbog napada druge kraljice.

Slika 5.9 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, \dots, 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 $1, 2, \dots, (k-1)$ -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;
    x[1]=0; k=1; /* 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 */
    }
}
```

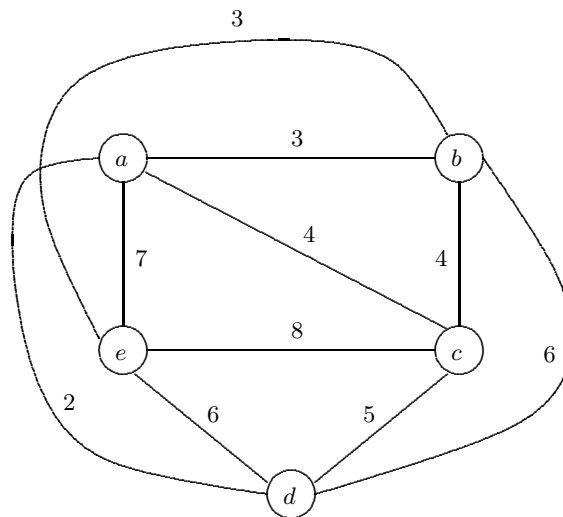
Backtracking se često koristi za probleme optimizacije. Dakle, od svih rješenja tražimo samo ono koje je optimalno u nekom smislu. Optimalnost se mjeri **funkcijom cilja** koju treba minimizirati odnosno maksimizirati. Ovisno o primjeni, ta funkcija cilja interpretira se kao cijena, trošak, dobitak, zarada, itd. Vidjeli smo da u stablu rješenja uvijek treba rezati one grane koje ne vode do rješenja. Kod problema optimizacije mogu se također rezati i one grane koje ne vode do boljeg rješenja (u odnosu na ono koje već imamo). Takva varijanta backtracking algoritma obično se naziva **branch-and-bound**.

Uzmimo zbog konkretnosti da rješavamo problem minimizacije. Funkcija cilja tada se interpretira npr. kao cijena koju želimo minimizirati. Za realizaciju branch-and-bound algoritma potreban nam je postupak određivanja donje ograde za cijenu po svim rješenjima iz zadanog pod-stabla. Ukoliko 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.

5.4.2 Problem trgovačkog putnika

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 zadaje jedan konkretan primjer problema trgovačkog putnika za $n = 5$ i $m = 10$.



Slika 5.10 Problem trgovačkog putnika.

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 se može predložiti kao m -torka (x_1, x_2, \dots, 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 5.11, dakle kao binarno. 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, \dots, 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:

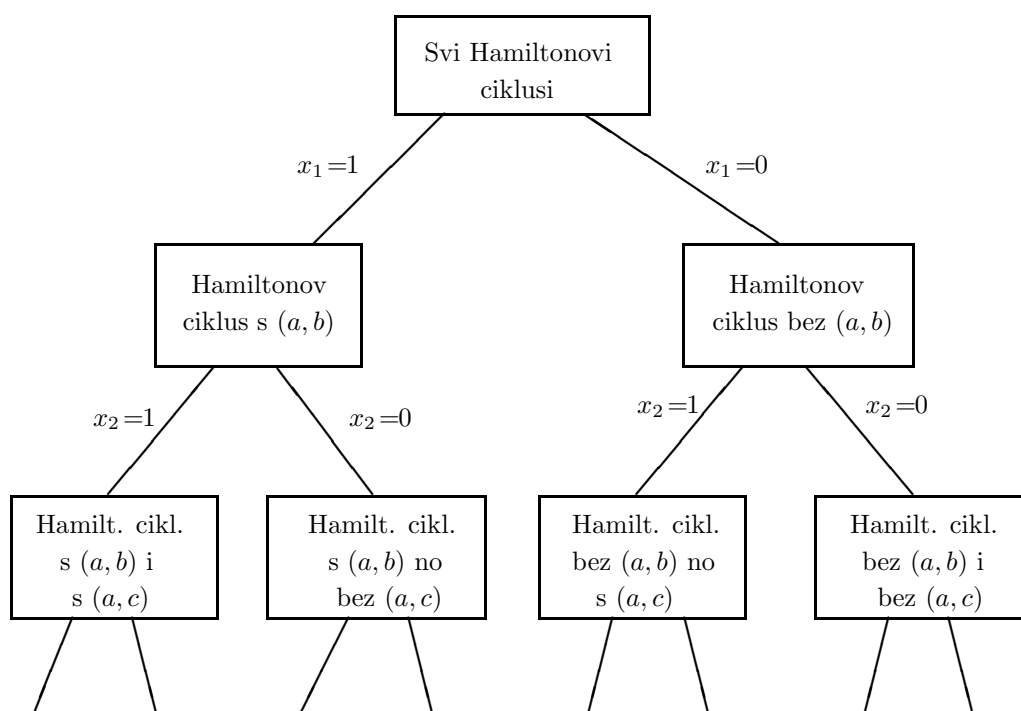
- Ako bi isključenje brida (x, y) učinilo nemogućim da x (odnosno y) ima bar dva incidentna brida u konfiguraciji, tada (x, y) mora biti uključen.
