SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Filip Višić

NAPREDNE STRUKTURE PODATAKA

ZAVRŠNI RAD

Varaždin, 2011.

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Filip Višić
Redoviti student
Broj indeksa: 32489/00.
Smjer: Informacijski sustavi
Preddiplomski studij

NAPREDNE STRUKTURE PODATAKA

ZAVRŠNI RAD

Mentor:
Tihomir Orehovački, mag. inf.

Varaždin, rujan 2011.

Sadržaj

1. Uvod

Za rješavanje problema u programiranju koristimo se pomoćnim postupkom algoritmom. Algoritam treba zadovoljavati tri osnovna uvjeta da bi bio ispravan. Mora biti konačan, uz iste ulazne uvjete uvijek mora dati isti rezultat te se mora prilagoditi računalu na kojem se izvršava. Kako bi algoritam bio što efikasniji i problem se lakše riješio, potrebno je napisati i prilagođenu strukturu podataka. Ako postoji više algoritama i različitih struktura podataka koji rješavaju isti učestali problem, potrebno je odlučiti kako izabrati i koristiti onaj „bolji“.

Određivanje kakvoće algoritma i primjerene strukture podataka olakšati će rješavanje specifičnog problema. Za mjeru kakvoće algoritma koristit ćemo složenost - O notaciju. Ona mjeri količinu resursa koje algoritmu treba da bi se riješio problem. Postoje dvije mjere koje se najčešće koriste za određivanje kakvoće algoritma, a to su vrijeme potrebno za izvršenje algoritma i prostor potreban za pohranjivanje ulaznih i izlaznih rezultata [3]. Svaki programski jezik podržava neke tipove podataka kao što su brojevni, slovni, logički. Prema potrebi programer može osmisliti vlastite tipove podataka, a takve tipove zovemo apstraktni tipovi podataka (ATP). Svaki ATP na računalu može se definirati na različite načine. Neke implementacije ATP pogodnije su za pojedine operacije primjerice čitanje, a druge za pretraživanje ili upisivanje novih podataka. Tako da ne postoji savršena izvedba ATP nego se primjenjuje ona koja ovisi o broju operacije koje se najčešće izvodi i daje najbolje rezultate.

Napredne strukture podataka omogućuju nam da se bavimo rješavanjem puno složenijim problemima. Zahvaljujući strukturama poput hrpa i grafa danas možemo lakše rješavati ozbiljnije različite probleme. Najpoznatiji su: problem naprtnjače, problem rasporeda, jednosmjernih ulica, bojanje grafova, problem trgovačkog putnika, turnir, protok u mrežama, neuronske mreže, linearno programiranje, prepoznavanje uzoraka - biometrija, AI, određivanje najkraćeg puta itd.. Kao što vidimo primjena naprednih struktura podataka je jako velika. Problematika svih ATP je u brzini pretraživanja, ubacivanjem novih ili mijenjanjem postojećih elemenata ponekad i u limitu maksimalnom broja elementa te njihovom brisanju. Zato s boljom organizacijom podataka u memoriji nabrojane napredne strukture podataka možda u početku nisu toliko zanimljive dok ih se ne poveže s odgovarajućim algoritmima gdje daju svoju maksimalnu primjenu. Iako ih nazivamo naprednim, njihova uloga i korištenje je sve češće pa stoga gotovo svaki današnji programeri direktno ili indirektno koristi neke od

naprednih struktura podataka. Za ljubitelje C/C++ programskih jezika razvijen je STL1 [15]. STL je gotov predložak koji ima implementiran kroz svoja zaglavljima sve važnije napredne i osnovne ATP. Također ima implementirane sve važnije algoritme koji ih mogu iskoristiti. Na taj način vrlo brzo i bez pogrešaka danas programeri mogu lako koristiti napredne stvari i rješavati konkretne probleme. Slično je i u ostalim programerskim jezicima.

Za ovu temu odlučio sam se upravo zbog toga što se bavim programiranjem. Iskoristio sam svoje iskustvo i znanje u pisanje ove teme.

Cilj ovog rada je pokazati osnove funkcioniranja naprednih ATP, njihove prednosti i nedostatke. Obradit će se prioritetni red, hrpa, graf i skup.

[1] STL Standard Template Library

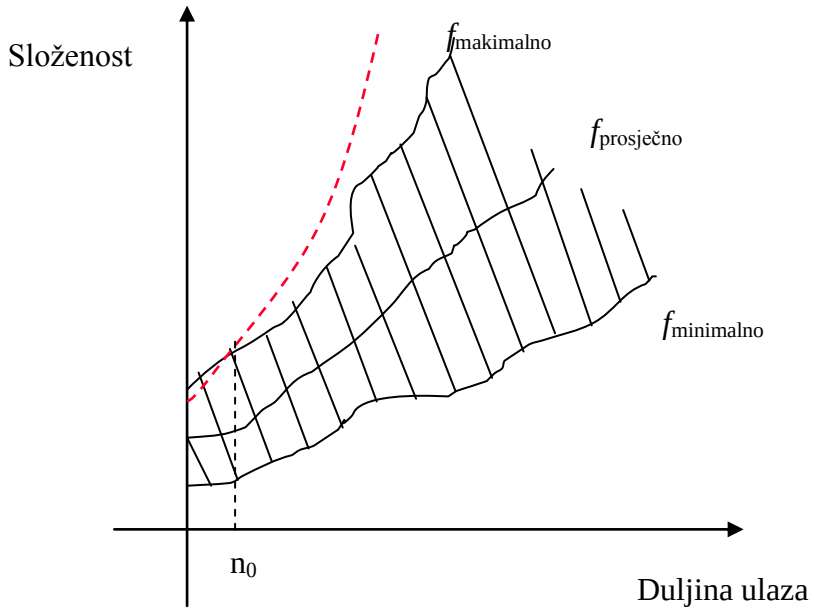
1.1. Osnovni pojmovi

1.1.1. Složenost algoritama

Složenost algoritama dijeli se na vremensku složenost2 i prostornu složenost3. Vremenskoj složenosti se pridaje znatno više značaja nego prostornoj. Razlog tome je iskustvo u razvoju algoritama, koje je pokazalo da je zauzeće memorije znatno manji problem od postizanja prihvatljivog vremena obrade. Analiza brzine izvršenja programa izračunava se na osnovu ukupnog broja operacija koje dominantno troše procesorsko vrijeme. Jedan od problema koji se javlja kod određivanja vremena složenosti algoritama jest kako naći mjeru koja neće ovisiti o brzini računala na kojem se algoritam izvodi, već samo o samom algoritmu i ATP koja se koristi.

„Vremenska složenost se izražava brojem elementarnih operacija koje algoritam mora izvesti da bi izračunao rješenje, a ne vremenom potrebnim da algoritam izračuna rezultat.“ [3]

Drugi problem je što u većini algoritama ulazni parametri nisu fiksni. Rješenje problema različitih dimenzija zahtijevat će različit broj elementarnih operacija za obradu. Međutim, masovni problem može imati različite instance iste duljine, čija vremenska složenost ne mora biti jednaka te složenost algoritma nije funkcija veličine ulaza slika 1 [3] .



Slika 1. Predstavlja grafički prikaz ovisnosti složenosti o duljini ulaza [3]

[2] engl. time complexity
[3] engl. space complexity

Ocjena složenosti povezuje se uz određenu instancu problema, a najčešće se koristi ocjena složenosti najboljeg, najgoreg i prosječnog slučaja. Odabirom jedne od ovih složenosti možemo dobiti funkciju složenosti algoritama u ovisnosti o duljini ulaza. Primjer pokazuje da se točan broj operacija za složenije algoritme vrlo teško može prebrojiti.

Red veličine složenosti potreban je za dobivanje ocjene složenosti, a složenost prikazujemo kao aproksimativnu vrijednost, a ne kao neki broj.

Razlika između algoritama bit će u tome koliko brzo će složenost algoritma rasti u ovisnosti porasta veličine ulaza. Zbog toga se uvodi notacija **O** koja će opisivati rast funkcije složenosti povećanjem nezavisne varijable.

U izračunavanju složenosti najčešće se dozvoljava da za male n-ove ova aproksimacija ne vrijedi, jer se pretpostavlja da će svi algoritmi raditi zadovoljavajuće za male n-ove te da će veći problemi nastati kad veličina ulaza poraste. Nas zapravo ne zanima stvarni iznos funkcije složenosti, već samo koliko brzo ta funkcija raste. **O** notaciju prvi je spomenuo Paul Backmann u svojoj knjizi "Analytische Zahlentheorie" (1894).

Definirane su asimptotske ocjene funkcija složenosti algoritma. [3]

Zadane su funkcije $f, g : R \to R$

1. Kažemo da je $f = O(g)$ ako postoji $n_0 \in N$ i $c \in R^+$ takvi da za svaki $n \geq n_0$ vrijedi $|f(n)| \leq c \cdot |g(n)|$. Funkcija f ne raste brže od funkcije g. $f(n) = \mathrm{O}(g(n))$.

