Implementacija apstraktnih tipova podataka u programskom jeziku C++

Oblak, Domagoj

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

DOMAGOJ OBLAK

IMPLEMENTACIJA APSTRAKTNIH TIPOVA PODATAKA U PROGRAMSKOM JEZIKU C++

Završni rad

Pula, rujan, 2018. godine

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

DOMAGOJ OBLAK

IMPLEMENTACIJA APSTRAKTNIH TIPOVA PODATAKA U PROGRAMSKOM JEZIKU C++

Završni rad

JMBAG: 0303061234, redoviti student
Studijski smjer: Informatika

Predmet: Strukture podataka i algoritmi
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor: doc. dr. sc. Tihomir Orehovački

Pula, rujan, 2018. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Domagoj Oblak, kandidat za prvostupnika informatike ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, rujan, 2018. godine

IZJAVA
o korištenju autorskog djela

Ja, Domagoj Oblak dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom „Implementacija apstraktnih tipova podataka u programskom jeziku C++“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.
Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, rujan, 2018. godine

Potpis

_____

Sažetak

Tema ovog završnog rada je „Implementacija apstraktnih tipova podataka u programskom jeziku C++“. Detaljno su opisani apstraktni tipovi podataka i operacije koje se izvršavaju nad njima. Oni nisu ugrađeni u programskom jeziku već ih programer sam implementira, a implementacije se razlikuju po strukturi i algoritmima za izvršavanje pojedine operacije, tako svaki apstraktni tip podatka može imati više implementacija. Dijele se na četiri vrste, a to su linearni, hijerarhijski, dvodimenzionalni nehijerarhijski i skupovni. U programskom jeziku C++ će biti implementirani svi navedeni apstraktni tipovi te na kraju svakog poglavlja određena složenost svake funkcije.

Ključne riječi: *apstraktni tip podataka, programski jezik C++, implementacija, složenost algoritama, vezana lista, dvostruko vezana lista, stog, red, stablo, binarno stablo, graf, skup, rječnik, prioritetni red,*

Abstract

In this bachelor's thesis „Implementation of abstract data types in C++ programming language“ the abstract data types and operations that are executed over them are described in detail. They are not embedded in the programming language, but are implemented by the developer, and implementations vary by structure and algorithms to execute an operation, so any abstract data type can have multiple implementations. They are divided into four types, linear, hierarchical, two-dimensional non-hierarchical and sets. In the C ++ programming language, all of the above mentioned types will be implemented, and at the end of each chapter will determine the complexity of each function.

Keywords: *abstract data type, C++ programming language, implementation, complexity, linked list, doubly linked list, stack, queue, tree, binary tree, graph, set, dictionary, priority queue*

Sadržaj

1. Uvod

Kroz povijest, apstraktni tipovi podataka su se konstantno razvijali kako bi se našlo što efikasnije rješenje za pojedini problem. Također nagli razvoj tehnologije uvelike je potpomogao razvoju boljih i efikasnijih apstraktnih tipova podataka. Postoji nekoliko razloga zbog kojih je dobro steći znanje o apstraktnim tipovima, npr. oni predstavljaju relativno jednostavan način da se nauči terminologija i glavni aspekti programiranja. Tako se dobivaju dobri temelji za dalje analiziranje složenijih algoritama.

Kako bi se riješio neki problem u programiranju potreban je algoritam. Uz algoritam pri rješavanju problema potrebna je i struktura podataka koja se mora prilagoditi kako bi se problem riješio efikasnije.

Tipovi podataka općenito opisuju neki skup objekata. Svaki programski jezik sadrži osnovne vrste, npr. integer, float i char, a koriste se za opisivanje skupa objekata koji se prikazuju na isti način. Apstraktni tipovi podataka nisu ugrađeni u programskom jeziku već ih programer sam implementira, te imaju više tipova podataka i funkcija.

Apstraktni tipovi kao i oni koji su već implementirani u samom programskom jeziku dijele se na nekoliko vrsta. Prva vrsta su linearni tipovi, točnije lista, stog i red kod kojih se većina funkcija obavlja sekvencijalno. Hijerarhijski tipovi predstavljaju drugu vrstu, a najpoznatiji tip je stablo gdje se koristi poznati odnos „podređeni-nadređeni“ tj. „roditelj-dijete“. Treća vrsta je dvodimenzionalni nehijerarhijski graf koji može biti usmjeren i neusmjeren, a predstavlja uređeni par čvorova i grana. Zadnja vrsta su skupovi koji mogu biti implementirani na više načina i mogu koristiti za široku upotrebu, najpoznatiji primjeri su rječnik i prioritetni red.

Da bi implementacija apstraktnog tipa podatka bila moguća moramo definirati strukturu podataka kako bi mogli prikazati podatke te funkcije kojima se operacije ostvaruju. Implementacije se razlikuju po strukturi i algoritmima za izvršavanje pojedine operacije, tako svaki apstraktni tip podatka može imati više implementacija. Ne postoji savršena implementacija za pojedini apstraktni tip, već se primjenjuje ona koja ovisi o broju operacija koje se najčešće izvode i da pri tome da najbolje rezultate.

Cilj ovog rada je uspješno objasniti sve funkcije pojedinog apstraktnog tipa podataka kako bi se odabrao onaj tip koji je najpogodniji za određeni zadatak.

Prvi dio rada se bavi algoritmima i njihovom složenošću, gdje je svrha optimizirati postojeći i razviti što učinkovitiji algoritam, a testiranje se vrši na osnovu ponašanja u najboljem i najgorem slučaju.

U drugom poglavlju opisuju se linearni apstraktni tipovi podataka te pojedine operacije koja su primjenjuju nad određenom apstraktnom tipu. Nakon opisa slijedi implementacija u programskom jeziku C++ te određivanje složenosti.

Treće poglavlje predstavlja hijerarhijski uređene apstraktne tipove gdje će biti objašnjena implementacija i složenost za svaki tip.

Četvrto poglavlje govori o grafovima koji se mogu primijeniti u mnogo različitih slučajeva te će također biti predstavljena implementacija i složenost u programskom jeziku C++.

U petom poglavlju opisuju se skupovi kao strukture podataka te njihova implementacija i složenost.

U posljednjem poglavlju bit će riječi o komparaciji apstraktnih tipova podataka prema implementaciji.

2. Složenost algoritama

Pri definiranju algoritma moramo paziti na dva glavna resursa računala, a to su vrijeme i memorija. Algoritmu kojem treba dugo vremena da bi se završio i potrošio nekoliko gigabajta radne memorije nije baš mnogo koristan, iako je u stvarnosti ispravan. Odrediti vrijeme izvršavanja algoritma može na dva načina, točnije na empirijski i analitički način. Empirijskim načinom samo izmjerimo stvarno vrijeme izvršavanja pojedinog algoritma, dok se analitičko određivanje zasniva na teorijskoj analizi rada algoritama i prebrojavanju pojedinih instrukcija koje se izvršavaju pri radu algoritma (Živković, 2010).

Algoritam mora biti precizno i jednoznačno definiran da riješi neki problem, također treba biti djelotvoran, da se u konačnom vremenu dobije rezultat. Učinkovitost je bitan faktor pri definiranju algoritma, točnije trebamo dobiti dobre rezultate s obzirom na utrošene resurse. Implementacija apstraktnih tipova podatak može se napraviti na različite načine, kako bi se utvrdilo koja je implementacija bolja, potrebno je uvesti mjeru koja će mjeriti kvalitetu algoritama, za to nam služi mjera složenosti algoritama, a može se definirati na dva načina: prema prostoru potrebnom da se podaci pohrane u memoriji, te prema vremenu potrebnom da se izvedu operacije nad podacima (Manger, 2013).

Svrha analize složenosti je optimizirati postojeći i razviti što učinkovitiji algoritam. Testiranje se vrši na osnovu ponašanja u najboljem i najgorem slučaju. *Apriori* analiza algoritama procjenjuje vrijeme izvođenja algoritma neovisno o korištenom programskom jeziku ili prevoditelju, dok *aposteriori* koristi stvarno vrijeme koje je potrebno za izvođenje algoritama na određenom računalu. Asimptotska analiza algoritama ima za cilj danu funkciju $f(n)$ pronaći funkciju $g(n)$ koja aproksimira $f(n)$ pri većim vrijednostima od n (Manger, 2013).

Postoje tri vrste notacija za analizu složenosti, a to su O (big-Oh), Ω (big-Omega) i Θ (big-Theta). Kod big-Oh notacije gleda se gornja granica tj. najgori slučaj izvođenja nekog algoritma, a kod big-Omega uzima se u obzir samo donja granica (najbolji slušaj) izvođenja algoritma, dok je kod big-Theta gornja i donja granica izvođenja algoritma jednaka (Manger, 2013).

Kao što je rečeno u prethodnom odlomku približno vrijeme izvršavanja naziva se asimptotsko vrijeme izvršavanja, ono je u stvari mjera brzine rasta točnog vremena izvršavanja algoritma sa povećanjem broja ulaznih podataka. Kod analize složenosti algoritama logaritam se uvijek uzima na s bazom 2. Brzina algoritma je jako bitna kada je riječ o velikom broju ulaznih podataka, jer se skoro kod svakog algoritma vrijeme povećava pri dodavanju više ulaznih podataka. Stoga za rješavanje istog problema moramo razmisliti koji algoritam odabrati, jer je važno vrijeme za koje će riješiti problem kada je broj ulaznih podataka vrlo velik. Većina algoritama je efikasna za mali broj ulaznih podataka i uglavnom nije bitno da li će neki algoritam raditi jednu milisekundu ili deset milisekundi duže kada je riječ o nekoliko ulaznih podataka. Ta dva vremena su svakako neprimjetna za čovjeka, ali ako algoritam radi 10 minuta ili 2 mjeseca kada je riječ o milijunu ulaznih podataka to jest svakako značajno za odluku koju ćemo donesti pri rješavanju tog problema (Živković, 2010).

A1, A2 i *A3* su algoritmi koji imaju vrijeme izvršavanja 2^n, $5n^2$ i 100n, gdje n predstavlja veličinu ulaznih podataka. Vrijednosti ovih funkcija vide se u lijevoj tablici 1, ako se svaki algoritam obavlja jedan milion (10^6) instrukcija u sekundi tada će svaki algoritam biti izvršen u stvarnom vremenu čije se vrijednost vide u tablici 2.

n	2^n	$5n^2$	*100n*
1	2	5	100
10	1024	500	1000
100	2^{100}	50000	10000
1000	2^{1000}	$5x10^6$	100000

Tablica 1. Vrijednost funkcija (Živković, 2010, str. 46)

U tablici 2 vidimo da za veliko *n* (npr. *n > 100*) algoritam *A1* je najsporiji, zatim algoritam *A2* srednji, dok je algoritam *A3* najbrži. Znači, oblici funkcija od *n* (2^n, n^2 *i n)* su važniji od konstantnih faktora (tj. 1, 5, 100) tih funkcija. Najjednostavniji izraz složenosti algoritama *A1, A2 i A3* bio bi, redom, $T_1(n)=2^n$, $T_2(n)=n^2$ *i* $T_3(n)=n$ (Živković, 2010).

n	*A1*	*A2*	*A3*
1	1 µs	5 µs	100 µs
10	1 ms	0,5 ms	1 ms
100	2^{70} g	0,05 s	0,01 s
1000	2^{970} g	5 s	0,1 s

Tablica 2. Vremena izvršavanja tri algoritma (Živković, 2010, str. 46)

U nastavku slijedi primjer analize vremena izvršavanja algoritma i određivanje *T(n).* Pogledajmo algoritam koji računa produkt *p* dva realna vektora *a* i *b* duljine *n.*

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

Analiziranjem ovog algoritma broje se operacije množenja, zbrajanja i pridruživanja realnih brojeva. Brojač petlje možemo zanemariti jer je manje zahtjevan, a ionako se pojavljuje uz operacije s realnim brojevima. Algoritam se sastoji od *n* množenja, *n* zbrajanja i *n+1* pridruživanja, stoga je njegova složenost *T(n) =3n + 1* (Manger, 2013).

- Algoritmi s vremenom *T(n)=O(1)* su ograničeni konstantom, tj. vrijeme ne ovisi o veličini ulaznih podataka i vrlo su upotrebljivi i poželjni.
- Algoritmi s vremenom *T(n)=O(log n)* su vrlo brzi jer im vrijeme izvršavanja raste prilično sporije nego skup ulaznih podataka.
- Algoritmima s vremenom *T(n)=O(n)* vrijeme izvršavanja raste linearno s veličinom ulaznih podataka.
- Algoritmima s vremenom *T(n =O(n log n), T(n)=O(n^2) ili T(n)= O(n^3)* izvršavanja raste neproporcionalno s obziru na veličinu ulaznih podataka.
- Algoritmi s vremenom *T(n)=O(n^k)* upotrebljivi su samo kada je *k* mali jer u stvarnosti znaju biti spori.
- Algoritmi s vremenom *T(n)=O(n^l)* su upotrebljivi samo za male *n-ove* (Manger, 2013).

O notacija koristi se kako bismo točno znali da li je neka funkcija „manja“ od druge. U rezultatu dobijemo samo gornju granicu vremena izvršavanja što nam ide u korist jer vrijeme rada algoritma određujemo za najgori slučaj.