- Ako bi uključivanje brida (x, y) uzrokovalo da x (ili y) ima više od dva incidentna brida u konfiguraciji, tada (x, y) mora biti isključen.
- Ako bi uključivanje brida (x, y) zatvorilo ne-Hamiltonov ciklus u konfiguraciji, tada (x, y) mora biti isključen.

Primijetimo da cijenu zadanog Hamiltonovog ciklusa možemo računati kao sumu, tako da za svaki vrh grafa pribrojimo cijenu dvaju njemu incidentnih bridova iz ciklusa. 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” sume, dobivene za pojedine vrhove, zbrojimo zajedno u “veliku” sumu;
- “veliku” sumu podijelimo s 2.

Npr. za graf sa slike 5.10 donja ograda bi izašla $(5 + 6 + 8 + 7 + 9)/2 = 17.5 \rightarrow 18$ (zbog cjelobrojnosti).



Slika 5.11 Stablo rješenja za problem trgovačkog putnika.

Ukoliko želimo precizniju donju ogradu za cijenu Hamiltonovog ciklusa koji pripada određenom podstablu iz stabla rješenja, tada postupamo slično kao gore, no uzimamo u obzir već fiksiranu uključenost odnosno isključenost nekih bridova:

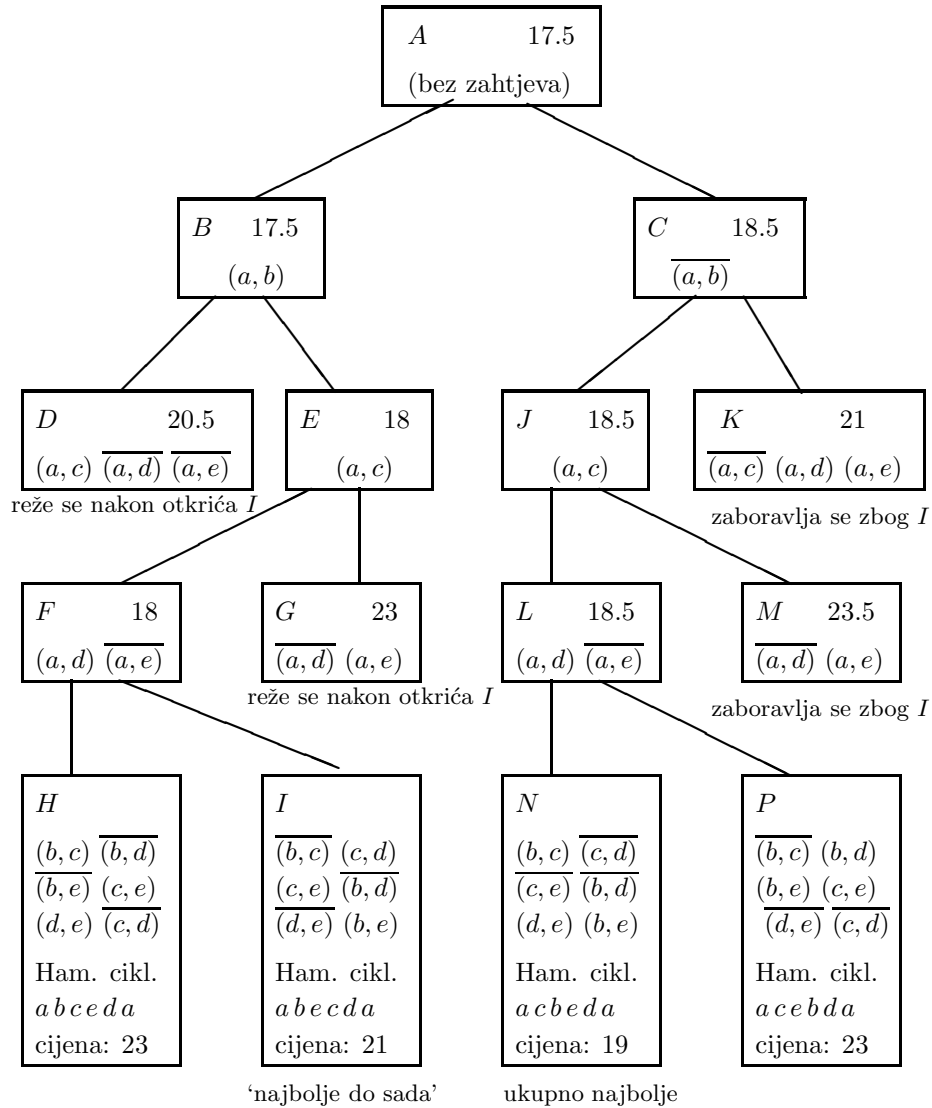
- ako je brid (x, y) uključen, tada njegovu cijenu treba uračunati u “male” sume vrhova x i y , bez obzira da li je on jeftin ili nije;
- ako je brid (x, y) isključen, tada njegovu cijenu ne smijemo uračunati u “malu” sumu vrha x odnosno y .

Npr. za podstablo gdje je (a, b) isključen i (a, c) uključen, donja ograda bi izašla $(6 + 7 + 8 + 7 + 9)/2 = 18.5 \rightarrow 19$.

Na osnovu svih ovih ideja moguće je konstruirati backtracking algoritam za rješavanje problema trgovačkog putnika, štoviše njegovu branch-and-bound varijantu. Za konkretni graf sa slike 5.10, algoritam generira stablo prikazano slikom 5.12. Pronalazi se optimalni Hamiltonov ciklus (a, c, b, e, d, a) sa cijenom 19.