2. Kažemo da je $f = \Omega(g)$ ako postoji $n_0 \in N$ i $c \in R^+$ takvi da za svaki $n \geq n_0$ vrijedi $|f(n)| \geq c \cdot |g(n)|$. Funkcija f ne raste sporije od funkcije g. $f(n) = \Omega(g(n))$

3. Kažemo da je $f = o(g)$ ako postoji $n_0 \in N$ i $c \in R^+$ takvi da za svaki $n \geq n_0$ vrijedi $|f(n)| < c \cdot |g(n)|$. Funkcija f raste sporije od funkcije g. $f(n) = o(g(n))$

4. Kažemo da je $f = \omega(g)$ ako postoji $n_0 \in N$ i $c \in R^+$ takvi da za svaki $n \geq n_0$ vrijedi $|f(n)| > c \cdot |g(n)|$. Funkcija f raste brže od funkcije g. $f(n) = \omega(g(n))$

5. Kažemo da je $f = \Theta(g)$ ako postoji $n_0 \in N$ i $c_1, c_2 \in R^+$ takvi da za svaki $n \geq n_0$ vrijedi $c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|$. Funkcija f i g rastu jednako brzo. $f(n) = \Theta(g(n))$

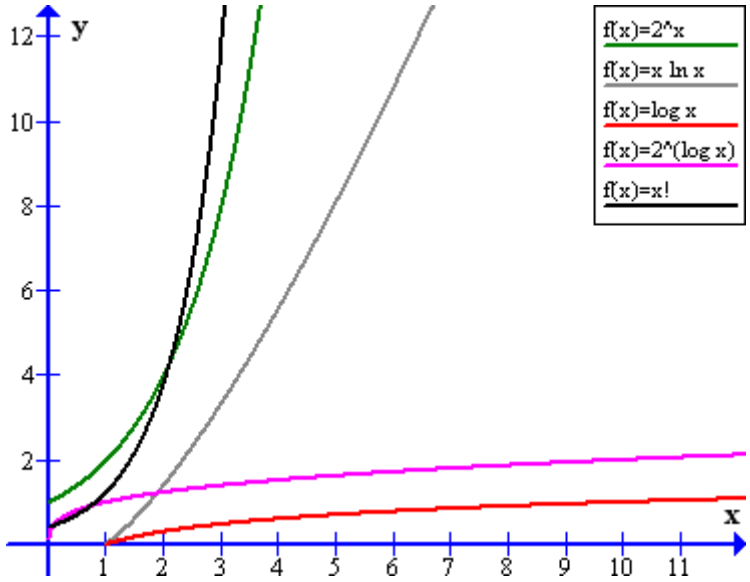
Složenost je funkcija sa skupa prirodnih brojeva u skupu prirodnih brojeva, ali se njihova brzina rasta ocjenjuje analitičkom funkcijom kojoj kodomena nije skup prirodnih brojeva.

Ako su O i o ocjene koje ocjenjuju funkciju ograničavajući je odozgo, onda su Ω i ω ocjene koje to rade odozdo. Ocjena složenosti Θ objedinjuje u sebi ocjene O i Ω.

Ovo su neke uobičajene veličine složenosti koje se javljaju kod algoritama:

- konstanta O(1)
- logaritamski (podpolinom): O(log(n)), O($\log^k$(n))
- stupanjski (polinom): O(n^k), k>2
- linearno logaritamski: O($n^k \times \log(n)$),
- linearan: O(N)
- podeksponecijalna: O($2^{\log(n)}$)
- eksponencijalna: O(2^n)
- faktorijelni (nadeksponencijalna): O(n!), O(2^{n^n})

Grafički prikaz na slici 1.1. omogućuje okvirni prikaz utvrđivanja složenosti redova i odabir prikladnijeg reda.



Slika 1.1. Red složenosti najčešćih funkcija

1.1.2. Osnovni pojam algoritam

Knuth je dao 5 svojstava koja algoritam mora zadovoljavati.[3]

1. **Konačnost** - Algoritam mora završiti nakon izvršenih konačno mnogo koraka koji također moraju biti konačne duljine.
2. **Definiranost** - Svaki korak mora biti nedvosmislen
3. **Ulaz** - Svaki algoritam mora imati 0 ili više ulaza
4. **Izlaz** - Svaki algoritam mora imati barem jedan ili više izlaza
5. **Efektivnost** - Algoritam mora biti učinkovit

Algoritam je postupak za rješavanje nekog masovnog problema. Za algoritam kažemo da rješava masovni problem ako daje rješenje za svaku pojedinu instancu.

2. Prioritetni red

Osim standardnih podataka ponekad je potrebno odrediti njihovu važnost tj. prioritet. Obično prioritet je cijeli ili realni broj. Slično se koristi u bolnicama gdje nije bitan redoslijed dolaska u bolnicu, već stanje svakog pristiglog pacijenta. U operacijskim sustavima slična je situacija. Svaki proces može imati zadan prioritet i izvršava se u određenim vremenskim ciklusima. [7]

Osnovne operacije implementacije prioritetnog reda su ubacivanje novog elementa u red čekanja, te izbacivanje tj. obrada elementa s najvišim prioritetom ili najnižim ovisno o potrebi i implementaciji.

Spomenute operacije svode se na jednostavniji problem ubacivanja elemenata u skup, te izbacivanja najmanjeg elementa. Naime, umjesto originalnih elemenata x, promatramo uređene parove (prioritet(x), x). Za uređene parove definiramo leksikografski uređaj: (prioritet(x_1), x_1) je manji ili jednak od (prioritet(x_2), x_2) ako je (prioritet(x_1) manji od prioritet(x_2)) ili (prioritet(x_1) jednak prioritet(x_2) i x1 manji-ili-jednak od x_2). Ova dosjetka opravdava sljedeću definiciju apstraktnog tipa podataka [18]:

Apstraktni tip podataka prioritetni red (priority queue)

- ELEMENTTYPE ... bilo koji tip s totalnim uređajem <=
- PRIORITY_QUEUE ... podatak ovog tipa je konačni skup čiji elementi su podaci tipa elementtype i međusobno su različiti,
- MAKE NULL(&A) ... funkcija pretvara skup A u prazni skup.
- EMPTY(A) ... funkcija vraća “istinu” ako je A prazan skup, inače vrača “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.

Primijetimo da se ATP prioritetni red može smatrati restrikcijom ATP red. Naime, funkcija DELETE MIN( ) zapravo je kombinacija od MIN( ) i DELETE( ).

2.1. Implementacija prioritetnog reda

Implementacija prioritetnog reda može se izvesti pomoću sortirane vezane liste, binarnog stabla traženja ili pomoću hrpe.

Od raznih varijanti najbolja se čini sortirana vezana lista.

DELETE MIN( ) pronalazi i izbacuje prvi element u listi, pa ima vrijeme O(1).

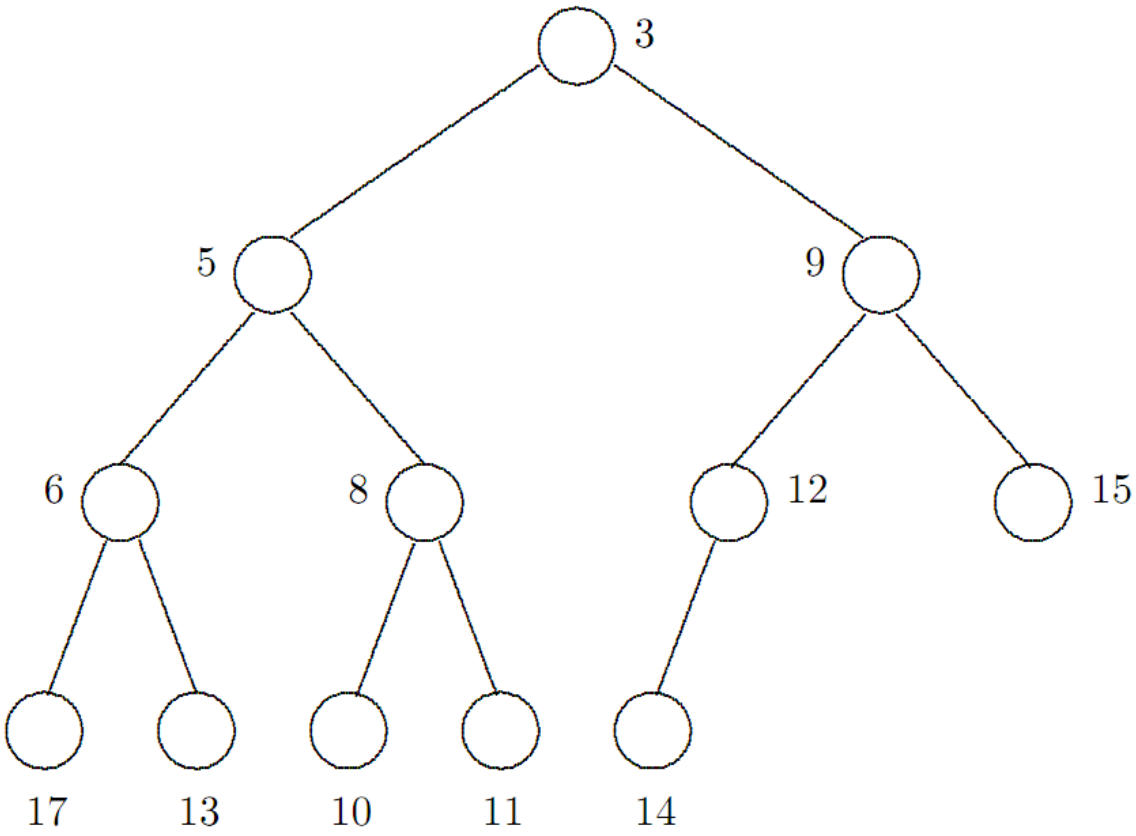
INSERT( ) mora ubaciti novi element na "pravo mjesto", što znači da mora pročitati u prosjeku pola liste. Zato INSERT( ) ima vrijeme O(n), gdje je n broj elemenata u redu.

Kod implementacija prioritetnog reda pomoću hrpe imamo potpuno binarno stablo. Pritom moramo voditi računa da su ispunjeni sljedeći uvjeti [18]:

- čvorovi 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.

Implementacije je potrebno izvesti na sljedeći način. Prioritetni red prikazujemo hrpom. Svakom elementu reda odgovara točno jedan čvor hrpe i obratno. Element reda služi kao oznaka odgovarajućeg čvora hrpe kao što će se vidi na sljedećoj slici.