Pri analiziranju složenijih algoritama moramo poznavati relativne odnose osnovnih funkcija, kao na primjer:

- Izraz n^a ima prednost nad n^b za $a > b$

Brzina algoritma je sve bolja ako se njegova funkcija vremena izvršavanja nalazi što bliže vrhu tablice 3, tj. algoritam se brže izvodi ako njegova funkcija vremena izvršavanja „manja od“ funkcije izvršavanja drugog algoritma. U tablici 3 vidimo tipične funkcije koje se koriste pri određivanju vremena izvršavanja pojedinog algoritma. Za dovoljno veliki *n* vrijedi : $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \ldots < O(2^{\log n}) < O(2^n) < O(n!)$ (Živković, 2010).

Naziv	Oznaka
Konstantan	$O(1)$
Logaritamski	$O(\log n)$
Linearan	$O(n)$
Linearno-logaritamski	$O(n \log n)$
Kvadratni	$O(n^2)$
Stupanjski	$O(n^m)$, $m > 2$
Podeksponentni	$O(m^{\log n})$, $m > 1$
Eksponentni	$O(m^n)$, $m > 1$
Faktorijelni	$O(n!)$

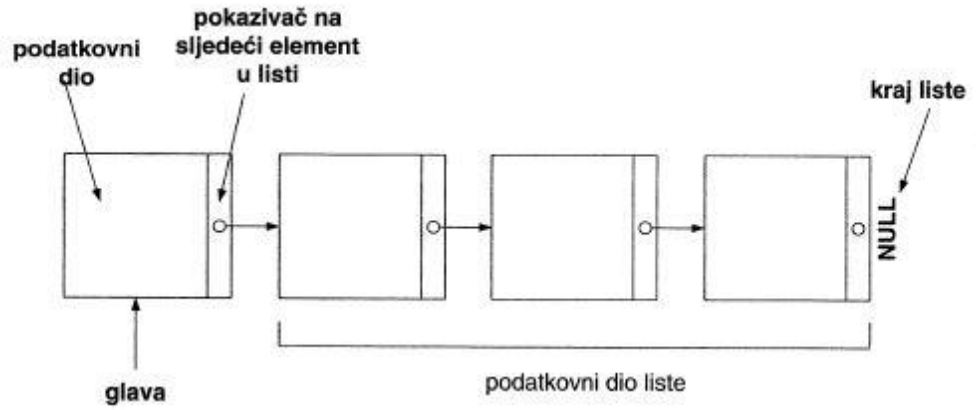
Tablica 3. Kategorizacija algoritama

3. Apstraktni tip podataka lista

Lista (vezana lista) koristi strukturu podataka koja je slična nizu jer predstavlja sekvencijalni niz elemenata koji su istog tipa. Ali za razliku od niza gdje se struktura dobije indeksiranjem elemenata, kod liste se takav poredak određuje pokazivačima koje sadrži svaki element (Živković, 2010).

Lista se ubraja pod linearne apstraktne tipove podataka, gdje je dozvoljeno izbacivanje i ubacivanje podataka na bilo kojem mjestu u tom nizu. Ima dinamičku strukturu podataka jer broj elemenata nije unaprijed određen. Pristup elementima ide sekvencijalno (slijedno), dok se elementima kod polja može izravno pristupiti, uz pomoć indeksa. Elementi liste se grupiraju u prvi slobodni dio memorije, a kada pojedini element više nije nužan zauzeti dio memorije postaje slobodan. Prvi element u listi naziva se glava liste i ima ulogu da sačuva početak liste. Element liste zapravo predstavlja strukturu koja se sastoji od podatkovnog dijela, pokazivača na sljedeći element i ako se radi o dvostruko vezanoj listi pokazivač na prethodni element (Manger, 2013).

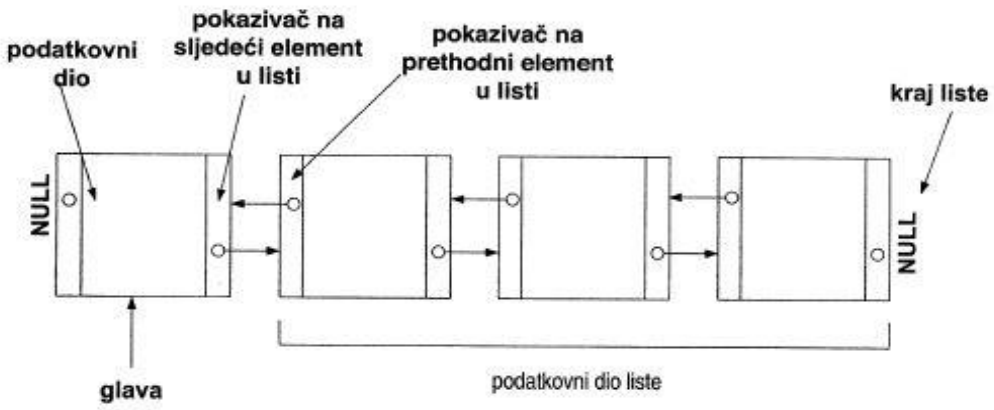
Slika 1 prikazuje listu s podatkovnim dijelom liste čiji početak počinje glavom liste, a završava s vrijednošću NULL, što označava zadnji element, NULL služi za izlazak iz iteracije kada dođe kraj liste.



Slika 1. Primjer vezane liste (Radošević, 2007, str. 30)

Dvostruko vezana lista je rješenje za liste kod kojih se treba pristupiti prethodnom elementu, ona osim jednog pokazivača sadrži i drugi koji pokazuje na prethodni element u listi.

Slika 2 prikazuje dvostruko vezanu listu koja ima sličnu strukturu kao i vezana lista, gdje se vidi podatkovni dio liste čiji početak počinje glavom liste, a završava s vrijednošću NULL, što označava zadnji element. Svaki element sadrži pokazivač za sljedeći i prethodni element u listi.



Slika 2. Primjer dvostruko vezane liste (Radošević, 2007, str. 40)

Funkcije ATP lista (list):

- listaFirst(L) – funkcija vraća prvi element liste
- listaEnd(L) – funkcija vraća mjesto iza zadnjeg elementa liste
- listaNext(p,L) – funkcija vraća sljedbenika elementa p
- listaPrevious(p,L) – funkcija vraća prethodnika elementa p
- listaLocate(x,L) – funkcija vraća prvi element liste čija je vrijednost jednaka x
- listaInsert(x,p,L) – dodaje se novi element na p-tu poziciju liste
- listaDelete(p,L) – briše se element s pozicije p iz liste L
- listaRetreive(p,L) – funkcija koja vraća vrijednost elementa liste L na poziciji p
- listaDeleteAll(L) – funkcija koja briše sve elemente iz neke postojeće liste L
- listaInit(L) – iniciranje prazne liste L

3.1. Implementacija liste

Implementacija liste pomoću polja zasniva se na strukturi podataka koja je ugrađena u programski jezik C++. Kursor *last* pokazuje na zadnji element, što znači da je lako pročitati, ubaciti ili obrisati kraj liste. Također pristup *i-tom* elementu je lagan jer je svakom elementu pridružen njegov indeks. Ubacivanje i izbacivanje zahtjeva pomicanje tj. prepisivanje dijela podataka. Implementacija liste pomoću polja uz navedene prednosti ima i nekoliko mana koje ograničavaju ovu implementaciju. Najuočljivija mana je da se polje može lako prepuniti, stoga je pogodna za koristiti kada nema mnogo ubacivanja i izbacivanja u sredini liste (Manger, 2013).

Jedna od mana implementacije pomoću pokazivača je ta da troši više memorije po elementu jer uz svaki podatak ima i pokazivač koji pamti poziciju sljedećeg elementa. Također je spor pristup *i-tom* elementu, odnosno zadnjem i prethodnom elementu. Uz sve mane navedena implementacija ima i nekoliko prednosti, npr. kod ubacivanja ili brisanja elementa nema potrebe za prepisivanjem drugih podatka. Također se zaobilazi problem prepunjenja liste koji postoji kod implementacije pomoću polja (Manger, 2013).

Implementacija pomoću pokazivača listu prikazuje kao niz klijetki, a svaka klijetka predstavlja jedan element liste. Funkcija insertL() dinamičkom alokacijom memorije stvara novu klijetku gdje se upisuje novi element te uključuje u listu na poziciji *p*, točnije nova klijetka ide ispred klijetke na poziciji *p*. Funkcija deleteL() prvo mijenja pokazivač klijetke koja se nalazi na poziciji *p* i dinamičkom dealokacijom se briše određeni element. InitL() funkcija stvara novu klijetku i u nju upisuje pokazivač *NULL,* te adresu upisuje u varijablu. Kod funkcija firstL(), nextL() i retrieveL() implementacija je trivijalna i svodi se samo na pridruživanje. U nastavku biti će prikazana implementacija pomoću pokazivača.

```
typedef int elementtype;

struct element{
      elementtype vrijednost;
      element *sljedeci;
};

typedef element *lista;
```

```
element listaFirst (lista L)
{
	return L;
}

element listaEnd (lista L)
{
	element temp = L;
	while(temp->sljedeci)
	{
		temp = temp->sljedeci;
	}
	return temp;
}

element listaNext (element p, lista L)
{
	return p->sljedeci;
}

element listaPrevious (element p, lista L)
{
	if(p == firstL(L))
		cout << "Funkcija je nedefinirana" << endl;
	element temp = L;
	while(temp->sljedeci!=p)
		temp = temp->sljedeci;
	return temp;
}

element listaLocate (element x, lista L)
{
	element temp = new element;
	while(temp != endL(L))
	{
		if(temp->sljedeci == x)
			return temp;
		temp = nextL(temp,L);
	}
	return endL(L);
}

int listaInsert (element x, element p, lista L)
{
	element novi = new element;
	novi->sljedeci = p->sljedeci;
	p->sljedeci = novi;
	novi = x;
	if( (p->sljedeci) == novi) return 1;
	else false;
}

int listaDelete (element p, lista L)
{
	if(p == endL(L)) return 0;
	element del = p->sljedeci;
	p->sljedeci = del->sljedeci;
	delete del;
	return 0;
}
```

```
element listaRetrieve (element p, lista L)
{
	return p->sljedeci;
}

void listaDeleteAll (lista *L)
{
	element del = L->sljedeci, iduci;
	while(del)
	{
		iduci = del->sljedeci;
		delete del;
		del = iduci;
	}
	L->sljedeci = NULL;
}

lista listaInit (lista L)
{
	lista L = new lista;
	L->sljedeci = NULL;
	return L;
}
```

Implementacija 1. Lista

Tablica 4 prikazuje funkcije apstraktnog tipa podataka liste gdje je ubacivanje i izbacivanje elemenata vrlo je jednostavno, dok je čitanje malo teže jer je potrebno čitati redom svaki element. Također teže je odrediti kraj ili prethodni element.

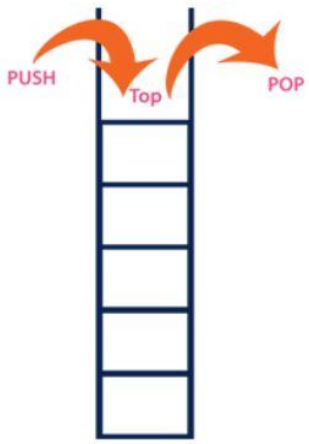
Funkcija	Složenost
listaFirst	O(1)
listaEnd	O(n)
listaNext	O(1)
listaPrevious	O(n)
listaLocate	O(n)
listaInsert	O(1)
listaDelete	O(1)
listaRetrieve	O(1)
listaDeleteAll	O(n)
listaInit	O(1)

Tablica 4. Složenost liste (Manger, 2013, str. 23)

4. Apstraktni tip podataka stog

Stog (eng. stack) je posebna vrsta liste gdje se ubacivanje i izbacivanje obavlja samo na jednom kraju, zato element koji je zadnji ubačen prvi će biti izbačen. Kraj stoga gdje se obavljaju sve operacije zove se vrh stoga, a tamo gdje je prvi element koji je ušao naziva se dno stoga. Stog je poznat još kao *LIFO* lista (eng. 12asti n first out). Operacije upisa, čitanja i brisanja mogu se izvršavati samo na vrhu stoga, dok se ostatku elemenata može pristupiti samo brisanjem elemenata koji su kasnije upisani (Manger, 2013).

Slika 3 prikazuje stog gdje se pristup odvija preko vrha koji je promjenjiv i on ukazuje na posljednji dodani element. Neki primjeri stoga su: složeni tanjuri u restoranu, spremnik za metke kod pištolja, program koji se sastoji od više procedura, stog CD-ova itd. (Živković, 2010).



Slika 3. Primjer stoga (Btechsmartclass, 2015)

Funkcije ATP stoga (stack):

- stogTop(S) – funkcija koja vraća vrijednost koja se nalazi na vrhu stoga
- stogPush(x,S) – funkcija koja dodaje novi element na vrh stoga
- stogPop(S) – funkcija koja briše element vrha stoga
- stogInit(S) – funkcija koja inicira prazan stog
- stogIsEmpty(S) – funkcija koja odgovara da li je stog prazan

4.2. Implementacija stoga

Implementacija stoga pomoću polja koristi strukturu kao i lista koja je opisana u prethodnom poglavlju, samo što kursor *top* pokazuje na vrh stoga. Prilikom ubacivanja ili izbacivanja ne moramo prepisivati ostale elemente jer podatke ubacujemo u „donji“ dio polja. Znači stog raste prema gore tj. prema manjim indeksima. Kod navedene implementacije može doći nekoliko problema, npr. kada dođe do prepunjenja jer nema mjesta te ako želimo dodati ili obrisati element, a stog je prazan (Manger, 2013).