U svaki čvor na slici 5.12 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 svakom nivou stabla uvodi se zapravo samo jedan novi zahtjev, ostali upisani

zahtjevi su posljedica naših pravila. Npr. u čvoru D se originalno 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 u redosljedu njihovog nastajanja. Taj redosljed je posljedica naše heuristike da se prije razgrađuje onaj brat koji ima manju donju ogradu.



Slika 5.12 Stablo generirano backtracking algoritmom za rješavanje problema trgovačkog putnika.

Literatura

- [1] Aho A.V., Hopcroft J.E., Ulman J.D., *Data Structures and Algorithms*, 2nd edition. Addison-Wesley, Reading MA, 1987.
- [2] Horowitz E., Sahni S., Anderson-Freed S., *Fundamentals of Data Structures in C*. W.H. Freeman & Co., New York, 1992.
- [3] Horowitz E., Sahni S., Rajasekaran S., *Computer Algorithms / C++*. Computer Science Press, New York, 1997.
- [4] Kruse R.L., Leung B.P., Tondo, C.L., *Data Structures and Program Design in C*, 2nd edition. Prentice-Hall, Englewood Cliffs NJ, 1996.
- [5] Preiss B.R., *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*. John Wiley & Sons, New York, 1999.
- [6] Lipschutz S., *Data Structures*. Schaum's Outline Series, McGraw-Hill, New York, 1986.
- [7] Goodrich M.T., Tamassia R., *Algorithm Design - Foundations, Analysis, and Internet Examples*. John Wiley & Sons, New York, 2002.
- [8] Loudon K., *Mastering Algorithms with C*. O'Reilly, Sebastopol CA, 1999.