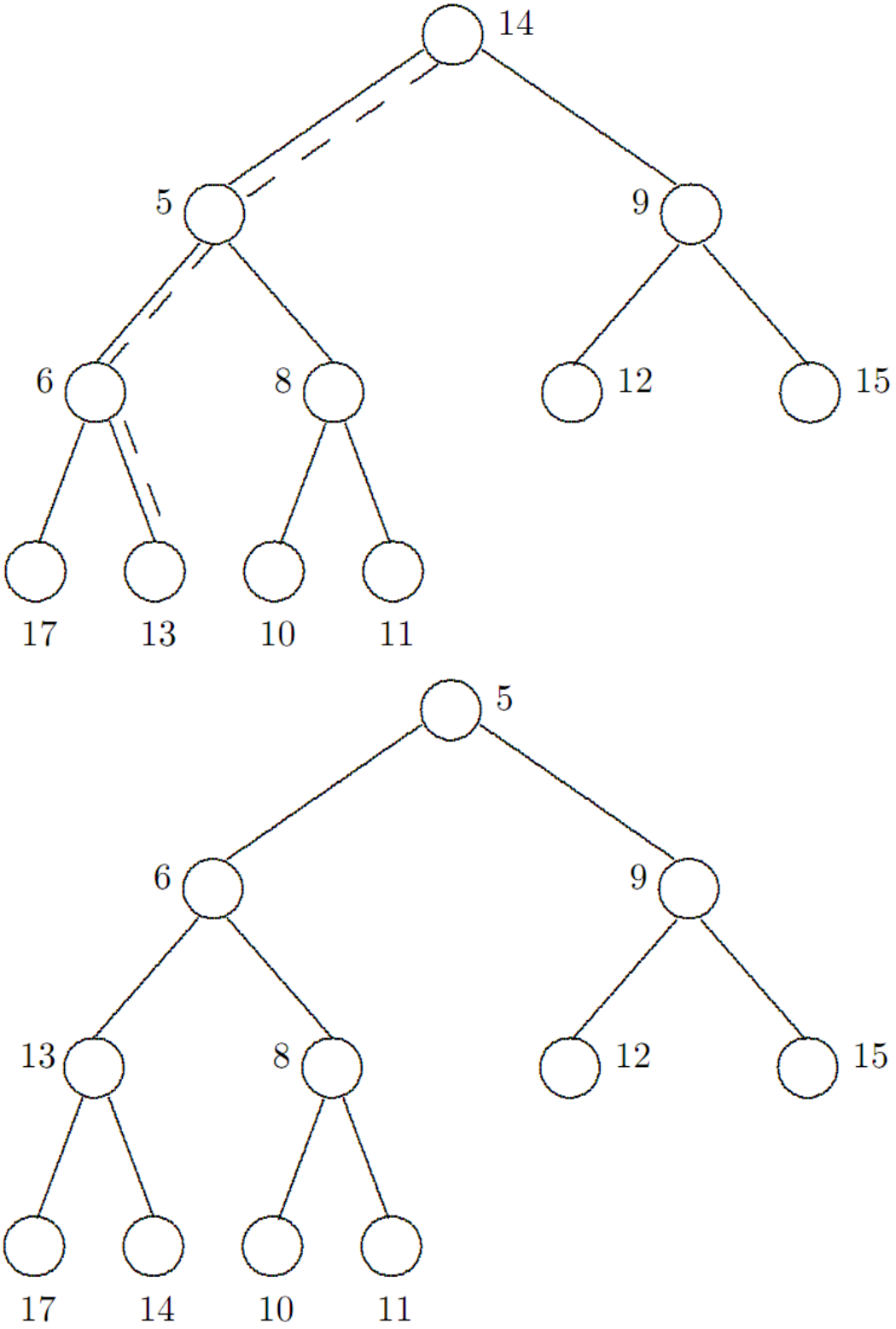
Na Slici 1. vidi se prikaz prioritetnog reda: A = {3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 17} pomoću hrpe. Prikaz nije jedinstven. Iz svojstava hrpe slijedi da je najmanji element u korijenu slika 2..[18]



Slika 2. Prikaz prioritetnog reda pomoću hrpe [18]

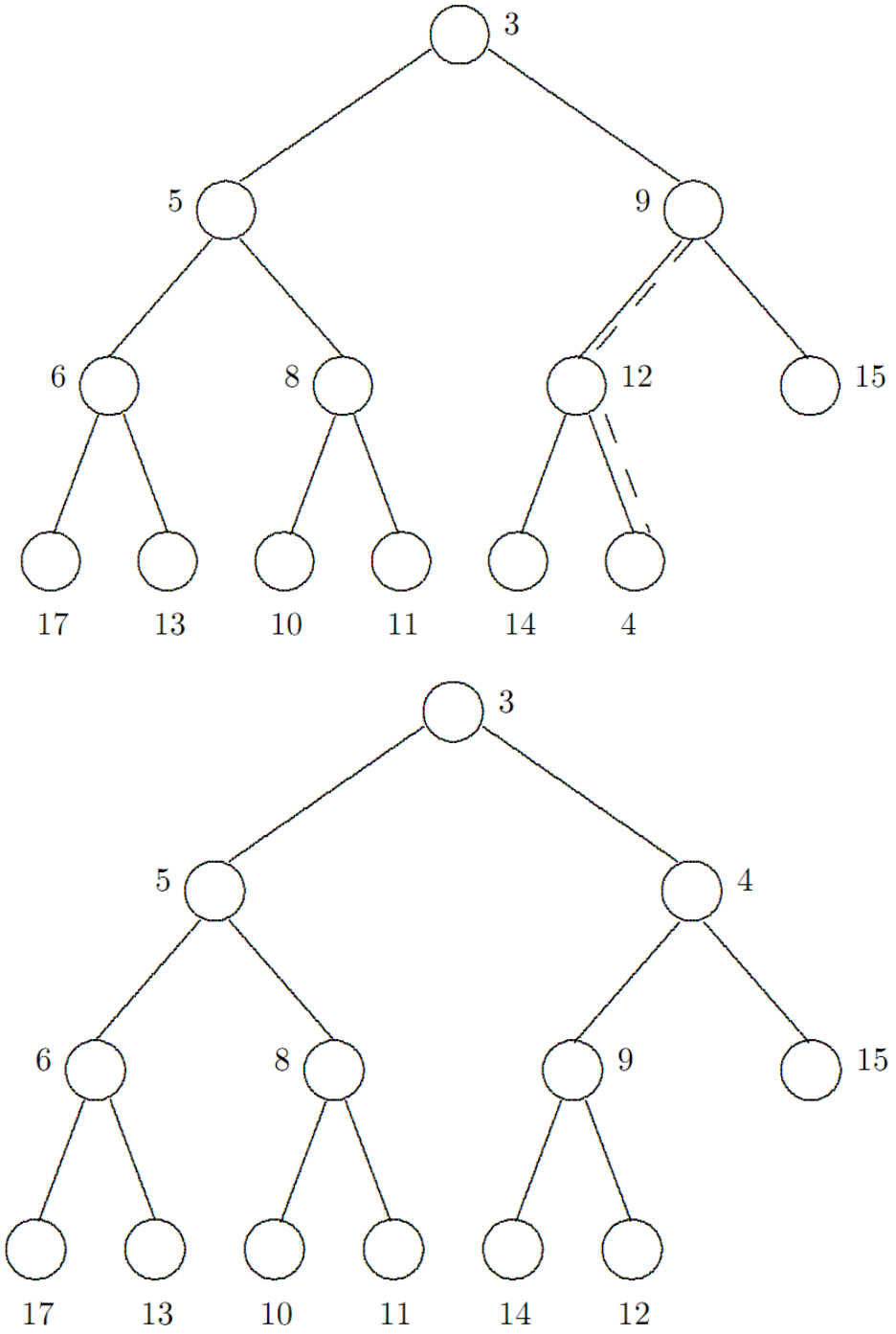
Za operaciju DELETE MIN( ), moramo vraćati element iz korijena do zadnjeg čvora na zadnjem nivou. U protivnom kad bi smo izbacili korijen stablo bi se raspalo. Te izbacujemo zadnji čvor na zadnjem nivou, a njegov element stavimo u korijen. Time se sigurno pokvarilo svojstvo hrpe. Sada slijedi popravak koji se obavlja na sljedeći način. Zamijenit ćemo 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 1. vidi se na slici 1.1.



Slika 2.1 Prikaz operacije DELETE MIN( ).[18]

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 na isti način kao kod DELETE MIN. 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 slika 2.2..



Slika 2.2 učinak ubacivanja elementa s vrijednošću 4 u hrpu [18]

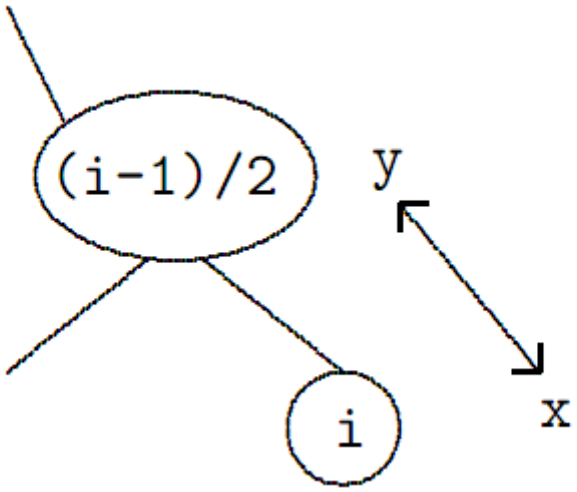
Da bismo implementaciju razradili do kraja, potrebno je odabrati strukturu podataka za prikaz hrpe. S obzirom da je hrpa potpuno binarno stablo, možemo koristiti prikaz pomoću polja

Dakle, potrebne su nam sljedeće definicije [18]:

```
# 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("prioritetni red je pun");
      } 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]) ) {
                  temp = Ap->elements[i];
                  Ap->elements[i] = Ap->elements[(i-1)/2];
                  Ap->elements[(i-1)/2] = temp;
                  i = (i-1)/2;
            }
      }
}
```



Slika 2.3 Određivanje djetetovog roditelja [18].

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

Funkcije MAKE NULL( ) i EMPTY( ) su trivijalne, pa ih nećemo opisati. Primijetimo da funkcija INSERT( ) ima jednu manjkavost: ona stvara novi čvor s elementom x čak i onda kad x već jeste u prioritetnom redu A [18]. 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.

Funkcije INSERT( ) i DELETE MIN( ) obilaze jedan put u potpunom binarnom stablu. Zato je njihovo vrijeme izvršavanja u najgorem slučaju 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.