Implementacija stoga pomoću pokazivača bazira se na vezanoj listi, struktura je slična kao i kod općenite liste, s malim promjenama. U strukturi više ne postoji pojam pozicije, pa nije više potrebno zaglavlje. Struktura se sastoji od podatka koji sadržava neku vrijednost i pokazivača na sljedeći element. Prednost ove implementacije je ta da izbjegava prepunjenost jer koristi dinamičku alokaciju za stvaranje novog elementa.

Funkcija Push() dinamičkom alokacijom stvara novu klijetku te joj pridružuje vrijednost x. A funkcija Pop() briše klijetku iz vezane liste, kod liste se to odvija u sredini, dok kod stoga na početku liste tj. vrhu stoga. Funkcijom Init(), pridružuje pokazivač *NULL* i stvara novu klijetku. IsEmpty() provjerava da li je *S* jednako *NULL*, a Top() vraća element koji je na vrhu stoga.

```
typedef int elementtype;

struct element
{
      elementtype vrijednost;
      element *sljedeci;
};

typedef element* stog;

elementtype stogTop(stog S)
{
      return S->vrijednost;
}

void stogPush(elementtype x, stog *S)
{
      element *novi = new element;
      novi->vrijednost = x;
      novi->sljedeci = *S;
      *S = novi;
}
```

```
void stogPop(stog *S)
{
	element *t = *S;
	*S = t->sljedeci;
	delete t;
}

void StogDeleteAll(stog *S)
{
	element *t, *t1;
	t = *S;
	while(t != NULL)
	{
		t1 = t;
		t = t->sljedeci;
		delete t1;
	}
	*S = NULL;
}

bool stogIsEmpty(stog S)
{
	if (S == NULL)
	{
		return 1;
	}
	return 0;
}

void stogInit(stog *S)
{
	*S = NULL;
}
```

Implementacija 2. Stog

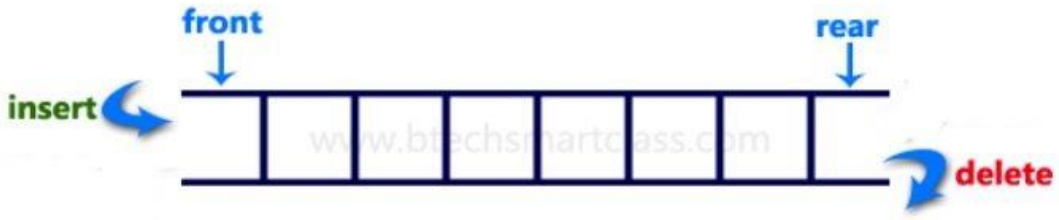
Tablica 5 prikazuje složenost stoga kada je riječ o implementaciji s pokazivačima. Sve operacije imaju složenost O(1) jer se obavljaju na vrhu stoga i nije potrebno prolaziti kroz sve elemente.

Funkcija	Složenost
stogTop	O(1)
stogPush	O(1)
stogPop	O(1)
stogInit	O(1)
stogIsEmpty	O(1)

Tablica 5. Složenost stoga (Manger, 2013, str. 30)

5. Apstraktni tip podataka red

Red (eng. queue) je kao i stog posebna vrsta liste gdje se elementi ubacuju na kraju, a izbacuju na početku, što znači prvi element koji je ubačen, prvi će biti izbačen. Još je poznat kao *FIFO* lista (eng. first in first out). Na slici 4 vidi se primjer reda gdje *front* pokazuje na početak, a rear na kraj reda. Najjednostavniji primjer reda je red ljudi koji stoje pred blagajnom u kinu, menzi ili supermarketu. Prva osoba koja dođe na blagajnu prva i odlazi, što znači da je podržano svojstvo reda (Goodrich, M, Tamassia, R, Mount, D, 2011).

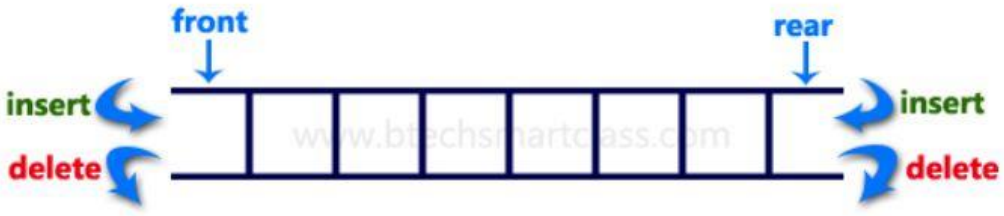


Slika 4. Primjer reda (Btechsmartclass, 2015)

Red s dva kraja koji je vidljiv na slici 5, a odnosi se na jednu od posebnih modifikacija reda gdje se operacije dodavanja i brisanja mogu koristiti i na početku i kraju reda.

Ovakav red ima tri vrste, a to su:

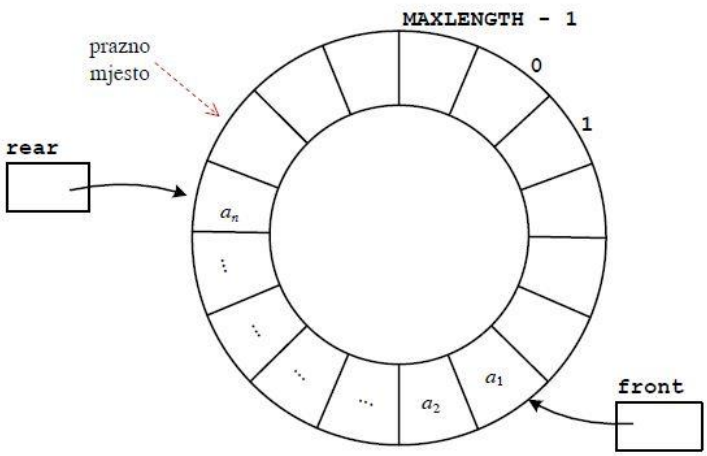
- red sa čitanjem i pisanje na oba kraja
- red s ograničenim ulazom
- red s ograničenim izlazom



Slika 5. Red s dva kraja (Btechsmartclass, 2015)

Kod reda s čitanjem i pisanjem na oba kraja moguće je koristiti operacije dodavanja i brisanja na oba kraja, dok je kod reda s ograničenim ulazom na jednoj strani moguće su obje operacije, a na drugoj samo brisanje. Kod zadnje vrste su također moguće na jednoj strani obje operacije, a na drugoj samo dodavanje (Btechsmartclass, 2015)

Slika 6 prikazuje kružni (cirkularni) red koji ima takvu strukturu da prvi element niza slijedi odmah iza posljednjeg. Npr. red *q* ima kursor *f* koji pokazuje na prvi element tj. glavu (front) te kursor *r* koji pokazuje na posljednji element niza (rear). Kako bi razlikovali kada je red pun, a kada prazan uvodi se jedno prazno mjesto između početka i kraja reda. Što znači, da nikad ne smije biti više od *MAXLENGHT – 1* (Živković, 2010).



Slika 6. Kružni red (Manger, 2013, str. 39)

Funkcije ATP reda (queue):

- redFront(Q) – Funkcija vraća vrijednost koja se nalazi na početku reda
- redEnQueue(x,Q) – Funkcija koja dodaje novi element na kraj reda
- redDeQueue(Q) – Funkcija koja briše element s početka reda
- redInit(Q) – Funkcija koja inicira prazan red
- redIsEmpty(Q) – Funkcija koja provjerava da li je red prazan

5.2. Implementacija reda

Implementacija reda pomoću kružnog polja je ista kao i kod liste samo što se početak reda naziva čelo. Kako bismo izbjegli prepisivanje podataka pri unošenju novog elementa uvodimo još jedan kursor koji pamti trenutni položaj čela. Kružni red ima strukturu gdje nakon zadnjeg elementa slijedni početni. Kursor *front* pokazuje na čelo reda, a kursor *rear* na začelje, pri ubacivanju elemenata red se kreće u smjeru kazaljke na satu. Kako bi se razlikovalo kad je red pun, a kad prazan uvodi se prazno mjesto između prvog i zadnjeg elementa. Kao i u prethodnim implementacijama s poljima, ova ima problema kada ubacujemo novi element, gdje može doći do prepunjenja te kada pokušamo čitati ili brisati prazan red (Manger, 2013).

Navedena implementacija može se izvesti pomoću kružnog (cirkularnog) polja i pokazivača, efikasnost im je ista jer se sve operacije izvršavaju u vremenu O(1). Jedna od prednosti implementacije s pokazivačima je to što zaobilazi problem prepunjenog reda jer ima dinamičku strukturu kod koje ne moramo unaprijed odrediti broj elemenata. Početak predstavlja čelo i u kodu je implementiran kao pokazivač *front*, dok je zadnji dio prikazan kao pokazivač *rear* (Manger, 2013).

Funkcija InitQ() samo stvara zaglavlje reda i postavlja pokazivače *front* i *rear* da pokazuju na njega. Funkcija IsEmptyQ ispituje da li je red prazan tako da provjeri da li pokazivači *front* i *rear* pokazuju na zaglavlje reda. EnqueueQ() stvara novu klijetku i preko pokazivača *rear* priključuje na kraj reda. Funkcija DequeueQ() izbacuje prvi element tako da pokazivač *front* premjesti na sljedeću klijetku koja će predstavljati početak. Zadnja funkcija apstraktnog tipa podataka reda je FrontQ(), a ona čita prvi element reda preko pokazivača *front*.

```
typedef int elementtype;

struct element
{
      elementtype vrijednost;
      element *sljedeci;
};

struct red
{
      element *front, *rear;
};
```

```
void redInit (red *Q)
{
	Q->front = NULL;
	Q->rear = NULL;
}

bool redIsEmpty (red Q)
{
	if(Q.front == NULL && Q.rear == NULL)
	{
		cout << "Red je prazan!" << endl;
		return 1;
	}
	return 0;
}

elementtype redFront (red Q)
{
	return Q.front->vrijednost;
}

void redEnqueue (elementtype x, red *Q)
{
	element *novi = new element;
	novi->vrijednost = x;
	novi->sljedeci = NULL;
	if(Q->front == NULL)
	{
		Q->front = novi;
		Q->rear = novi;
		return;
	}
	Q->rear->sljedeci = novi;
	Q->rear = novi;
}

void redDequeue (red *Q)
{
	element *t;
	t = Q->front;
	Q->front = t->sljedeci;
	delete t;
	if(Q->front == NULL)
		Q->rear = NULL;
}
```

Implementacija 3. Red

Tablica 6 prikazuje složenost pojedine funkcije kod implementacije reda s pokazivačima, efikasnost im je ista jer se sve operacije izvršavaju u vremenu O(1).

Funkcija	Složenost
redFront	O(1)
redEnqueue	O(1)
redDequeue	O(1)
redInit	O(1)
redIsEmpty	O(1)

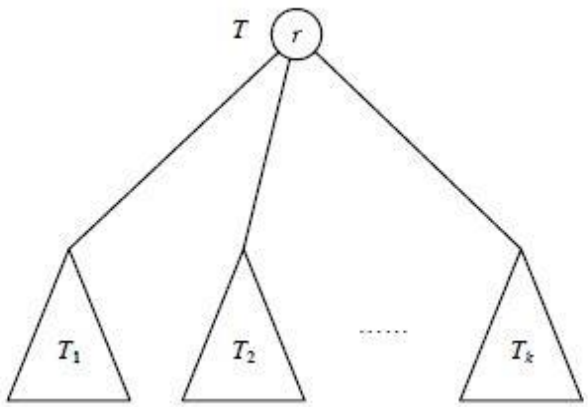
Tablica 6.Složenost reda (Manger, 2013, str. 41)

6. Apstraktni tip podataka stablo

Za razliku od liste, stoga i reda koji koriste linearnu strukturu, stablo je zasnovano na hijerarhijskoj strukturi podataka koja je poznata koš kao odnos „podređeni-nadređeni“ odnosno „roditelj-dijete“. Sastoji se od skupa elemenata koji se nazivaju čvorovi i skupa uređenih parova različitih čvorova povezanih linijama koji se nazivaju grane (Živković, 2010).

Na slici 7 vidljiv je glavni čvor koji se zove korijen (eng. root) stabla iz kojeg se grade ostali čvorovi u konačan niz od nula ili više manjih stabala. Manja stabla se zovu podstabla korijena, korijen nema svog roditelja dok svaki od preostalih čvorova ima točno jednog roditelja (Manger, 2013).

U stablu čvorovi mogu imati nula ili više djece, ali svaki čvor osim korijena ima točno jednog roditelja dok je korijen jedini čvor bez roditelja. Stablo karakteriziraju dva svojstava: ako T nije prazno stablo, ono ima jedinstven poseban čvor (korijen) koji nema roditeljski čvor, svaki čvor *u* u *T* osim korijena ima jedinstven roditeljski čvor (Živković, 2010).



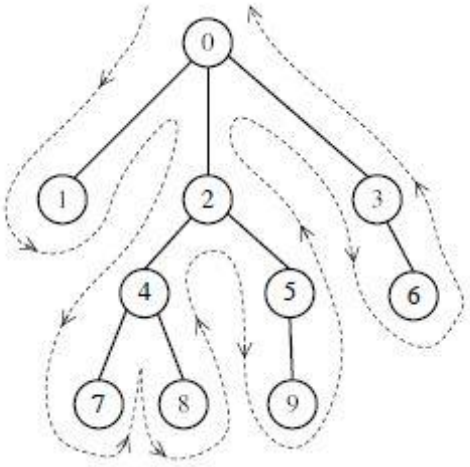
Slika 7. Primjer stabla (Manger, 2013, str. 45)

Obilazak stabla je algoritam kojim se posjećuje svaki čvor stabla tako da se svaki čvor posjeti samo jednom. Obilasci se zasnivaju na rekurzivnoj funkciji, a najčešći obilasci su preorder, inorder i postorder Na slici 8 vidljiv je dijagram stabla obrubljen crtkanom linijom koja kreće s lijeve strane korijena i prolazi pokraj svih čvorova (s lijeve i desne strane) te završava s desne strane korijena (Manger, 2013).

Pomoću tri pravila vrlo lako se dođe do rješenja za svaki obilazak stabla, a ona glase:

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

- 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 8. Primjer obilazaka stabla (Manger, 2013, str. 50)

Funkcije ATP stablo (tree):

» ParentT(n,T) – funkcija vraća roditelja čvora n
» FirstChildT(n,T) – funkcija vraća prvo lijevo dijete čvora n
» NextSiblingT(n,T) – funkcija vraća sljedećeg brata čvora n
» LabelT(n,T) – funkcija vraća vrijednost koju sadrži čvor n
» RootT(T) – funkcija vraća korijen stabla
» CreateT(x,n,T) – procedura dodaje x kao dijete čvora n
» ChangeLabelT(x,n,T) – mijenja oznaku n u stablu T na vrijednost x
» DeleteT(n,T) – procedura briše čvor n i sve njegove potomke
» InitT(x,T) – inicijalizira stablo

6.1. Implementacija stabla

Jedna od mogućnosti implementacija stabla koristi se na način da svakom čvoru zapišemo njegovog roditelja. Veze bilježimo kursorima, a imena čvorova su u obliku 1, 2, 3, …. Stablo se prikazuje pomoću dva polja *label* i *parent* gdje *i-te* klijetke opisuju čvor *i* tj. u njima piše oznaka čvora. Nepostojeći čvor koristi se oznaka -1. Navedena implementacija ima neke prednosti, ali i mane. Zato ovu implementaciju je poželjno koristiti samo kada nema mnogo ubacivanja i izbacivanja čvorova ili kada nije potrebna uređenost stabla (Manger, 2013).

U ovom primjeru implementacija stabla naziva se „prvo dijete – sljedeći brat“ gdje se stablo implementira pomoću polja čiji element, osim oznake čvora sadrži i čvor koji je prvo dijete promatranog čvora te čvor koji je sljedeći brat zadanog čvora. Navedenu implementacija najkorisnija je kada ima puno dodatnih ubacivanja čvorova.

```
Struct elem
{
	char label[35];
	int firstchild, nextsib;
};

struct stablo
{
	elem element[10000];
	int first;
};

stablo t;
bool init_t = 0, prvi = 0;

void InitT (int x, stablo &t)
{
	t.first=x;
	for (int i=0; i<10000; i++)
	{
		t.element[i].firstchild = -2;
		init_t=1;
	}
}

int FirstChildT (int n, stablo &t)
{
	if(t.element[n].firstchild == -2)
	{
		return -1;
	}
	else
	{
		 return t.element[n].nextsib;
	}
}
```

```
int NextSiblingT (int n, stablo &t)
{
	if(t.element[n].firstchild == -2)
	{
		return -1;
	}
	else
	{
		return t.element[n].nextsib;
	}
}

int ParentT (int n, stablo &t)
{
	if(n==t.first||t.element[n].firstchild == -2)
	{
		return -1;
	}
	for (int i=0; i<10000; i++)
	{
		int fc=FirstChildT(i,t);
		if(fc != -1)
		{
			if(fc == n)
			{
				return i;
			}
			else
			{
			while(NextSiblingT(fc,t) != -1)
			{
				fc=NextSiblingT(fc,t);
				if(fc == n)
				{
					return i;
				}

			}
			}
		}
	}
}

elem LabelT (int n, stablo &t)
{
	return t.element[n];
}

int RootT (stablo &t)
{
	if(!init_t)
	{
		return -1;
	}
	else
	{
		return t.first;
	}

}
```

```
void ChangeLabel T(elem x, int n, stablo &t)
{
	if(t.element[n].firstchild == -2) return;
	t.element[n] = x;
}

int CreateT (elem x, int n, stablo &t)
{
	int k=-1;
	int i=0;
	if(t.element[n].firstchild == -2 && prvi) return -2;
	if(t.element[RootT(t)].firstchild!=-2)
	{
		while(i<10000 && k == -1)
		{
			if(t.element[i].firstchild==-2) k = i;
			i++;
		}
		if(k == -1) return -1;
		if(FirstChildT(n,t)!=-1)
		{
			int fc=FirstChildT(n,t);
			while(NextSiblingT(fc,t)!=-1) fc=NextSiblingT(fc,t);
			t.element[fc].nextsib=k;
		}
		else t.element[n].firstchild = k;
	}
	else
	{
		k = n;
		prvi = 1;
	}
	t.element[k] = x;
	return k;
}


void DeleteT (int n, stablo &t)
{
	while(FirstChildT(n,t) != -1) DeleteT(FirstChildT(n,t),t);
	if(n != RootT(t))
	{
		int p=ParentT(n,t);
		int fcp=FirstChildT(p,t);
		if(fcp==n&&NextSiblingT(fcp,t) == -1) t.element[p].firstchild =
-1;
		else if(fcp == n && NextSiblingT(fcp,t) != -1)
t.element[p].firstchild=NextSiblingT(n,t);
		else
		{
			while(NextSiblingT(fcp,t) != n) fcp=NextSiblingT(fcp,t);
			t.element[fcp].nextsib = NextSiblingT(n,t);
		}
	}
	cout << „Obrisan je cvor „ << t.element[n].label << endl;
	t.element[n].firstchild=-2;
}
```

Implementacija 4. Stablo (Manger, 2013)

Kod implementacija stabla „prvo dijete – sljedeći brat“ sve operacije osim ParentT i DeleteT izvršavaju se u konstantom vremenu, što je vidljivo u tablici 7.

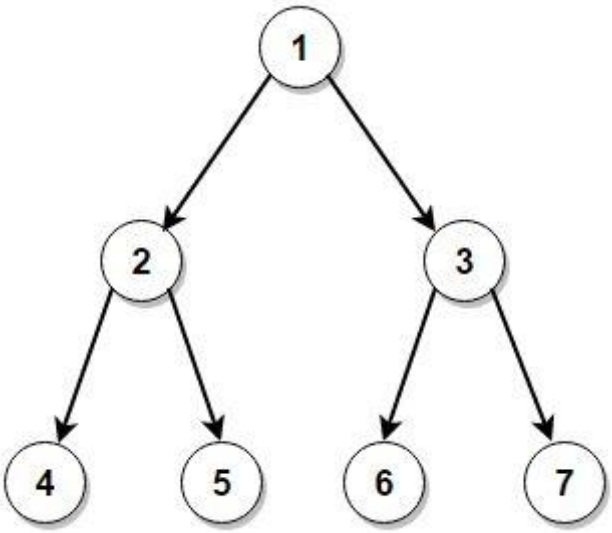
Funkcija	Složenost
ParentT	O(n)
FirstChildT	O(1)
NextSiblingT	O(1)
LabelT	O(1)
RootT	O(1)
CreateT	O(1)
ChangeLabelT	O(1)
DeleteT	O(n)
InitT	O(1)

Tablica 7.Složenost stabla (Manger, 2013, str. 54)

7. Apstraktni tip podataka binarno stablo

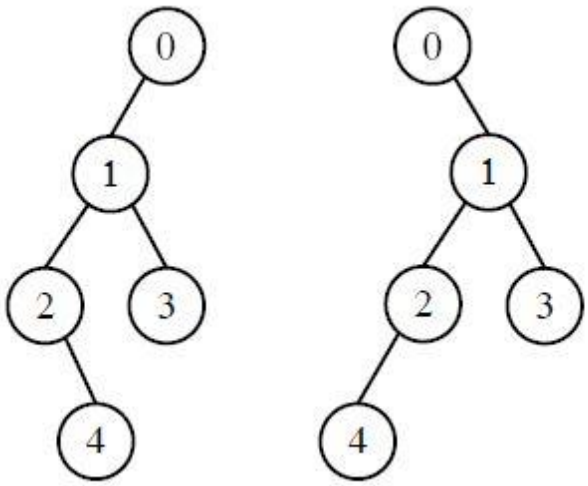
Binarno stablo je jedna od najvažnijih struktura u računalnoj znanosti. Postoji mnogo načina da se definira binarno stablo. Matematički gledano, binarno stablo je povezani, neusmjereni, konačni graf bez ciklusa. Na slici 9 vidljivo je da svaki čvor ima najviše dva djeteta, što znači da može imat 0,1 ili 2 djeteta. Manja binarna stabla nazivaju se lijevo i desno podstablo, a roditelj ima lijevo i desno dijete. Čvorovi mogu imati samo jedno dijete, ali nije svejedno da li je ono lijevo ili desno, dok svako od podstabala može imati svoja manja podstabla (Morin, 2014).

Ako čvor ubacujemo kao list, njegovom adresom treba popuniti odgovarajući pokazivač u čvoru roditelja. Kad se novi čvor ubacuje između roditelja i njegovog djeteta, novi čvor preuzima na toj strani od roditelja odgovarajući pokazivač na dijete, dok roditelj pokazuje na novi čvor. Za obilazak binarnog stabla koriste se tri uobičajene metode kao i kod stabla, a to su preorder, inorder i postorder (Tomašević, 2004).



Slika 9. Primjer binarnog stabla (Techiedelight, 2017)

Pojam binarnog stabla nije isti kao kod običnog stabla koje je opisano u prethodnom poglavlju, jer svako dijete mora biti desno ili lijevo. Na slici 10 vidimo dva različita binarna stabla, primijetimo da je kod lijevog stabla čvor 1 lijevo dijete od čvora 0 i čvor 0 nema desno dijete, a kod desnog stabla je čvor 1 je desno dijete čvora 0 te čvor 0 nema lijevo dijete.



Slika 10. Primjer dva različita binarna stabla (Manger, 2013, str. 59)

Funkcije ATP binarno stablo (binary tree):

- ParentB(n,B) – funkcija vraća roditelja čvora n
- LeftChildB(n,B) – funkcija vraća lijevo dijete čvora n
- RightChildB(n,B) – funkcija vraća desno dijete čvora n
- LabelB(n,B) – funkcija vraća vrijednost koju sadrži čvor n
- ChangeLabelB(x,n,B) – mijenja oznaku čvora n na vrijednost x
- RootB(B) – funkcija koja vraća korijen
- CreateLeftB(x,n,B) – dodaje x kao lijevo dijete čvora n
- CreateRightB(x,n,B) – dodaje x kao desno dijete čvora n
- DeleteB(n,B) – briše čvor n i sve njegove potomke
- InitB(x,B) – inicijalizira novo binarno stablo

7.1. Implementacija binarnog stabla

Implementacija binarnog stabla pomoću polja je posebna vrsta stabla koja se naziva potpuno binarno stablo. Stoga se zbog „pravilnog oblika“ mogu lakše prikazati na računalu. Lijevo dijete čvora *i* je čvor *2i+1* i ako *2i+1 > n-1* čvor *i* nema lijevo dijete, dok je desno dijete čvora *i 2i+2* te ako je *2i+2 > n-1* tada čvor *i* nema desno dijete. Korijen je predstavljen *0-tom* klijetkom, a lijevo i desno dijete nalaze se u *(2i+1)-oj i (2i+2)-oj* klijetki. Roditelj čvora iz *i-te* klijetke nalazi se u *[(i-1)/2]*-toj klijetki (Manger, 2013).