2.2. Složenost prioritetnog reda

- *INSERT*
  - najgorem slučaju je O(N) vrijeme, prolazak kroz listu pomiče elemente, a kod hrpe O(log n) u najgorem jer obilaze jedan put cijelu hrpu
  - $T(n) <= C_1 + (n-1) * C_2$
  - $T(n) = O(n)$
- *DELETE MIN*
  - u svim slučajevima je O(1) vrijeme, jer uzima prvi element u listi ili prosječno O(log n) ako koristimo hrpu
  - $T(n) <= C_1 + n/2 * C_2$
  - $T(2^n) = R_1 + R_2 * 2^n$
  - $T(n) = R_1 + \log n$
  - $T(n) = O(\log n)$

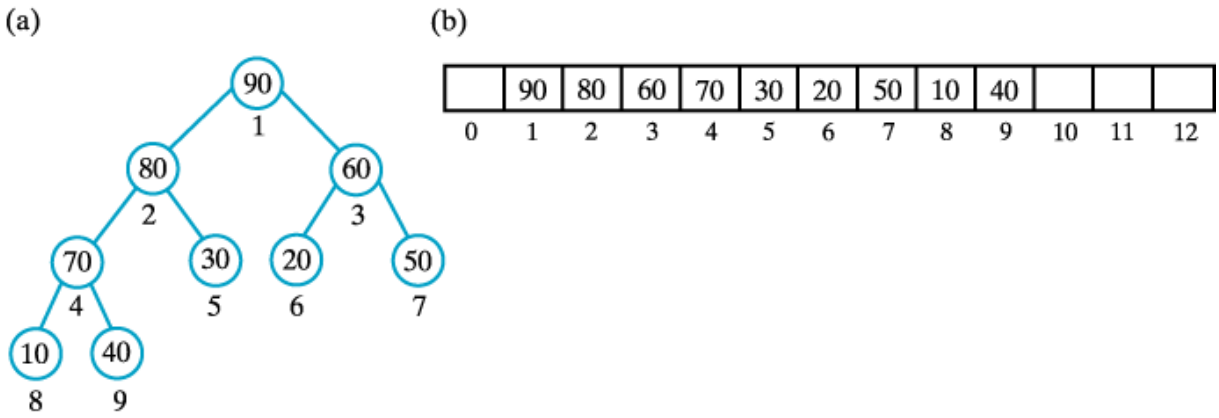
3. Hrpa

Hrpa je specijalni slučaj strukture stabla koji ima sljedeća svojstava [4]. Element s najmanjom vrijednošću uvijek je korijen hrpe. Takvu hrpu još zove i minimalna hrpa. Ona može biti i obratna, tako da najveća vrijednost bude u korijenu hrpe. Tada se radi o maksimalnoj hrpi. U ovom slučaju uzet ćemo da je hrpa potpuno binarno stablo koje ima svojstva da element koji je roditelj ima manju vrijednost od svoje djece. Popunjavanje ide redom lijevo na desno u svim razinama. Hrpa mora biti potpuno popunjena u svim razinama osim iznimka u zadnjem redu. Uvijek popunjavamo s lijeva na desno. Čvorovi međusobno moraju biti usporediva. Hrpa je maksimalno efikasna struktura apstraktnog tipa podataka (STP). Najčešća primjena je u prioritetnom redu, Dijkstinom algoritmu, heapsortu itd. Postoji nekoliko vrsta hrpa, binarna, binomna (brže spajanje dvije hrpe), Fibonacijeva (kolekcija stabla, brže performanse), uparivanja, soft hrpa, itd..

Hrpa se obično implementira pomoću polja, liste ili nesortirane liste.

Za hrpu je idealna implementacija binarnog stabla pomoću polja, a implementacija obično sadržava ove funkcije:

- InitHeap (MAX_SIZE) – kreira praznu hrpu koja može sadržavati MAX_SIZE elemenata HeapFull(heap, n) – ako je n jednak MAX_SIZE vrati true inače false
- Insert (heap, item, n) – ako hrpa nije puna ubaci novi element u hrpu i vrati hrpu inače prikazi poruku o grešci
- Empty (heap, n) – ako je n veći od 0 vrati false inače true
- Delete (heap,n) – ako hrpa nije puna vrati najmanji element te ga izbriši iz hrpe inače prikazi poruku o grešci.



Slika 3. Prikaz hrpe a) pomoću polja u memoriji b) [24]

Pri tome se implementacija može još pojednostavniti. Naime, u hrpi se uvijek stablo puni tako da je u polju u kojem je stablo zapisano elementi nalaze u prvi n elemenata, slika 3. Time omogućujemo jednostavno određivanje lokacije elementa u čvorovima roditelja ili djece po sljedećoj formuli.

Roditelj čvora na *i* indeksu u polju se može pronaći ako *i/2* osim ako je *i=1* odnosno korijen

Dijete čvora na *i* indeksu u polju se može pronaći na *2i* i *2i+1* (lijevo i desno dijete) slika 2.3.

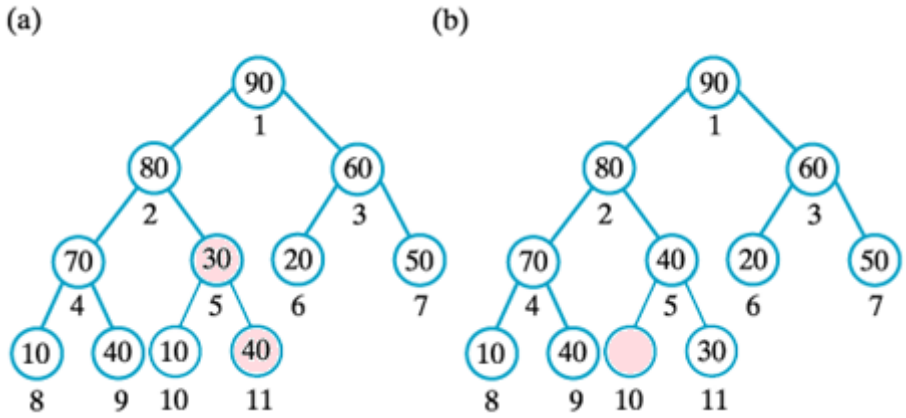
$$\text{Roditelj}(i) = \lfloor i/2 \rfloor$$

$$\text{Lijevo dijete}(i) = 2i$$

$$\text{Desno dijete}(i) = 2i + 1$$

Dakle, nije potrebno pamtiti koji su elementi polja zauzeti. Sortiranje liste koje se temelji na algoritmu dodavanja i brisanja iz hrpe naziva se heap-sort jedan je od boljih algoritama sortiranja.

Primjeri neispravne hrpe



Slika 3.1 Primjeri neispravnih hrpa [24]

Slika 3.1 a), svojstva hrpe nisu poštovana

Slika 3.1 b), prazan element u zadnjem retku nije dozvoljen, elementi se popunjavanju s lijeva na desno po redu bez preskakanja.

Svojstva hrpe:

- Oznake čvorova imaju uređeni uređaj <.
- Oznake oba djeteta su veća od oznake njihova roditelja.
- Stablo je potpuno, tj. u svim nivoima stabla osim pretposljednjeg i posljednjeg svi čvorovi imaju po dvoje djece.
- Posljednji se nivo uvijek puni s lijeva nadesno.
- Dozvoljena je operacija brisanja korijena stabla. Ona se izvodi tako da nakon brisanja stablo i dalje ostaje hrpa.

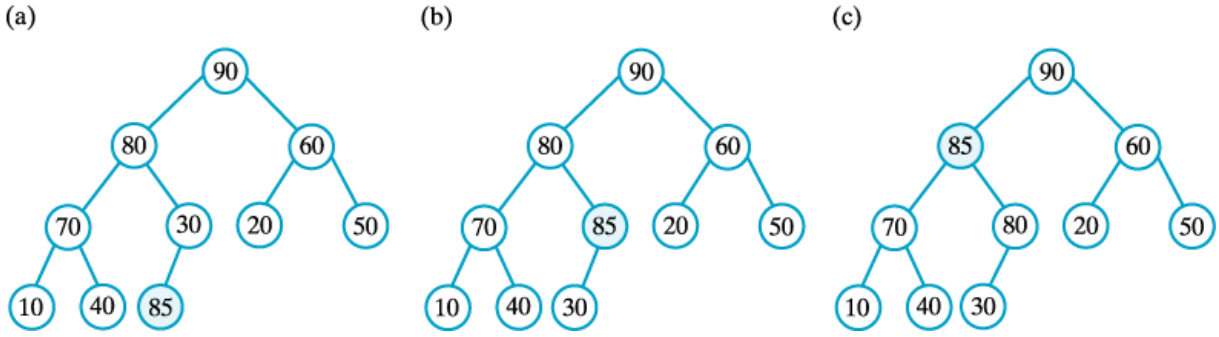
3.1. Implementacija hrpe

3.1.1. Punjenje hrpe

Pseudo algoritam za dodati novi element

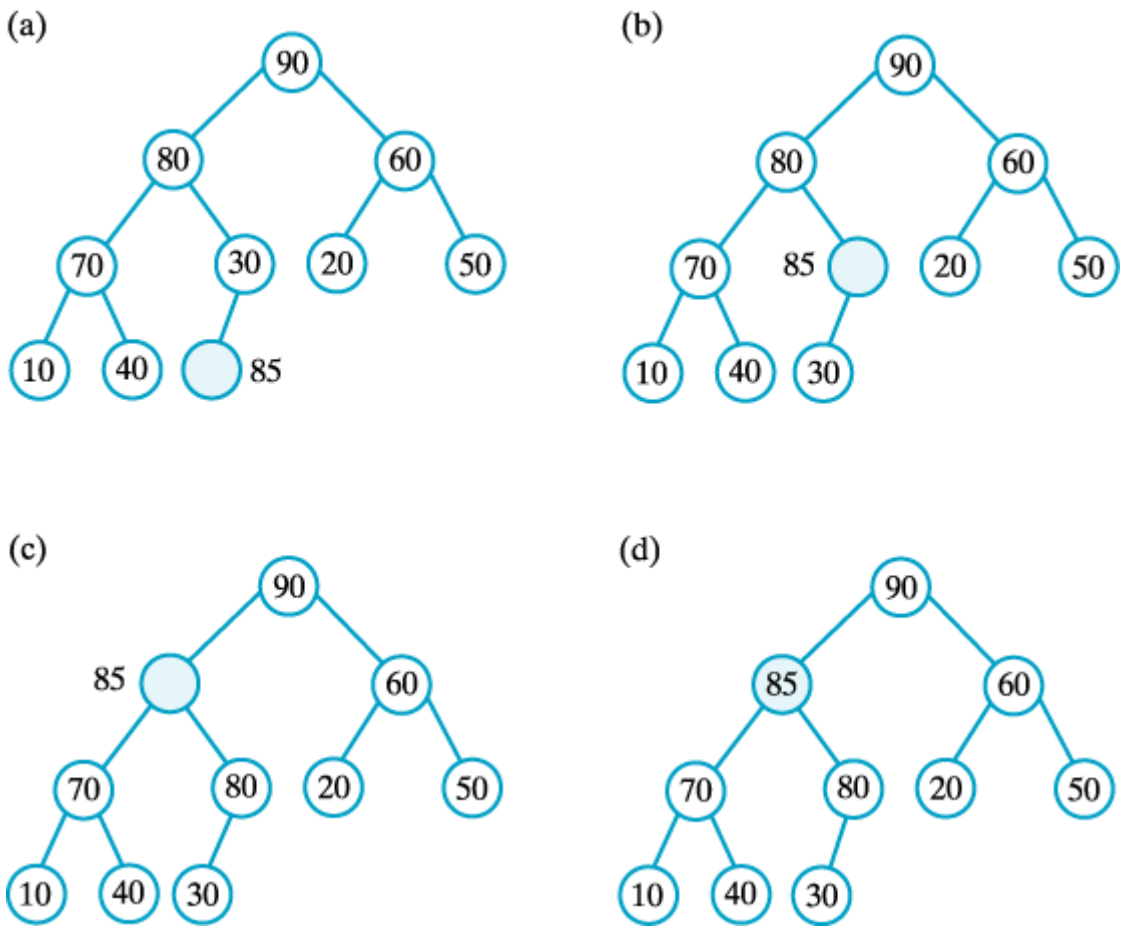
```
function insert(ključ)
if (!hrpa.isFull()) {
      noviIndeks = indeks slijedećeg slobodne lokacije u polju
      roditeljIndeks = noviIndeks/2 //(roditelj slobodne lokacije)
      while (ključ > hrpa[roditeljIndeks])
      {     hepa[noviIndeks] = hrpa[roditeljIndeks]
            // premjestimo roditelja na slobodnu lokaciju
            // ažuriranje indeksa
            noviIndeks = roditeljIndeks
            roditeljIndeks = noviIndeks/2
      } // kraj while
}
```