Kod implementacije binarnog stabla pomoću pokazivača zapišemo sve čvorove te svakom čvoru njegovo lijevo i desno dijete. Veze zabilježimo pomoću kursora ili pokazivača. Čvorove prikazujemo klijetkama te im pridružujemo oznake, ako čvor nema desno ili lijevo dijete, pridružujemo mu pokazivač *NULL.*

Funkcija CreateRightB() fizički stvara novi čvor kojemu se dodjeljuje vrijednost n. Funkcije LeftChildB() i RightChildB() su jednostavne, a vraćaju lijevo odnosno desno dijete. ParentB() pomoću selekcije *if* vraća roditelja čvora n. InitB() dinamičkom alokacijom stvara novi čvor te postavlja njegovu djecu na *NULL.* DeleteB() također pomoću selekcije *if* prvo provjerava te briše lijevo ili desno dijete.

```
#define LAMBDA NULL
typedef  int labeltype;

struct cell
{
      labeltype label;
      cell *left, *right;
};

typedef cell* BinaryTree;
typedef cell* node;

node InitB (labeltype x, BinaryTree *B)
{
      cell *novi = new cell;
      novi->label = x;
      novi->left = LAMBDA;
      novi->right = LAMBDA;
      *B = novi;
      return *B;
}

node RootB (BinaryTree B)
{
      return B;
}
```

```
node CreateLeftB (labeltype l, node n, BinaryTree *B)
{
	if (n->left != LAMBDA)
	{
		cout<<"Greska! Dijete vec postoji."<<endl;
	}
	else
	{
		cell *novi = new cell;
		novi->label = l;
		novi->left = NULL;
		novi->right = NULL;
		n->left = novi;
		return novi;
	}
}

labeltype LabelB (node n, BinaryTree B)
{
	if(n != NULL) return n->label;
}



node CreateRightB (labeltype l, node n, BinaryTree *B)
{
	if (n->right != LAMBDA)
	{
		cout<<"Greska! Dijete vec postoji."<<endl;
	}
	else
	{
		cell *novi = new cell;
		novi->label = l;
		novi->left = NULL;
		novi->right = NULL;
		n->right = novi;
		return novi;
	}
}

void ChangeLabelB (labeltype l, node n, BinaryTree *B)
{
	if(n != NULL) n->label = l;
}

node LeftChildB (node n, BinaryTree B)
{
	return n->left;
}

node RightChildB (node n, BinaryTree B)
{
	return n->right;
}
```

```
node ParentB (node n, BinaryTree B)
{
      if (n == B || B == NULL) return NULL;
      if (B->left != NULL)
            if(B->left == n) return B;
      if (B->right != NULL)
            if (B->right == n) return B;
      node t = ParentB(n,B->left);
      if (t) return t;
      t=ParentB(n,B->right);
      if (t) return t;
}

void DeleteB (node n, BinaryTree *B)
{
      node t = ParentB(n, *B);
      if (t != NULL) {
            if (t->left == n)
                  t->left = NULL;
            if (t->right == n)
                  t->right = NULL;
      }
      if (n->left != NULL) DeleteB(n->left, B);
      if (n->right != NULL) DeleteB(n->right, B);
      delete n;
}
```

Implementacija 5. Binarno stablo (Manger, 2013)

U tablici 8 vidimo funkcije koje imaju konstantnu vremensku složenost svih operacija osim ParentB() i DeleteB().

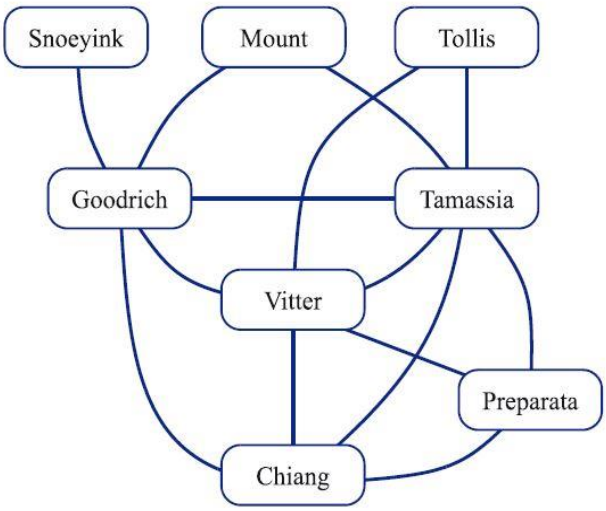
Funkcija	Složenost
ParentB	O(n)
LeftChildB	O(1)
RightChildB	O(1)
LabelB	O(1)
ChangeLabelB	O(1)
RootB	O(1)
CreateLeftB	O(1)
CreateRightB	O(1)
DeleteB	O(n)
InitB	O(1)

Tablica 8. Složenost binarnog stabla (Manger, 2013, str. 62)

8. Apstraktni tip podataka graf

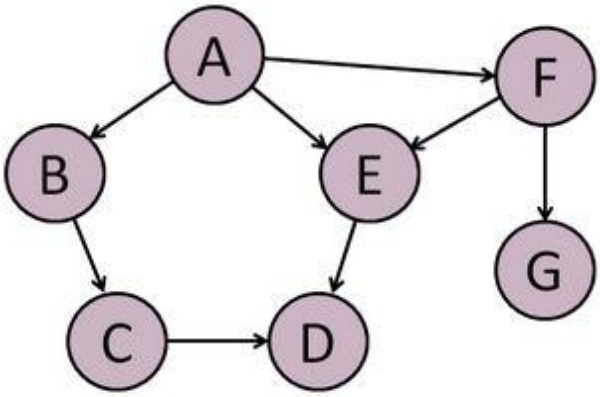
Graf se sastoji od skupa čvorova i skupa grana koje povezuju čvorove. Grafovi se predstavljaju kao dijagrami gdje su čvorovi kružići, a grane linije (ili strelice) koje spajaju pojedine kružiće. Sastoji se od skupa čvorova *V* i skupa grana *E,* pri čemu svaka grana iz *E* spaja točno dva čvora iz *V*. Graf se označava zapisom *G = (V, E)* gdje je *V* skup čvorova, a *E* skup grana u grafu *G.* Stupanj čvora je broj grana nekog čvora koje ga spajaju sa drugim čvorovima tj. stupanj čvora je broj njegovih susjeda (Živković, 2010).

Slika 11 predstavlja se uređeni par čvorova (vrhova) i grana (bridova) gdje svaki čvor predstavlja jedan grad, a grane predstavljaju ceste koje povezuju te gradove. Grafovi mogu biti usmjereni i neusmjereni, a primjenjuju se u mnogo različitih slučajeva, npr. računalne mreže mogu se modelirati kao grafovi, tako da vrhovi predstavljaju računala, a bridovi veze među njima. Također gradske ulice se mogu modelirati pomoću grafova tako da vrhovi predstavljaju križanja, a bridovi ulice koje se pridružuju križanjima (Morin, 2014).



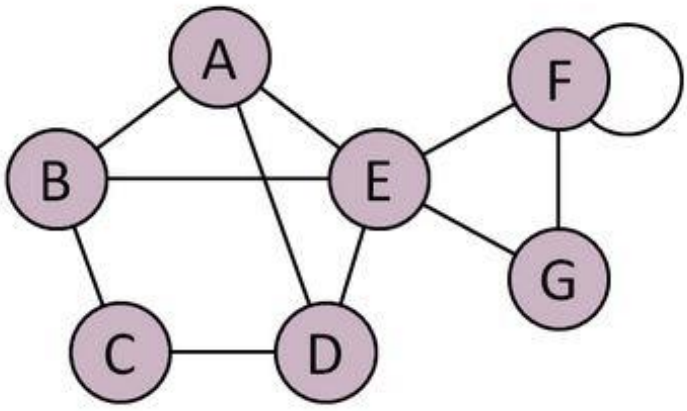
Slika 11. Primjer grafa (Morin, 2014, str. 140)

Usmjereni graf je uređeni par *(V, E),* pri čemu je *V* skup vrhova grafa koji je konačan, ima orijentirane bridove koje prikazujemo strelicama, a svaki brid ima smjer od početka prema kraju (Divjak, 2008).



Slika 12. Primjer usmjerenog grafa (Boljiprogramer, 2017)

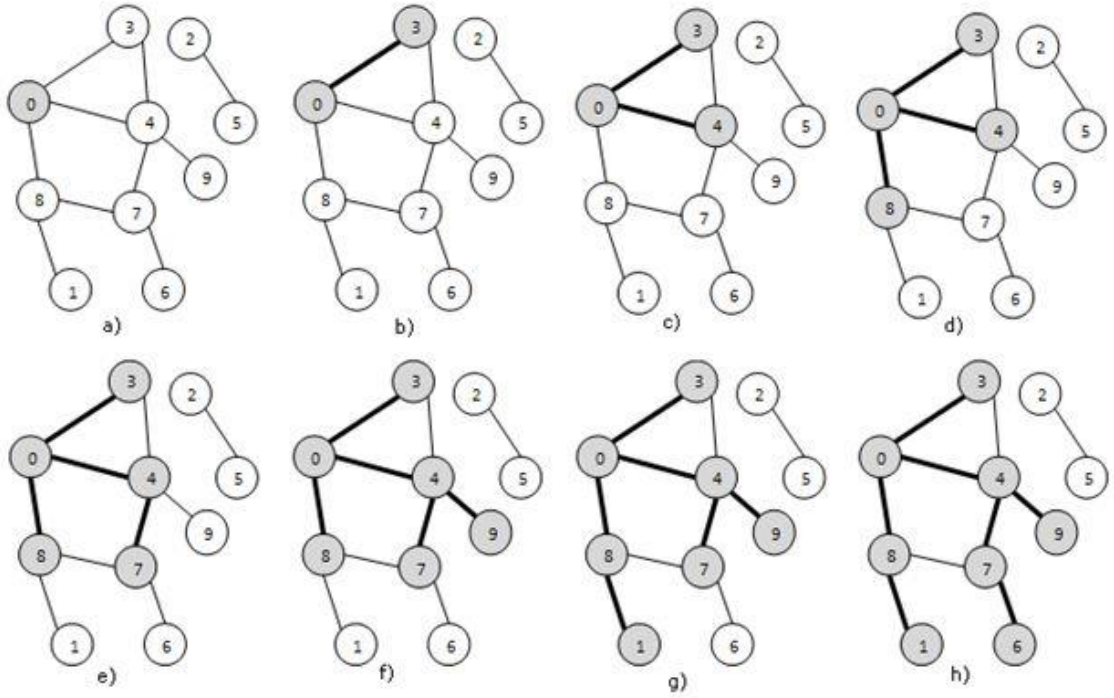
Neusmjereni graf je uređeni par *(V, E)*, pri čemu je V skup vrhova grafa koji je konačan i skup parova elemenata iz *V*, koji čini skup bridova grafa (Divjak, 2008).



Slika 13. Primjer neusmjerenog grafa (Boljiprogramer, 2017)

Pretraživanje u širinu najjednostavniji je algoritam za određivanje povezanosti u grafu. Za dani graf *G=(V,E)* i njegov čvor s, kreće se od čvora *s* i u svakoj fazi otkrivaju se novi čvorovi. U prvoj fazi prolazimo kroz čvorove koji su spojeni granom s čvorom *s.* Druga faza se sastoji od svih novih čvorova koji su spojeni granom s nekim čvorom iz prve faze. Treća faza se sastoji od svih novih čvorova koji su spojeni granom s nekim čvorom iz druge faze i tako se taj postupak ponavlja dok se ne može otkriti nijedan novi čvor (Živković, 2010).

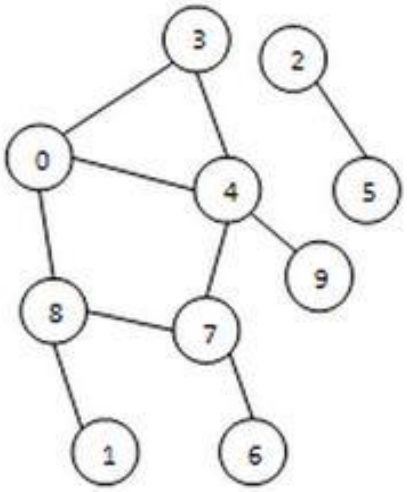
Na slici 14 vidljivo je da početak pretrage kreće od čvora 0, prvo se obilaze svi njegovi direktni susjedi redom 3,4,8. Zatim susjede čvora 3, kojih nema pa slijedi čvor 4 gdje obilazimo čvorove 7 i 9. Susjed čvora 8 je čvor 1. Na kraju neposjećene susjede 7, 9 i 1, a to je sam čvor 6 kao susjed čvora 7. Konačan redoslijed glasi: 0, 3, 4, 8, 7, 9, 1, 6 .



Slika 14. Primjer pretraživanja u širinu (Đurišić, 2014)

Pretraživanje u dubinu potiče od toga što se ide sve dublje u graf dokle god se put ne može nastaviti bilo to jer posljednji čvor nema grana koje vode od njega ili zato što sve grane od njega vode do već posjećenih čvorova. Čvorove i grane zamislimo kao labirint međusobno povezanih prolaza kroz koje treba pronaći izlaz od početnog čvora *s,* prvo se proba prva grana koja vodi do *s.* Ako vodi do čvora *v* onda se i dalje slijedi prva grana koja vodi do *v.* Zatim se postupak ponavlja do čvora u koji se dođe od čvora *v,* sve dok se može, odnosno do čvora čiji su susjedi već posjećeni. Nakon toga se vraća putem kojim se došlo do prvog čvora čiji susjedi još nisu posjećeni te zatim se otud ponavlja isti postupak. Kod svakog čvora se ponavlja ista aktivnost, stoga se pretraga u dubinu može najlakše opisati u rekurzivnom obliku (Živković, 2010).

Na slici 15 vidljivo je pretraživanje u dubinu gdje se pretražuje u ciklusima, kreće se od čvora 0 do čvora 3, od čvora 3 do čvora 4, zatim od čvora 4 ponovo do čvora 0 i tako se ponavlja u krug. Bitno je da jedan čvor prođemo točno jednom, zato svaki put kada posjetimo novi čvor obilježimo ga



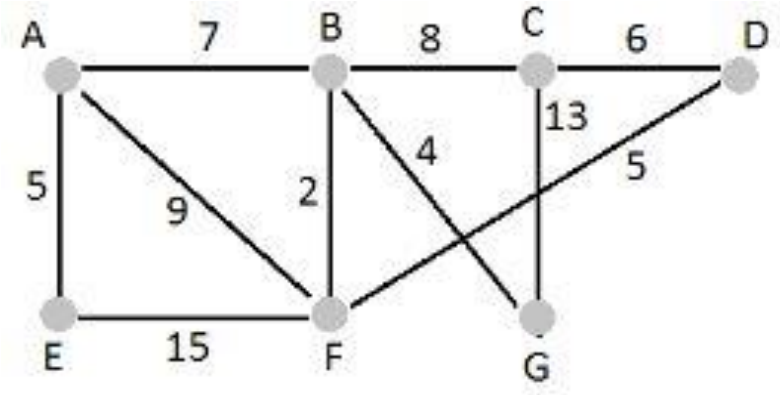
Slika 15. Pretraživanje u dubinu (Đurišić, 2014)

Eulerov ciklus koristi se kod problema kineskog poštara. Kada poštar u poštanskom uredu pokupi poštu odlazi je razdijeliti i vraća se u ured. Mora odabrati rutu gdje će što manje hodati tj. želi proći svakom ulicom samo jednom. Cilj je da se nađe Eulerova tura najmanje težine tzv. optimalna Eulerova tura. Ako je G Eulerov graf, onda je svaka Eulerova tura na G optimalna, jer se njome svaki brid prijeđe samo jedan put. U navedenom slučaju problem se rješava Fleuryjevim algoritmom koji konstituira Eulerovu turu tako da počne u nekom vrhu i u svakom koraku bira vezni brid neprijeđenog podgrafa (Veljan, 2003).

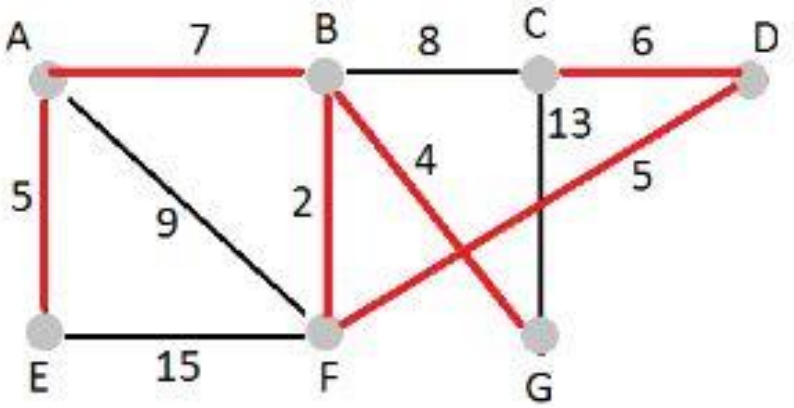
Rješavanje problema trgovačkog putnika pruža veliki izazov zbog njegove težine, a i zbog povezanosti s praktičnim i teorijskim pitanjima. U navodnom problemu cilj je obići neke gradove i vratiti se na mjesto polaska tako da svakim gradom i cestom prođemo samo jednom. Hamiltonov put na grafu *G* je put koji sadrži sve vrhove od *G*. Hamiltonov ciklus na grafu *G* je ciklus koji sadrži sve vrhove od G. Dakle, to je razapinjući ciklus grafa i prolazi kroz sve njegove vrhove točno jednom. Graf je Hamiltonov ako ima Hamiltonov ciklus (Jurašić, 2011).

Kod Kruskalovog algortima se polazi od šume *(V,Ø)* koja se sastoji od svih čvorova i nema nijednu granu. Točnije, vidljiva je samo šuma bez pojedinačnih čvorova. Redom se ispituju grane prema rastućem redoslijedu njihovih težina. Kako se dolazi do neke grane *e,* provjera se da li *e* proizvodi ciklus kada se doda granama koje su do tada izabrane za minimalno razapinjuće stablo. Stoga ako grana *e* povezuje dva čvora iz dva različita stabla dodaje se skupu izabranih grana, ako *e* povezuje dva čvora iz istog stabla *e* se odbacuje kako nebi došlo do ciklusa (Živković, 2010).
Slike 16 i 17 prikazuju Kruskalov algoritam za pronalaženje minimalnog razapinjućeg stabla koji se može opisati kroz nekoliko koraka:

- Početi od potpuno nepovezanog grafa T
- Poredati sve grane tako da im težina bude u rastućem redoslijedu
- Početi od najjeftinije grane i dodavati ih u graf te paziti da se ne stvori ciklus
- Ponoviti korak 3 dok graf ne bude imao n-1 granu

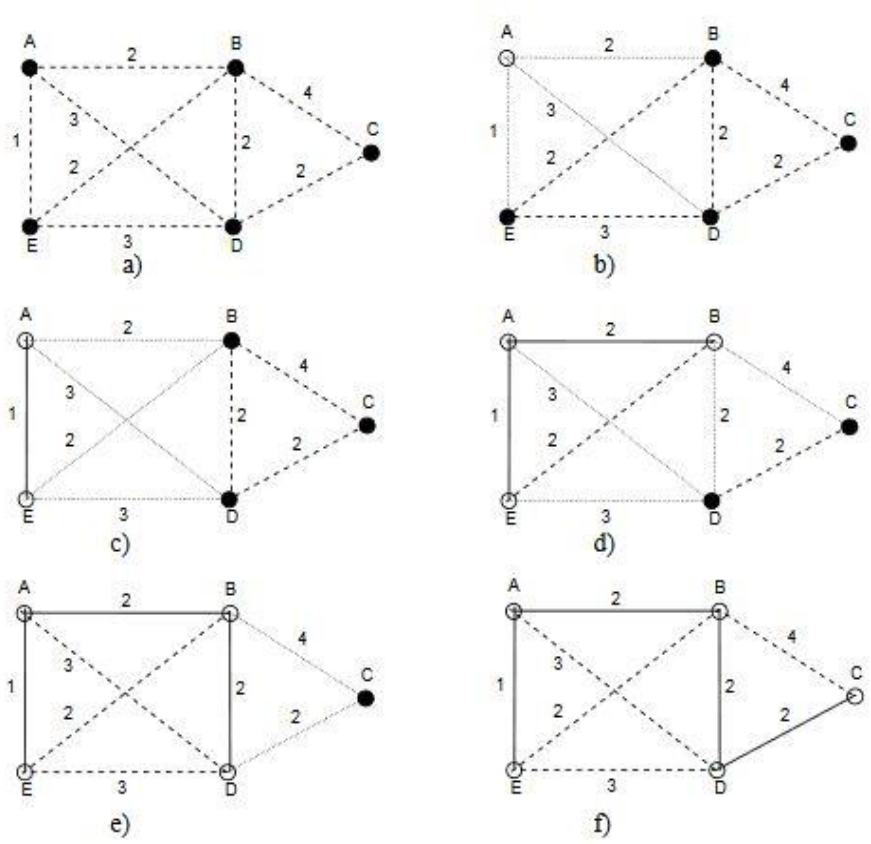


Slika 16. Kruskalov algoritam (problem) (Algoritmi na grafovima, 2013)



Slika 17. Kruskalov algoritam (rješenje) (Algoritmi na grafovima, 2013)

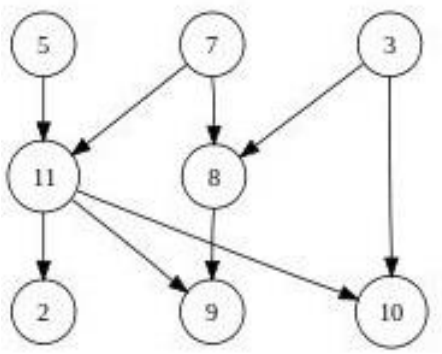
U Primovom algoritamu nasuprot Kruskalovog kod kojeg se „pohlepno“ biraju grane gdje se pazi da se ne formira ciklus se dobiva minimalno razapinjajuće stablo tako da se početno stablo polako proširuje. Početno stablo dobijemo tako da odaberemo neki čvor *s* i u svakom koraku djelomično dodajemo novi čvor na drugom kraju grane, a da ima najmanju težinu. Na slici 18 vidljivo je da u prvom koraku uzimamo proizvoljan vrh iz skupa, postupak nastavljamo dodavanjem brida u stablo koje ima svojstvo da povezuje jedan vrh koji se već nalazi u stablu i jedan koji se u njemu ne nalazi, pazeći da težina brida bude minimalna (Živković, 2010).



Slika 18. Primovom algoritam (Marinović, 2004)

Topološko sortiranje usmjerenog grafa je linearni poredak njegovih čvorova tako da svaka grana *uv* od čvora *u* do čvora *v* dolazi prije *v* u sortiranju. Primjerice, čvorovi grafa mogu biti zadaci koji se trebaju izvršiti, a grane predstavljaju ograničenja da jedan zadatak mora biti izvršen prije prelaska na drugi. Da bi topološko sortiranje bilo moguće u grafu ne smije biti ciklusa. Zbog redoslijeda u kojem se obavljaju poslovi topološko sortiranje proučavano je i primijenjeno kod tehnike PERT (tehnika za raspoređivanje i upravljanje projektima).

Slika 19 prikazuje niz mogućih topoloških sortiranja, primjerice niz 5, 7, 3, 11, 8, 2, 9, 10 dobije se sortiranjem od lijeva prema desno pa odozgo prema dolje. Ako krenemo od čvora s najmanjom vrijednošću dobijemo niz 3, 5, 7, 8, 11, 2, 9, 10.



Slika 19. Topološko sortiranje (Geeksforgeeks, 2014)

Funkcije ATP graf (graph):

- Create() – kreiranje grafa
- InsertVertix(graph) – ubacivanje novog retka i stupca u matricu
- DeleteVertix(graph,v) – izbacivanje v retka i stupca iz matrice
- 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.
- 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
- IsEmpty(graph) – provjereva da li je graf prazan
- Adjacet(graph,v1,v2) – vraća težinsku vrijednost između vrhova

8.1. Implementacija grafa

Implementacija grafa izvedena je pomoću polja, točnije pomoću matrice susjedstva.

```
# define MAXSIZE ...

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

graph Create (graph *G)
{
	graph G = new graph;
	G->last=0;
	return G;
}

graph InsertVertix (graph *G)
{
	return G->last++;
}

int Adjacent (graph *G, v1, v2)
{
	if(v1<G->last && v2<G->last)
	{
		return G[v1][v2];
	}
	else
	{
		cout << "Ne postoje ti vrhovi" << endl;
	}
}
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
	{
		cout << "Nema mjesta za novi vrh" << endl;
	}
}

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
	{
		cout << "Ne postoje ti vrhovi" << endl;
	}
}

graph DeleteEdge (graph *G, int v1, int v2)
{
	if(v1<G->last && v2<G->last)
	{
		return G[v1-1][v2-1] = 0;
	}
	else
	{
		cout << "Ne postoje ti vrhovi" << endl;
	}
}

bool IsEmpty (graph *G)
{
	if(G->last==0)
	{
		return 1;
	}
	else
	{
		return 0;
	}
}
```

Implementacija 6. Graf (Višić, 2011)

Tablica 9 prikazuje složenost pojedine funkcije apstraktnog tipa podataka gdje se sve operacije osim DeleteVertix izvode u konstantnom vremenu.

Funkcija	Složenost
InsertVertix	O(1)
DeleteVertix	$O(n^2)$
InsertEdge	O(1)
DeleteEdge	O(1)
Adjacent	O(1)

Tablica 9. Složenost grafa (Višić, 2011)

9. Apstraktni tip podataka skup

Skup je sagrađen od objekata koji su svi neposredno istaknuti ili zadovoljavaju neki zahtjev, te objekte zovemo elementima ili članovima skupa. Neki od primjera skupa su: skup učenika u učionici, skup brojeva 2, 4, 6, 8 itd. Skup je zadan ako za svaki element možemo odrediti pripada li on skupu ili ne. Skup koji nema niti jedan element ili člana zovemo prazan skup, a označava se simbolom Ø (Skupovi brojeva, 2010).
Svi elementi moraju biti istog tipa te im vrijednost mora biti različita. Definira se kao apstraktni tip podatka s operacijama uobičajenim u matematici (Skupovi, 2007).

Funkcije ATP skup (set):

- SET – konačni skup čiji su elementi tipa elementtype
- MAKE_NULL(&A) – funkcija pretvara skup u prazan skup
- INSERT(x,&A) – ubacivanje elementa x
- DELETE(x,&A) – brisanje elementa x
- MEMBER(x,A) – funkcija vraća true ako je x element od A, inače false
- MIN(A), MAX(A) – vraća najmanji (najveći) element skupa
- SUBSET(A,B) – funkcija vraća true ako je A podskup od B
- 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

9.1. Implementacija skupa

Kod implementacije skupa pomoću bit-vektora skup prikazujemo poljem bitova. Bit s indeksom *i* je 1 ako *i-ti* element pripada skupu. Navedena implementacija ima jednu veliku manu, a to je da joj treba puno prostora. Na primjer ako se koriste 32-bitni cijeli brojevi tada će biti potrebno potrošiti 4 milijarde bitova. Vrijeme izvršavanja je veoma dugo i implementacija je neupotrebljiva za velik broj ulaznih podataka (Manger, 2013).

U ovom dijelu implementacije skupa prikazat ćemo operacije UNION(), INTERSECTION(), DIFFERENCE() i SUBSET() jer ostale se teško implementiraju da učinkovito obavljane jedne operacije ne uspori ostale operacije. Implementacija skupa

biti će prikazana pomoću sortirane vezane liste. Lista mora biti sortirana kako bi se operacije efikasnije obavljale.