Primjer dodavanja broja 85 u hrpi gdje je korijen maksimalna vrijednost slika 3.2.



Slika 3.2 Predstavlja dodavanje elementa u hrpu. [24]

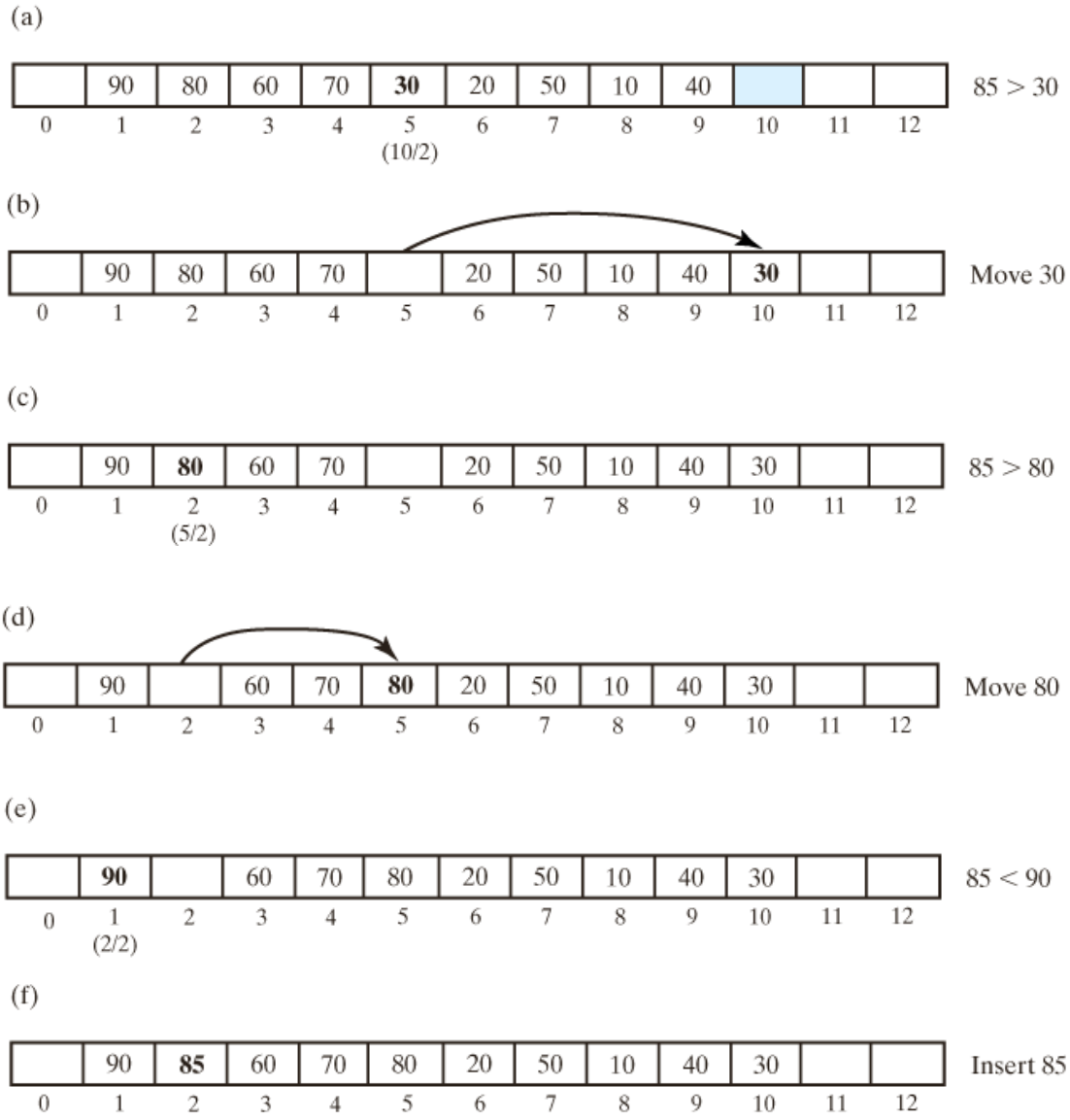
Prema slici 3.2. prvi korak 85. smo zapisali na prvo slobodno mjesto, a to je u zadnjem redu. Nakon toga stablo više ne poštuje svojstva hrpe, pa u koraku b) zamijenit ćemo vrijednosti 85 s manjim roditeljem 30. Ako provjerimo još jednom svojstva hrpe primijetit ćemo da je 85 veći od svog novog roditelja 80. U koraku c) ponovno zamjenu i provjerimo dali je stablo hrpa.



Slika 3.2 Predstavlja dodavanje elementa u hrpu, malo bolji način.[24]

Primjer je sličan kao prethodni s razlikom samo u implementaciji. Ova implementacija je optimizirana da novu vrijednost ne zapisuje odmah u hrpu već u privremenu varijablu. Kako nije moguće direktno vršiti zamjenu npr. A s B, (izgubila bi se prethodna vrijednost), potrebno je vrijednost A prebaciti u neku pomoćnu varijablu prilikom zamjene. Taj korak u prošlom slučaju radimo tri puta, a korištenjem ove optimizacije broj koraka se prilikom dodavanje novog elementa smanjuje.

Slika 3.3 je sličan primjer kao prethodni slučaj samo što ovog puta ne gledamo graf već polje u memoriji te možemo vidjeti kako se operacija dodavanje zbilja radi. Svaki korak je detaljno pojašnjen. Ovo se koristi optimizirana izvedba s minimalnim brojem zamjena.



Slika 3.3 Primjer ako dodavanje novog elementa ako gledamo polje [24]

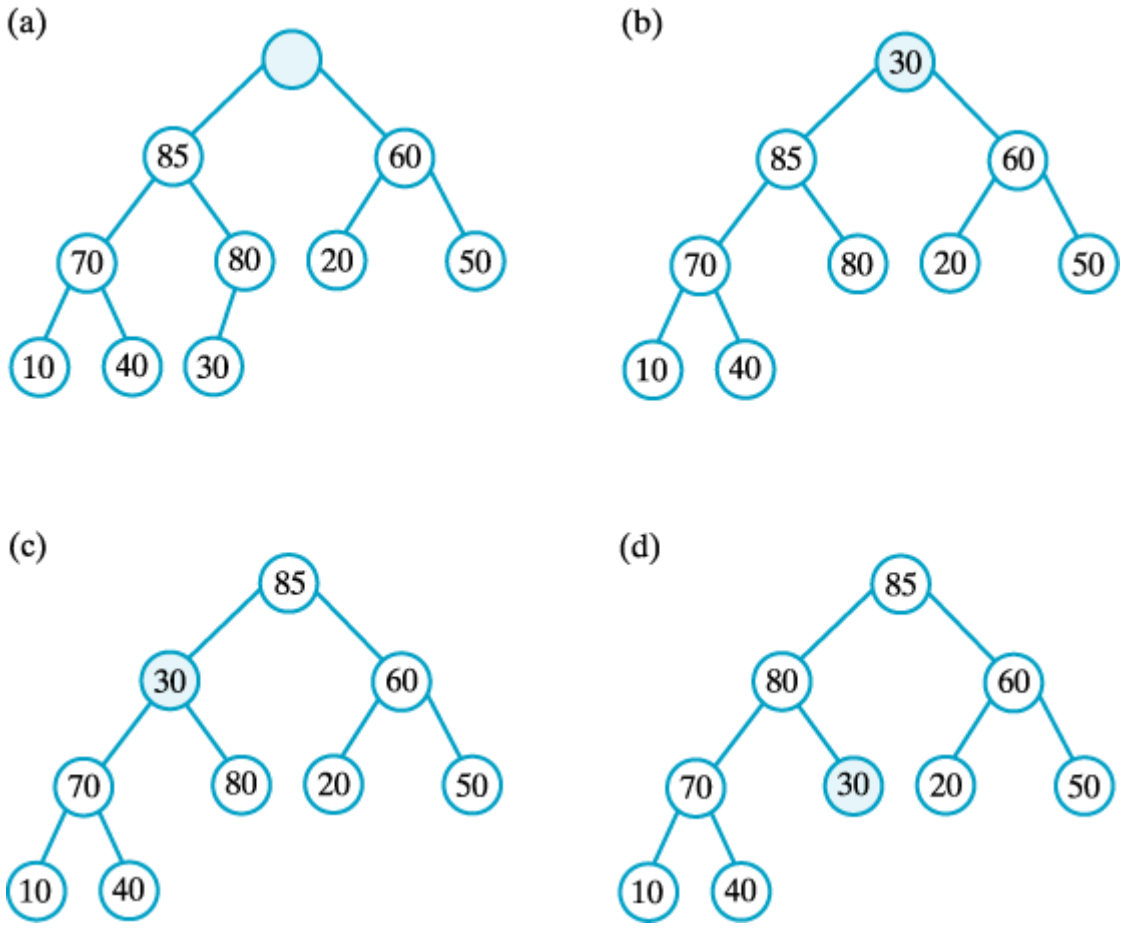
1. Čvor se stavlja na prvo prazno mjesto u posljednjem nivou stabla.
2. Pogleda se je li njegova oznaka manja od oznake njegova roditelja, te ako jest zamijene im se mjesta.
3. Korak 2 se ponavlja za upravo dodani čvor sve dok dodani čvor ne postane korijen ili dok ne dođe do toga da mu roditelj ima manju oznaku od njega.