Funkcija INTERSECTION() prvo stvara trenutne ćelije u listi za A,B i C zatim se uspoređuju elementi liste A i B te dodaje u presjek C. Na kraju provjerava da li su različiti (da li je manji onaj iz A ili B).

Funkcija UNION() uspoređuje elemente iz liste A i B, a da već nije u C, zatim dodaje novi iz liste A, pa dodaje novi iz B ako se već ne nalazi u C. Te na kraju dodaje samo iz A ako su A i B isti.

Funkcije DIFFERENCE() i SUBSET() su slične kao i prve dvije navedene operacije gdje DIFFERENCE() provjerava da li u presjeku skupova A i B je skup svih elemenata čiji su članovi i skupa A i skupa B. SUBSET() provjerava da li je svaki član skupa A također i član skupa B.

```
void INTERSECTION (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))
	{
		if ((ac->element) == (bc->element))
		{
			cc->next = new settype;
			cc = cc->next;
			cc->element = ac->element;
			ac = ac->next;
			bc = bc->next;
		}
		else if ((ac->element)<(bc->element))
		{
			ac = ac->next;
		}
		else
		{
			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))
	{
		if( cc->element!=ac->element && cc->element!=bc->element)
		{
			if(ac->element != bc->element)
			{
				cc->next = new settype;
				cc = cc->next;
				cc->element = ac->element;
				ac = ac->next;
				if(cc->element!=bc->element)
				{
					cc->next = new settype;
					cc = cc->next;
					cc->element = ab->element;
					bc = bc->next;
				}
			}
			else
			{
				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;
            }
        }
}
```

Implementacija 7. Skup (Manger, 2013)

Tablica 10 predstavlja složenost operacija Intersection, Union, Difference i Subset gdje sve četiri operacije imaju linearnu složenost. Vrijeme izvršavanja im je proporcionalno duljini jedne od listi.

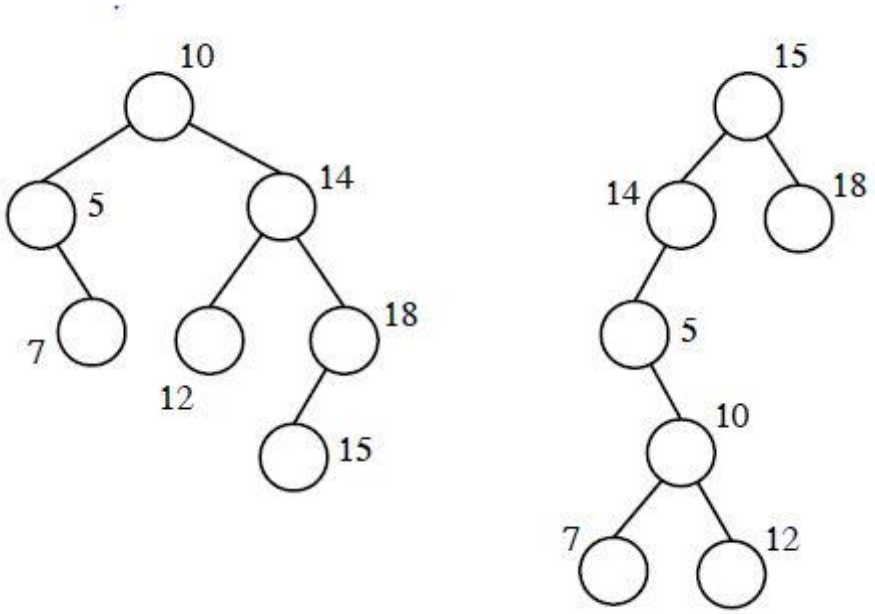
Funkcija	Složenost
Intersection	O(n)
Union	O(n)
Difference	O(n)
Subset	O(n)

Tablica 10. Složenost skupa (Manger, 2013, str. 72)

10. Apstraktni tip podataka rječnik

Rječnik je posebna vrsta skupa kod kojeg nisu predviđene složene operacije kao što su presjek, unija ili razlika. Izvršavaju se samo povremena izbacivanja i ubacivanja elemenata te utvrđuje se da li je neki element u skupu ili ne. Podaci moraju biti istog tipa i ne mogu postojati dva podataka s istom vrijednošću. Slika 14 prikazuje skup A = {5, 7, 10, 12, 14, 15, 18} koji je predstavljen pomoću binarnog stabla traženja. Najjednostavniji primjer rječnika je pravopis, tj. popis ispravno napisanih riječi gdje se provjerava točnost napisane riječi u nekom programu (Manger, 2013).

A = {5, 7, 10, 12, 14, 15, 18}



Slika 20. Primjer rječnika pomoću binarnog stabla traženja (Manger, 2013, str. 83)

Funkcije ATP rječnik (dictionary):

- MAKE_NULL (A) – pretvara skup u prazan skup
- INSERT(x, A) – ubacuje element x
- DELETE(x,A) – izacuje element x
- MEMBRE(x,A) – provjerava da li je x element od A

10.1. Implementacija rječnika

Implementacija rječnika pomoću bit-vektora je identična kao i kod općenitog skupa. Rječnik se prikazuje pomoću polja bitova gdje svaki bit određuje da li određena vrijednost postoji u rječniku ili ne. Isto kao i kod općenitog skupa vremenska složenost funkcija je O(1), ali implementacija postaje neupotrebljiva kada je riječ o velikom broju ulaznih podataka (Manger, 2013).

Sljedeća implementacija rječnika izvedena je pomoću liste. Listu pristavljamo pomoću polja, a pogodna je za statične rječnike. Većina operacije se odvija u linearnom vremenu jer se radi o polju gdje se pokušava zadržati sortiranost polja pa se moraju podaci prepisivati.

Kod implementacije pomoću hash tablice element *x* spremimo u pretinac s indeksom *h(x)* te ga kasnije u istom pretincu tražimo. Imamo dvije vrste „haširanja“, a to su otvoreno i zatvoreno. Kod otvorenog pretinci su povezani kao vezana lista. Pri ubacivanju elementa lista se produljuje još jednim zapisom. Pokazivači koji predstavljaju početak liste organiziraju se u polje, a nazivaju se zaglavlje. Tako kapacitet jednog pretinca je promjenjiv i neograničen. Zatvoreno „haširanje“ ima pretince s fiksnim kapacitetom. U svaki pretinac stane jedan element, ako trebamo ubaciti element, a klijetka je već puna gledamo *hh(1,x), hh(2,x)* itd., dok ne nađemo prvo slobodno mjesto, zato je još poznato kao linearno „haširanje“ (Manger, 2013).

Implementacija rječnika pomoću binarnog stabla traženja prikazuje se kao stablo i svakom elementu pripada točno jedan čvor stabla i obratno. Svaki element je spremljen u određenom čvoru, a služi kao oznaka odgovarajućeg čvora stabla, tako će svi čvorovi imati različite oznake.

Funkcija Make_null() je trivijalna i svodi se samo na pridruživanje pojedine vrijednosti pokazivaču *NULL.* Memebr() gleda korijen binarnog stabla, ako je binarno stablo prazno, onda *x* nije u A pa funkcija vraća 0. Ako je *x* spremljen u korijenu stabla, vratit će nam 1. Operacija Insert() traži mjesto u koje će se staviti novi element tako da provjeri je li postoji korijen stabla, ako ne postoji stvara se nova klijetka i pretvara u korijen. Pri izbacivanju funkcija Delete() gleda korijen odgovarajućeg binarnog stabla, tj. gleda se je li *x* manji ili veći od elementa u korijenu pa se rekurzivnim pozivom iste funkcije izbacuje iz lijevog ili desnog podstabla.

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

typedef celltype *DICTIONARY;


void Make_null(DICTIONARY *A)
{
      A->leftchild=NULL;
      A->rightchild=NULL;
}

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 return(Member(x,A->rightchild));
}

void Insert(elementtype x, DICTIONARY *Ap)
{
      if (*Ap == NULL)
      {
            *Ap = (celltype*) malloc (sizeof(celltype));
            (*Ap)->element = x;
            (*Ap)->leftchild = (*Ap)->rightchild = NULL;
      }
      else if (x < (*Ap)->element) Insert(x,&((*Ap)->leftchild));
      else if (x > (*Ap)->element) Insert(x,&((*Ap)->rightchild));
}

elementtype DeleteMin (Dictionary *Ap)
{
      celltype *temp;
      elementtype minel;
      if ( (*Ap)->leftchild)==NULL )
      {
            minel = (*Ap)->element;
            temp = (*Ap); (*Ap) = (*Ap)->rightchild; free(temp);
      }
      else
      minel = DeleteMin( &((*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) );
      else if ( ((*Ap)->leftchild==NULL) && ((*Ap)->rightchild==NULL))
      {
            free(*Ap); *Ap = NULL;
      }
      else if ( (*Ap)->leftchild == NULL )
      {
            temp = *Ap; *Ap = (*Ap)->rightchild; free(temp);
      }
      else if ( (*Ap)->rightchild == NULL )
      {
            temp = *Ap; *Ap = (*Ap)->leftchild; free(temp);
      }
      else
      (*Ap)->element = DiDeleteMin ( &((*Ap)->rightchild) );
}
```

Implementacija 8. Rječnik (Manger, 2013)

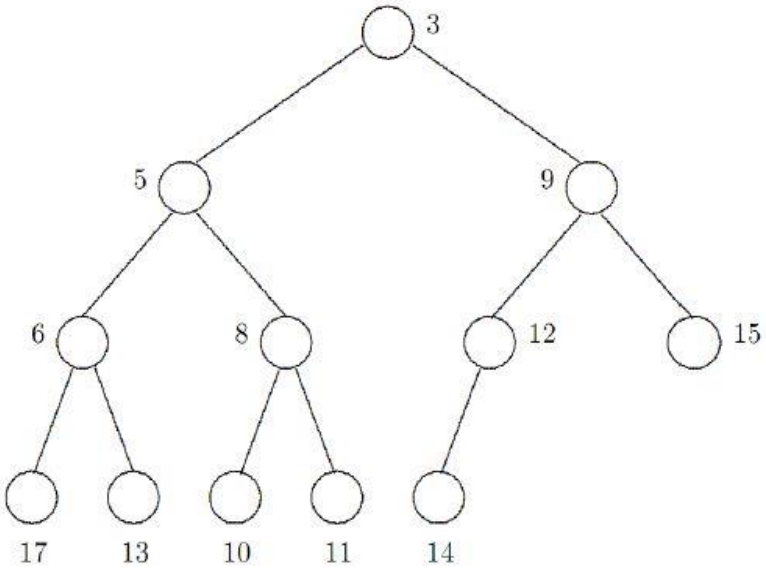
Tablica 11 prikazuje složenost funkcija kod implementacije apstraktnog tipa podataka rječnika pomoću binarnog stabla traženja gdje funkcije Delete, Member i Instert imaju logaritamsku složenost, a funkcija Make_null konstantnu.

Funkcija	Složenost
Make_null	O(1)
Member	O(log n)
Insert	O(log n)
Delete	O(log n)

Tablica 11. Složenost rječnika (Manger, 2013, str. 85)

11. Apstraktni tip podataka prioritetni red

Prioritetni red je jedna od posebnih vrsta skupa, jer se definiraju operacije ubacivanja novog elementa te izbacivanje elementa s najmanjim prioritetom. Red čekanja kod liječnika gdje najteži bolesnik ima prednost nad ostalima je jedan od primjera prioritetnog reda u stvarnom životu. Na slici 15 vidi se prikaz prioritetnog reda A = {3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 17} pomoću hrpe, najmanji element nalazi se u korijenu hrpe. Definiranjem leksikografskog uređaja za uređene parove dobivamo: (prioritet(x1), x1) je manji ili jednak od (prioritet(x2), x2) ako je (prioritet(x1) manji od prioritet(x2)) ili (prioritet(x1) jednak prioritet(x2) i x1 manji-ili-jednak od x2) (Manger, 2013).



Slika 21. Primjer prioritetnog reda pomoću hrpe (Manger, 2013, str. 90)

Funkcije ATP prioritetni red (priority queue):

- MAKE_NULL(&A) – pretvara skup u prazan skup
- EMPTY(A) – provjerava da li je skup prazan
- INSERT(x,&A) – izbacuje element x
- DELETE MIN(&A) – izbacuje najmanji element

11.1. Implementacija prioritetnog reda

Implementacija prioritetnog reda pomoću sortirane vezane liste predstavlja ga kao sortiranu listu. Znači Delete_min() pronalazi i briše element s početka liste i ima konstantnu složenost. Insert() u najgorem slučaju pri dodavanju mora proći cijelu listu i ima linearnu složenost, što znači da ovisi o veličine same liste.

Za implementaciju pomoću binarnog stabla traženja može se uzeti implementacija koja je objašnjena u prethodnom poglavlju pri implementaciji rječnika.

Prioritetni red prikazujemo hrpom, gdje svakom elementu reda odgovara točno jedan čvor hrpe. Funkcijom Insert() stvaramo novi čvor na prvom mjesto zadnje razine hrpe. Time se pokvarilo svojstvo hrpe te je potrebno zamijeniti element u novom čvoru i element u roditelju novog čvora, zatim se zamijeni element u roditelju i element u roditeljevom roditelju itd. dok ne dobijemo opet svojstvo hrpe. Prolazak kroz hrpu obavlja se na isti način kao i kod implementacije binarnog stabla pomoću polja (Manger, 2013).

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

void INSERT (elementtype x, priority_queue *Ap)
{
      int i;
      elementtype temp;
      if ( Ap->last >= MAXSIZE-1 )
      {
            cout << "Prioritetni red je pun" << endl;
      }
      else
      {
            (Ap->last)++;
            Ap->elements[Ap->last] = x;
            i = Ap->last;
            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;
            }
      }
}

elementtype DELETE MIN (priority_queue *Ap)
{
      int i, j;
```

```
        elementtype minel, temp;
        if ( Ap->last < 0 )
        {
                cout << "Prioritetni red je prazan" << endl;
        }
        else
        {
                minel = Ap->elements[0];
                Ap->elements[0] = Ap->elements[Ap->last];
                (Ap->last)--;
                i = 0;
                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 = 2*i+2;
                        }
                        if ( Ap->elements[i] > Ap->elements[j] )
                        {
                                temp = Ap->elements[i];
                                Ap->elements[i] = Ap->elements[j];
                                Ap->elements[j] = temp;
                                i=j;
                        }
                        else
                        {
                                return(minel);
                        }
                }
                return(minel);
        }
}
```

Implementacija 9. Prioritetni red (Manger, 2013)

Tablica 12 prikazuje složenost prioritetnog reda implementiranog pomoću hrpe. Operacije Make_null i Empty su trivijalne te se uvijek izvode u konstantnom vremenu. Insert ovisi o veličini hrpe tj. broju elemenata kroz koje mora preći. Delete min ima logaritamsku složenost jer je minimalni element u lijevom ili desnom podstablu.