3.1.2. Brisanje korijena

Pseudo algoritam za brisanje elementa

```
function remove() {
    hrpa[indeks] = hrpa[1] // postavljamo zadnji element u korijen
      indeks-- // smanjio se br. elemenata za 1
    if (!hrpa.isEmpty()) { // ako nije prazna hrpa
            // u korijen postavljamo zadnji element
            hrpa[noviIndeks] = hrpa[roditeljIndeks]
            roditelj = 1 // index korijena
            while (roditelj < indeks / 2) {
                  dijeteIndeks = 2 * roditelj + 1;
                  if (dijeteIndeks < indeks - 1
                        && hrpa[dijeteIndeks] > hrpa[dijeteIndeks + 1]) {
                        ++dijeteIndeks;
                  }
                  if (hrpa[roditelj] <= hrpa[dijeteIndeks]) {
                        break;
                  }
                  // zamijeni roditelj i dijete
                  swap(roditelj, dijeteIndeks);
                  korijen = dijeteIndeks;
            }
      }
}
```

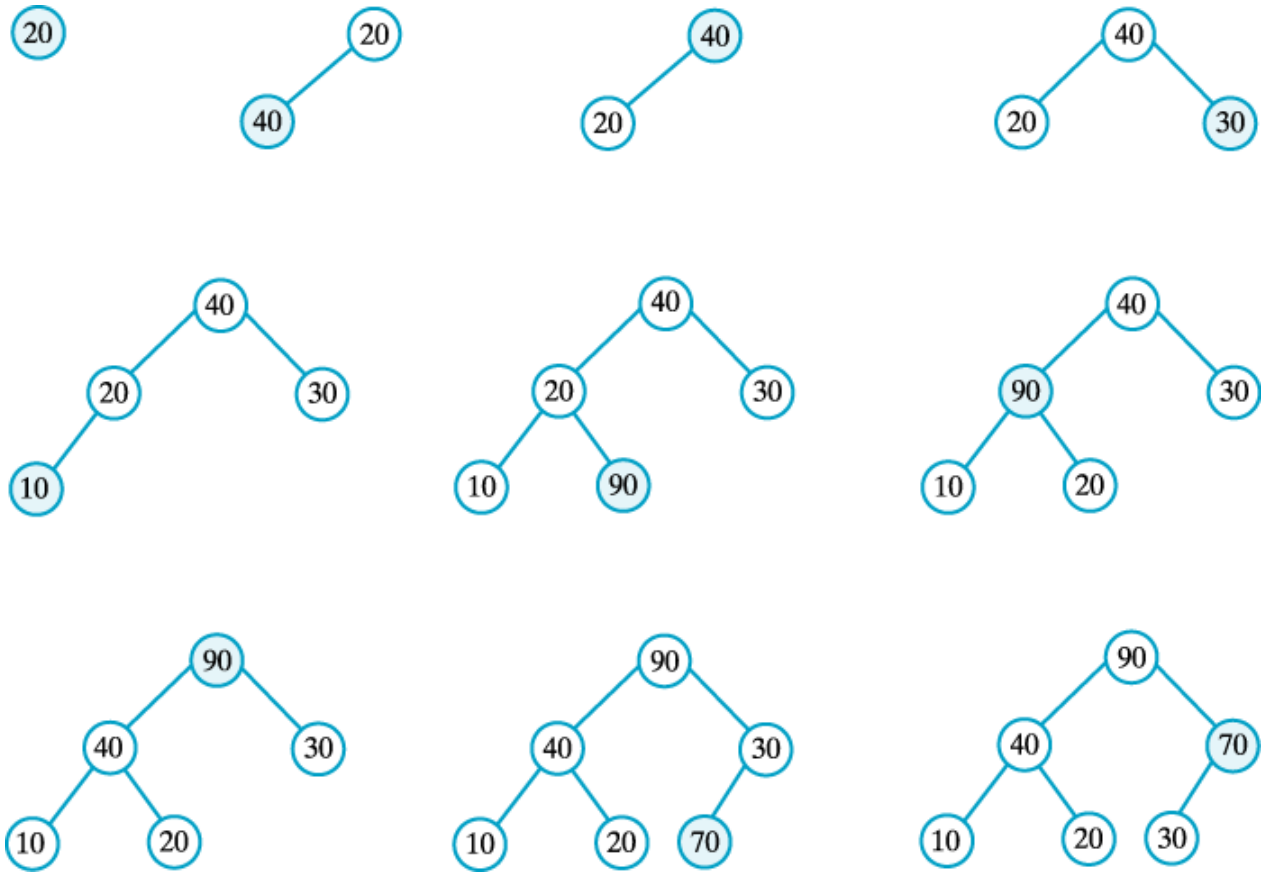
Primjer



Slika 3.4 Brisanje elementa, odnosno korijena i provjere da li su poštovana sva svojstva hrpe [24]

1. Obriše se posljednji čvor stabla, a njegova se oznaka prepiše u korijen stabla. Korijen postaje promatrani čvor
2. Nađe se dijete promatranog čvora s manjom oznakom. Ako to dijete ima manju oznaku od promatranog čvora. Zamijene im se mjesta, a promatrani čvor postaje onaj u kojem se nalazi promatrani čvor
3. Brisanje korijena
4. Korak 2 se ponavlja sve dok promatrani čvor ne postane list ili dok dijete tog čvora koje ima manju oznaku nema veću oznaku od tog čvora.

Kreiranje hrpe primjer 20,40,30,10,90,70



Slika 3.5. Predstavlja kreiranje hrpe gdje je najveća vrijednost u korijenu.[24]

Prvi korak postavljanje prvog elemenata koji je odmah i korijen. Svaki sljedeći element koji dodajemo u hrpu stavljamo u zadnji red na prvo slobodno mjesto. Kako u ovom slučaju nema djece, kreiramo prvo lijevo dijete. Sada uspoređujemo dijete s roditeljem, ako vrijednost djeteta veća od roditelja zamijenimo im mjesta. Sljedeći element stavljamo na prvo slobodno mjesto u zadnjem redu, to će biti desno dijete. Kako je dijete manje od roditelja nije potrebno raditi razmjene. Postupak se ponavlja sve dok ne dodamo sve elemente u hrpu na isti način.

3.2. Složenost hrpe

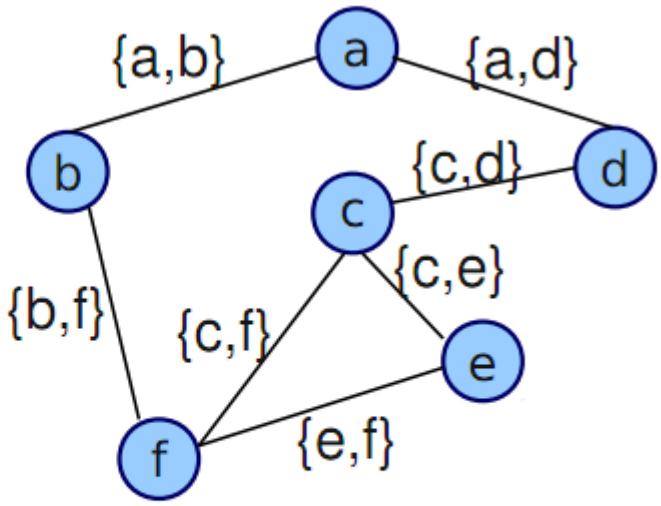
- *Insert*
  - najgori slučaj: O(log N) vrijeme, pomiče elemente od dna prema vrhu
  - $T(n) <= C_1 + n/2 * C_2$
  - $T(2^n) = R_1 + R_2 * 2^n$
  - $T(n) = R_1 + \log n$
  - $T(n) = O(\log n)$
- *DeleteMin*
  - najgori slučaj: O(log N) vrijeme
  - prosječno je O(log N) vrijeme (element ako je najveći element na vrhu, a najmanji element na dnu)
  - $T(n) <= C_1 + n/2 * C_2$
  - $T(2^n) = R_1 + R_2 * 2^n$
  - $T(n) = R_1 + \log n$
  - $T(n) = O(\log n)$

4. Graf

Graf je matematička struktura, a teorija grafova je matematička disciplina koja proučava zakonitosti na grafovima.

Grafovi se sastoje od vrhova V i bridova E, pa kažemo da je to uređeni par vrhova i bridova. Grafovi mogu biti neusmjereni i usmjereni. Postoji puno podjela u teoriji grafova, ali za sad nećemo ulaziti u njihove detalje.

Neusmjereni graf G je uređeni par (V, E), pri čemu je V skup vrhova grafa (konačan) i $E \subseteq V \times V$ skup parova elemenata iz V, koji čini skup bridova grafa slika 4 [4].

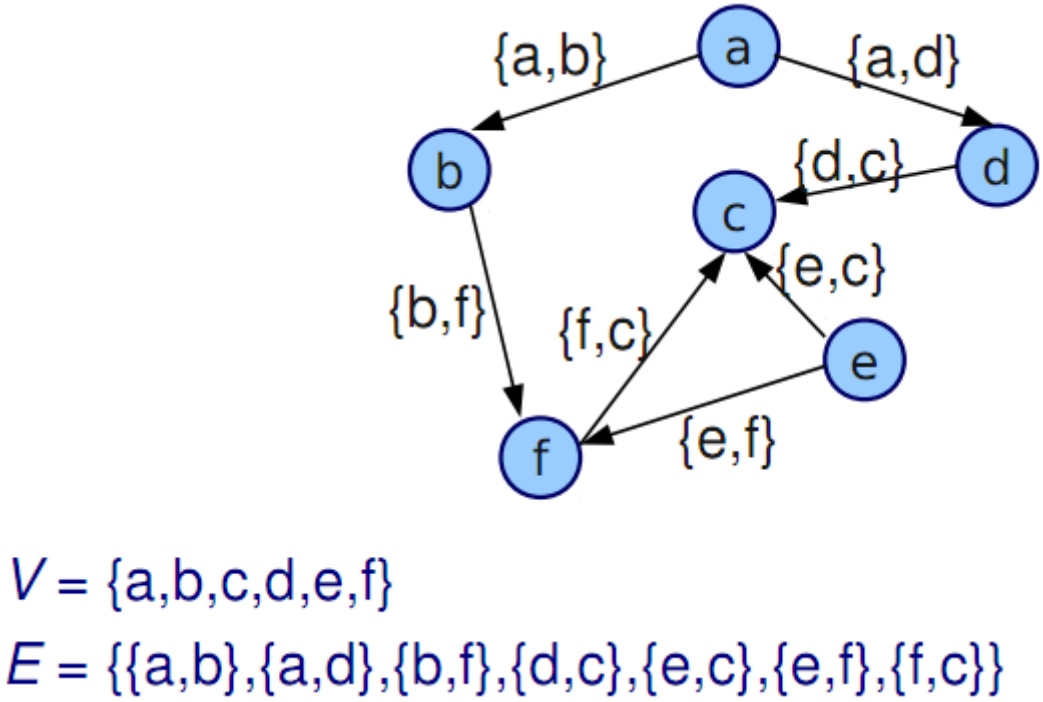


$V = \{a,b,c,d,e,f\}$

$E = \{\{a,b\},\{a,d\},\{b,f\},\{c,d\},\{c,e\},\{c,f\},\{e,f\}\}$

Slika 4. Primjer neusmjereni graf [18]

Usmjereni graf G je uređeni par (V, E), pri čemu je V skup vrhova grafa (konačan) i E⊆V×V, s tim da vodimo računa o smjeru veza između vrhova. Smjer se označuje strelicom slika 4.1 [4].



Slika 4.1 Primjer usmjerenog grafa [18]

Grafove možemo prikazati pomoću matrice na računalu. Postoje dva načina prikaza: pomoću matrice incidencije i matrice susjedstva. Matrica incidencije u redovima ima vrhove, a u stupcima bridove. Nedostatak matrice incidencije javlja se kod grafa koji ima puno bridova. Tada je ova matrica znatno veća od matrice susjedstva. Matrica susjedstva u redovima i stupcima ima vrhove. Zbog toga se češće koriste matrice susjedstva za reprezentaciju grafa u računalu. Matrica susjedstva zauzima $O(n^2)$, ali u O(1) vremena možemo saznati jesu li dva čvora povezana. Matrica incidencije zauzima O(n) memorije odnosno potreba za memorijom je proporcionalna zbroju stupnjeva čvorova.

Usmjereni graf je graf u kojem su bridovi „jednosmjerni“, tj. za brid se definira ulazni i izlazni vrh. U tom slučaju se definicija grafa neznatno mijenja tako da bridovi postaju uređeni parovi vrhova. Usmjereni i neusmjereni grafovi mogu biti težinski. U težinskom grafu svakom bridu dodjeljujemo njegovu težinu (broj koji označava njegovu vrijednost). Primijetimo da se ne težinski grafovi mogu smatrati težinskim, pri čemu svi bridovi imaju istu težinu (recimo 1).

Problem minimalnog razapinjućeg stabla. Pretpostavimo da imamo težinski graf, te da želimo naći skup bridova grafa minimalne ukupne težine koji će povezivati sve vrhove grafa. Za tako nešto poslužit će nam Kruskalov ili Primov algoritam. Primjena grafova je vrlo velika, moguće je primjerice odrediti najkraći put, najjeftiniji put, transportni put (opskrba vodom), pobjednika turnira, problem kineskog poštara između vrhova koristeći neke od ovih algoritama Dijkstrin, Floyd-Warshallov, Bellman-Ford, BFS[4], DFS[5] itd.. Složenost nabrojanih algoritama je eksponencijalna, a s povećanjem broja vrhova i bridova raste vrijeme potrebno da se izvrši algoritam. Najveći izazov današnjice je utvrditi veze između svih članova socijalane mreže Facebook.

[4] BFS – engl. Breadth first search, pretraživanje grafa u širinu
[5] DFS – engl. Deep first search. pretraživanje u dubinu

4.1. Implementacija grafa

Implementacija grafa se može izvesti na više načina. Ranije je navedeno da postoji puno različitih vrsta grafova, te je nužno prilagoditi implementaciju potrebama. Da bi se pojednostavnilo pisanje algoritama koriste se STL. U primjeru implementacije objašnjna je mogućnost da bude generalna za usmjerene ili neusmjerene grafove, kao i za težinske grafove.

4. ATP graf se sastoji od strukture podataka koja sadrži graf i sljedećih operacija:

   – Create()

      • omogućuje kreiranje grafa

   – InsertVertix(graph)

      • omogućuje ubacivanje novog retka i stupca u matricu

      • ako nema mjesta u matrici vraća grešku

   – DeleteVertix(graph,v)

      • omogućuje izbacivanje v retka i stupca iz matrice

      • ako vrhovi ne postoje vraća grešku

   – InsertEdge(graph,v1,v2,w)

      • ako se ne radi o usmjerenom grafu onda moramo voditi računa da ako gledamo težinu v1,v2 bude jednaka težini v2,v1. U praksi moramo dva puta pozvati funkciju InsertEdge jednom [v1][v2] = w, a drugi put [v2][v1] = w.

      • ako vrhovi ne postoje vraća grešku

   – DeleteEdge(graph,v1,v2)

      • ako se ne radi o usmjerenom grafu onda moramo voditi računa da uklonimo bridove v1,v2 i v2,v1 kao što smo to radili i kod InsertEdge

      • ako vrhovi ne postoje vraća grešku

   – IsEmpty(graph)

    - vraća da li je graf prazan

– Adjacent(graph,v1,v2)

    - vraća težinsku vrijednost između vrhova v1,v2
    - ako vrhovi ne postoje vraća grešku

Implementacija grafa pomoću polja tj. tablice susljednosti

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

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

graph Create(){
      G = new graph;
      G->last=0;
      return G;
}
graph InsertVertix(graph *G){
      return graph->last++;
}
graph DeleteVertix(graph *G, int v){
      int x,y;
      if(G->last<MAXSIZE){
            for(x=0; x<=G->last; x++){
                  for(y=0; y<=G->last; y++){
                        if(x>=v-1) G[x-1][y] = G[x][y];
                  }
                  if(y>=v-1) G[x][y-1] = G[x][y];
            }
            G->last--;
            return graph;
      } else {
            error("greska nema mjesta za novi vrh");
      }

}
graph InsertEdge(graph *G, int v1, int v2, int w){
      if(v1<G->last && v2<G->last){
            return G[v1-1][v2-1] = w;
      } else {
            error("greska ne postoji ti vrhovi");
      }
}
graph DeleteEdge(graph *G, int v1, int v2){
      if(v1<G->last && v2<G->last){
            return G[v1-1][v2-1] = 0;
      } else {
            error("greska ne postoji ti vrhovi");
      }
}
bool IsEmpty(graph *G){
      if(G->last==0){
            return true;
      } else {
            return false;
      }
}
int Adjacent(graph *G, v1, v2){
      if(v1<G->last && v2<G->last){
            return G[v1][v2];
      } else {
            error("greska ne postoji ti vrhovi");
      }
}
```

4.2. Složenost grafa

- *InsertVertix*
  - O(1) vrijeme
- DeleteVertix
  - $O(n^2)$ vrijeme
  - $T(n) <= n * n * C_1 * C_2 * C_3$
  - $T(n) = R_1 + n^2$
  - $T(n) = O(n^2)$
- InsertEdge
  - O(1) vrijeme
- DeleteEdge
  - O(1) vrijeme
- Adjacent
  - O(1) vrijeme

5. Skup

Skup je zbirka podataka istog tipa. U takvoj zbirci ne mogu postojati dva elementa sa istom vrijednošću. Unutar skupa ne zadaje se nikakav eksplicitni linearni ili hijerarhijski uređaj među podacima (kao kod liste i stabla), niti bilo kakav drugi oblik veza među podacima.

Skup ćemo definirati kao apstraktni tip podataka, s operacijama koje su uobičajene u matematici.[18]

Apstraktni tip podataka SET:

elementtype ... bilo koji tip s totalnim uređajem <=.[22]

- 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 unija{x}, ako x veći je element od A, tada INSERT( ) ne mijenja A.
- DELETE(x,A) ... funkcija izbacuje element x iz skupa A, tj. mijenja A u A \{x}, ako x nije element od A, tada DELETE( ) ne mijenja A.
- MEMBER(x,A) ... funkcija vraća „istinu“ ako je x element od A, inače „laž“
- MIN(A), MAX(A) ... funkcija vraća najmanji odnosno najveći element skupa A, u smislu uređaja <=. Nije definirana ako je A prazan skup.
- SUBSET(A,B) ... funkcija vraća „istinu“ ako je A podskup od B, inače vraća „laž“.
- UNION(A,B,C) ... funkcija pretvara skup C u uniju skupova A i B.
- INTERSECTION(A,B,C) ... funkcija pretvara skup C u presjek skupova A i B.
- DIFFERENCE(A,B,C) ... funkcija pretvara skup C u razliku skupova A i B.

Ovako zadani apstraktni tip podataka SET izuzetno je teško implementirati. Ako postignemo efikasno obavljanje jednih operacija, neke druge operacije obavljat će se sporije. Zbog toga ćemo se više posvetiti ovim operacijama UNION( ), INTERSECTION( ), DIFFERENCE( ), SUBSET( ).

5.1. Implementacija skupa

5.1.1. Implementacija skupa pomoću sortirane vezane liste

Prikazujemo skup kao listu pomoću pokazivača. Veličina skupova dobivenih operacijama union(), intersection() i difference() može jako varirati. Da bi se operacije efikasnije obavljale, dobro je da lista bude sortirana u skladu s <=.

Pretpostavimo da su definirani funkcije intersection(), union( ), difference( ) i subset( ) su vrlo slične. Vrijeme izvršavanja je proporcionalno duljini jedne od listi [22].

```
void INTERSECTION (SET *A, SET *B, SET *C) {
      settype *ac, *bc, *cc;
      /* tekuće ćelije u listi za A i B, te zadnja ćelija u listi za C */
      C = new settype;
      /* stvori header liste za C */
      ac = A->next;
      bc = B->next;
      cc = *C;
      while ((ac!=NULL) && (bc!=NULL)) {
            /* usporedi tekuće elemente liste A i B */
            if ( (ac->element) == (bc->element) ) {
                  /* dodaj element u presjek C */
                  cc->next = new settype;
                  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;
}


void UNION (SET *A, SET *B, SET *C) {
      settype *ac, *bc, *cc;
      C = new settype;
      ac = A->next;
      bc = B->next;
      cc = *C;
      while ((ac!=NULL) && (bc!=NULL)) {
            // usporedi tekuće elemente liste A i B, a već nije u C
            if( cc->element!=ac->element && cc->element!=bc->element) {
                  if(ac->element != bc->element) {
                        // dodaj novi iz A
                        cc->next = new settype;
                        cc = cc->next;
                        cc->element = ac->element;
                        ac = ac->next;
```

```
                        // dodaj novi iz B ako se već ne nalazi u C
                        if(cc->element!=bc->element) {
                              cc->next = new settype;
                              cc = cc->next;
                              cc->element = ab->element;
                              bc = bc->next;
                        }
                  } else {
                        // dodaj samo iz A je su A i B isti
                        cc->next = new settype;
                        cc = cc->next;
                        cc->element = ac->element;
                        ac = ac->next;
                  }
            } else {
                  bc = bc->next;
                  ac = ac->next;
            }
      }
      cc->next = NULL;
}



void DIFFERENCE (SET *A, SET *B, SET *C) {
      settype *ac, *bc, *cc;
      C = new settype;
      ac = A->next;
      bc = B->next;
      cc = *C;
      while (ac!=NULL) {
        bool novi = true;
            while (bc!=NULL) {
            if(ac->element==bc->element){
                novi = false;
                break;
            }
        }
        if(novi) {
            cc->next = new settype;
                  cc = cc->next;
                  cc->element = ac->element;
                  ac = ac->next;
                  bc = bc->next;
        }
    }
}



void SUBSET (SET *A, SET *B, SET *C) {
      settype *ac, *bc, *cc;
      C = new settype;
      ac = A->next;
      bc = B->next;
      cc = *C;
    while (ac!=NULL) {
        bool novi = true;
        while (bc!=NULL) {
            if(ac->element==bc->element){
```

```
                novi = false;
                break;
            }
        }
        if(!novi) {
            cc->next = new settype;
                  cc = cc->next;
                  cc->element = ac->element;
                  ac = ac->next;
                  bc = bc->next;
        }
    }
}
```

5.2. Složenost skupa

- *Intersection*
  - O(n) vrijeme
  - $T(n) <= C_1 * n * C_2 * C_3$
  - $T(n) = R_1 + R_2 * n$
  - $T(n) = R_1 + n$
  - $T(n) = O(n)$
- *Union*
  - O(n) vrijeme
  - $T(n) <= C_1 * n * C_2 * C_3 * C_4$
  - $T(n) = R_1 + n$
  - $T(n) = O(n)$
- *Difference*
  - O(n) vrijeme
  - $T(n) <= C_1 * n * C_2$
  - $T(n) = R_1 + n$
  - $T(n) = O(n)$
- *Subset*
  - $O(n^2)$ vrijeme
  - $T(n) <= C_1 * n * C_2 * n * C_3$
  - $T(n) = R_1 * n + R_2 * n + R_3$
  - $T(n) = R_1 * n^2$
  - $T(n) = O(n^2)$

6. Zaključak

Tijekom ovog rada najviše su me dojmili ATP grafovi pomoću kojih se mogu prikazati i riješiti gotovo sve vrste problema. . Što je graf kompliciraniji to je efikasnije rješavanje problema pomoći njega. U slučaju grafova postoji i negativna strana Primjerice za jednu relativno jednostavnu operaciju kao što je jesu li dva grafa izomorfna, predstavlja veliki problem računalu ako graf ima više tisuća bridova i vrhova. Problematika je u očuvanju susjedstva grafa koju treba preispitati.

Fasciniralo me je i kako pohraniti graf bio on usmjereni ili ne usmjerenim, težinski ili ne težinski, a da sve implementirane operacije funkcioniraju bez greške. Rješenje je u matrici susjedstva. Ukoliko je ona simetrična, lako možemo zaključiti da se radi o ne usmjerenom grafu. Ako matrica ima različite vrijednosti od 0 i 1, radi se o težinskom grafu. Vrijednost u matrici predstavlja težinu između ta dva vrha. Kada se radi o kombinaciji težinskog i usmjerenog grafa, takav graf nazivamo di-graf. Graf dopušta da vrh bude povezan bridom sa samim sobom, a to nazivamo petljom. Uz već nabrojano i to je moguće prikazati uz pomoć matrice susjedstva. Taj podatak se nalazi na dijagonali. Kod ne težinskih grafova, ako je vrijednost veća od jedan, postoje više takvih petlji oko istog vrha. Broj vrijednosti u retku ili stupcu predstavlja broj stupnjeva vrha. Glavna prednost matrica susjedstva je da olakšava i na vrlo jednostavan način omogućuje korištenje grafova u računalu.

Prioritetni red se najviše koristi na Internetu kod obrade paketa u usmjerivačima. Protokoli koji rade u stvarnom vremenu poput prijenosa zvuka i/ili slike, gdje kašnjenje nije dozvoljeno imaju veći prioritet i potrebno ih je prve obraditi i preusmjeriti na usmjerivačima. Upravo je to svrha ATP prioritetnog reda.

Hrpa je specijalna vrsta stabla koja ima jedan vrh za korijen i mora se pridržavati definiranih pravila. Hrpa je zapravo. Sortiranje pomoću hrpa je najefikasnije sortiranje.

Najmanje impresivnim pokazao se skup jer njegova implementacija je poprilično kompliciranа, a operacije koje se danas nude najčešće se koriste u bazama podataka kroz upite između relacija. Korištenje implementacija naprednih struktura podataka sve više postaje temeljni element programiranja. Time se osigurava da se programeri maksimalno posvete

onome što trebaju programirati. ATP pruža i neovisnost od implementacije čime se osigurava fleksibilnost i pouzdan razvoj.

7. Literatura

1. Aho A.V., Hopcroft J.E., Ulman J.D., Data Structures and Algorithms, 2nd edition. Addison− Wesley, Reading MA, 1987.
2. Cormen, T. H., Leiseron, C. E., Riivest, R. L., Clifford Stein Introduction Algorithms, third edition 2009.
3. Divjak, B., Lovrenčić, A., Diskretna matematika s teorijom grafova, Fakultet organizacije i informatike, Varaždin, 2005, str. 133.
4. Divjak, B., Prezentacije s kolegija Diskretne strukture s teorijom grafova, 2008, DSTG7 grafovi.
   Dostupno 25.10.2011. na http://servisi.foi.hr/elf2008/mod/resource/view.php?id=12048
5. Drozdek A., Data Structures and Algorithms in C++, 3rd edition. Thompson Course Technology, Boston MA, 2004.
6. Goodrich M.T., Tamassia R., Algorithm Design - Foundations, Analysis, and Internet Examples.John Wiley & Sons, New York, 2002.
7. Goodrich, M. T., Tamassia, R., MIT, CS 61B: Lecture 25 Priority Queues 25.08.2010. Dostupno 25.10.2010. na http://www.cs.berkeley.edu/~jrs/61b/lec/25
8. Goodrich M.T., Tamassia R., Mount D., Data Structures and Algorithms in C++. John Wiley & Sons, New York, 2004.
9. Horowitz E., Sahni S., Anderson-Freed S., Fundamentals of Data Structures in C. W.H. Freeman & Co., New York, 1992.
10. Horowitz E., Sahni S., Mehta D., Fundamentals of Data Structures in C++, 2nd edition. Silicon Press, Summit NJ, 2006.
11. Horowitz E., Sahni S., Rajasekaran S., Computer Algorithms / C++. Computer Science Press,New York, 1997.
12. Hubbard J.R., Schaum's Outline of Data Structures with C++. McGraw-Hill, New York, 2000.

13. Knuth, D. E., The Art of Computer Programming: Fundamental Algorithms, Addison-Wesley, 1973.

14. Knuth, D. E., The Art of Computer Programming: Sotring and Searching, Addison-Wesley, 1973.

15. Korolev, D., Power up C++ with the Standard Template Library, 1996. Dostupno 25.10.2011. na http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=standardTemplate Library

16. Kruse R.L., Leung B.P., Tondo, C.L., Data Structures and Program Design in C, 2nd edition. Prentice-Hall, Englewood NJ, 1996.

17. Loudon K., Mastering Algorithms with C. O'Reilly, Sebastopol CA, 1999.

18. Manger, R., Marušić, M., Strukture podataka i algoritmi, Prirodoslovno Matematički Fakultet, 2007. Dostupno 25.10.2011. na http://web.studenti.math.hr/~manger/spa/skripta.pdf

19. Preiss B.R., Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, New York, 1999.

20. Sahni S., Data Structures, Algorithms, and Applications in C++, 2nd edition. Silicon Press, Summit NJ, 2004.

21. Skiena, S. S., The Algorithm Design Manual, Springer 2nd Edition Nov. 2010.

22. Soić, Skupovi, Struktura podataka i algoritmi ostupno 25.10.2010. na http://lnr.irb.hr/soya/nastava/predavanje07-cb-5.pdf

23. Weiss M.A., Data Structures and Algorithm Analysis in C++, 3rd edition. Addison-Wesley, Reading MA, 2006.

24. Zhu, M. M., Southern Illinois University 23.09.2009 Dostupno 25.10.2011. na http://www2.cs.siu.edu/~mengxia/Courses%20PPT/220/carrano_ppt18.ppt