Funkcija	Složenost
Make_null	O(1)
Empty	O(1)
Insert	O(n)
Delete min	O(log n)

Tablica 12. Složenost prioritetnog reda (Manger, 2013, str. 93)

12. Komparacija apstraktnih tipova podataka prema implementaciji

Implementacija liste pomoću polja pruža brz pristup određenom elementu preko indeksa, zato se funkcije End() i Previous() izvršavaju u konstantnom vremenu, što nije slučaj kod implementacije s pokazivačima. Ubacivanje i izbacivanje zahtjeva prepisivanje dijela podataka. Također polje se lako može prepuniti jer je unaprijed određen konstantom. Implementacija pomoću pokazivača pruža ubacivanje i izbacivanje elemenata bez prethodnog prepisivanja podataka i ne može se prepuniti jer broj elemenata nije unaprijed određen. Ali zato navedena implementacija troši više memorije po elementu jer svaki element sadrži pokazivač koji pamti poziciju sljedećeg elementa.

Kod implementacije stoga pomoću polja podaci se upisuju u donji dio polja, zato se podaci ne moraju dodatno prepisivati. Ova implementacija ima isti problem kao u prethodnom odlomku kada dolazi do prepunjenja polja. Implementacija pomoću pokazivača nudi rješenje za prepunjenost tako što uvodi dinamičku alokaciju memorije. Kod obje implementacije složenost svih operacije je O(1) jer se obavljaju na vrhu stoga pa nije potrebno prolaziti kroz sve elemente stoga.

Red može biti implementiran pomoću kružnog polja ili pokazivača. Implementacija pomoću kružnog polja izbjegava prepisivanje podataka jednostavnim dodavanjem kursora koji pamti trenutni položaj čela. Kao i u prethodnim implementacijama pomoću polja dolazi do problema kada želimo dodati element, a nema mjesta. Implementacija s pokazivačima također kao u prethodnim implementacijama zaobilazi navedeni problem jer koristi dinamičku strukturu podataka. Sve se operacije izvode u vremenu O(1) što znači da je efikasnost ove dvije implementacije ista.

Jedna od mogućnosti implementacije stabla je da svakom čvoru zapišemo njegovog roditelja. Ovakvu vrstu implementacije poželjno je koristiti kada nema puno dodatnih ubacivanja i izbacivanja čvorova te kada nije potrebna uređenost čvorova. Druga implementacija naziva se „prvo dijete – sljedeći brat“ gdje se zapisuju oznake čvora, prvo dijete određenog čvora i čvor koji je sljedeći brat tog određenog čvora. Ova implementacija pogodna je kada ima puno naknadnih ubacivanja čvorova.

Implementacija binarnog stabla koristi poseban način spremanja čvorova u polje. Lijevo dijete čvora *i* je čvor *2i+1* i ako *2i+1 > n-1* čvor *i* nema lijevo dijete, dok je desno dijete čvora *i 2i+2* te ako je *2i+2 > n-1* tada čvor *i* nema desno dijete. Operacije se brže izvode i koristi manje prostora ako je stablo balansirano. Kod implementacije binarnog stabla pomoću pokazivača zapišemo sve čvorove te svakom čvoru njegovo lijevo i desno dijete. Veze zabilježimo pomoću kursora ili pokazivača. Zato su neke operacije sporije jer se manipulira pokazivačima, a koristi manje prostora ako stablo nije balansirano. Složenost svih operacije osim ParentB() i DeleteB() obavljaju se u konstantnom vremenu dok ParentB() i DeleteB() ovise o veličini stabla.

Pomoću matrice na računalu prikazujemo grafove na dva načina: pomoću matrice incidencije i matrice susjedstva. Nedostatak matrice incidencije je kod grafa koji ima puno bridova jer se u redove stavljaju vrhovi, a u stupce bridovi. Zato je navedena matrice puno veća od matrice susjedstva i to je razlog zašto se matrice susjedstva više koriste.

Kod implementacije skupa pomoću bit-vektora koristi se polje bitova. Ova implementacija troši puno prostora, primjerice jedan cijeli broj ima 4 milijarde bitova. Kada je riječ o velikom broju ulaznih podataka navedena implementacija postaje neupotrebljiva jer bi zauzela veliki broj bitova. Kada je riječ o implementaciji pomoću sortirane vezane liste, ona nudi bolju iskorištenost prostora te je korisnija pri većem broju ulaznih podataka. Operacije imaju linearnu složenost, što znači da im je vrijeme izvršavanja proporcionalno duljini jedne od listi.

Rječnik može biti implementiran pomoću bit-vektora, liste (polja), hash tablice i binarnog stabla traženja. Implementacija bit-vektorima je identična kao u prethodnom odlomku i dijeli iste probleme. Iako operacije imaju složenost O(1) kada dođe do većeg broja ulaznih podataka implementacija postaje beskorisna. Druga implementacija zbog održavanja sortiranosti mora prepisivati podatke. Hash tablice moguće je primijeniti na dva načina: kao otvorene i zatvorene hash tablice. Kod otvorenog pretinci funkcioniraju kao vezana lista, a kod zatvorenog pretinci su fiksni te u svaki pretinac stane jedan element. Implementacija rječnika pomoću binarnog stabla traženja prikazuje se kao stablo i svakom elementu pripada točno jedan čvor stabla i obratno. Kod ove implementacije funkcije Delete(), Member() i Instert() imaju logaritamsku složenost, a funkcija Make_null() konstantnu.

Za implementaciju prioritetnog reda pomoću binarnog stabla traženja može se uzeti implementacija koja je objašnjena u poglavlju implementacije rječnika. Prioritetni red prikazujemo hrpom, gdje svakom elementu reda odgovara točno jedan čvor hrpe. Funkcijom Insert() stvaramo novi čvor na prvom mjestu zadnje razine hrpe. Insert() ovisi o veličini hrpe tj. broju elemenata kroz koje mora preći. Delete_min() ima logaritamsku složenost jer je minimalni element u lijevom ili desnom podstablu.

.

13. Zaključak

Implementacija apstraktnih tipova podataka nudi mnoštvo korisnih mogućnosti koje uvelike pomažu u programiranju. Iz analize se zaključuje da je neophodno analizirati pojedine apstraktne tipove podataka kako bi se odabrao najpogodniji koji odgovara pojedinom problemu. Zato neke implementacije su pogodne samo kada je broj ulaznih podataka relativno malen.

Cilj ovog rada je savladati tehnike implementacije pojedinog apstraktnog tipa podataka u programskom jeziku C++, kao i principe rada pojedine operacije te ukazati na prednosti i nedostatke različitih vrsta implementacija i pojedinih operacija.

Apstraktni tip podataka je neizbježan pojam kod programiranja jer pruža jednostavno i korisno prikazivanje podataka neovisno o implementaciji. Budući da postoji mnogo različitih načina da se implementira jedan apstraktni tip podataka, baš ta neovisnost pruža programeru da mijenja pojedine detalje u implementacije bez da se promjeni način korištenja te se korisnik može usredotočiti na proces rješavanja problema.

Kod implementacije operacije glavna dva faktora su vrijeme izvođenja i zauzeće memorije. Za mjerenje vremena nam služi analiza složenosti gdje je svrha da se pri implementiraju funkcije optimizira i razvije što učinkovitija funkcija. Testiranje se obavlja na osnovu ponašanja u najboljem i najgorem slučaju, gdje apriori analiza procjenjuje vrijeme izvođenja pojedine operacije.

Lista koristi linearnu strukturu podatka gdje se elementi grupiraju u prvi slobodni dio memorije, a kada više nije potreban zauzeti dio memorije postaje slobodan. Svaki element ima podatkovni dio i pokazivač na sljedeći element, a glavni dio svake liste je početak koji se naziva glava i služi kako bi se sačuvao početak liste.

Stogu se može pristupiti samo preko vrha i on ukazuje na element koji je posljednji dodan, sve operacije se izvršavaju na vrhu, a pristup ostalim elementima je moguć samo brisanjem ostalih. Operacije se izvode u konstantnom vremenu jer se obavljaju na vrhu pa nije potrebno prolaziti kroz sve elemente. Jedan od primjena stoga u računalnom sustavu je program koji koristi više procedura.

Apstraktni tip podataka red je jedna vrsta liste gdje je moguće dodavanje elemenata samo na kraju, a izbacivanje na početku. Kada je red implementiran pomoću polja, broj

elemenata je ograničen, dok kod implementacije pomoću vezane liste može primiti neograničen broj elementa. Primjer reda u računalnoj znanosti možemo vidjeti kada se podaci pohranjuju u red i kasnije obrađuju.

Hijerarhijsku strukturu koriste stablo i binarno stablo, a sastoje se od elemenata (čvorova) i linija (grana). Čvorovi mogu imati nula ili više djece, ali svaki čvor osim korijena ima točno jednog roditelja dok je korijen jedini čvor bez roditelja. Za obilazak stabla koriste se metode preorder, inorder i postorder.

Graf se sastoji od skupa čvorova i skupa grana koje povezuju čvorove, a mogu biti usmjereni i neusmjereni. Imaju primjenu u mnogo različitih slučajeva, npr. modeliranje gradskih ulica pomoću grafova ili modeliranje računalnih mreža. Prikazivanje na računalu moguće je pomoću matrice incidencije te matrice susjedstva. Implementacija se može izvesti na više načina, stoga je potrebno prilagoditi implementaciju potrebama.

Skup je točno zadan ako možemo odrediti pripada li element skupu ili ne, svi elementi moraju biti istog tipa, a vrijednost im mora biti različita. Operacije apstraktnog tipa podataka skupa definiraju se kao i operacije nad skupovima u matematici, a to su unija, presjek, razlika. Kod implementacije skupa pomoću vezane liste, lista mora biti sortirana zbog efikasnijeg rada operacija.

Rječnik je posebna vrsta skupa gdje se ne koriste složene operacije kao kod skupa, nego se izvršavaju samo povremena izbacivanja i ubacivanja. Također podaci moraju biti istog tipa i ne smiju postojati dva podatka s isto vrijednošću. Implementacija se prikazuje pomoću binarnog stabla traženja.

Prioritetni red isto pripada jednoj od vrsta skupa gdje se ubacuje novi element, a izbacuje onaj s najmanjim prioritetom. Kod implementacije se prikazuje hrpom gdje svaki element odgovara jedan čvor hrpe. Jednostavan primjer prioritetnog reda je red čekanja kod liječnika gdje pacijent s najopasnijom ozljedom ima prednost.

Literatura

1. Boljiprogramer (2018), dostupno na: http://boljiprogramer.com/napredno-programiranje/algoritmi-sa-grafovima/uvod-u-teoriju-grafova
2. Btechsmartclass (2015), dostupno na: http://btechsmartclass.com [9.8.2018.]
3. Cplusplus, (2000), dostupno na: www.cplusplus.com/reference [12.6.2018.]
4. Divjak, B., Prezentacije s kolegija Diskretne strukture s teorijom grafova, 2008, DSTG7 grafovi. [17.8.2018.]
5. Drozdek, A. (2001) Data Structures and Algorithms in C++. 2. izd. Books/Cole. Dostupno na: http://library.aceondo.net/ebooks/Computer_Science/Data_Structure_And_Algorithms_In_C++2nd.ed-Adam.Drozdek.pdf [25.8.2018.]
6. Đurišić, M. (2014) Pretraga grafa u širinu (bfs). Dostupno na: http://grafovi.weebly.com/pretraga-grafa/pretraga-grafa-u-sirinu-bfs
7. Geeksforgeeks, (2018), dostupno na: https://www.geeksforgeeks.org/abstract-data-types/ [23.6.2018.]
8. Goodrich, M, Tamassia, R, Mount, D. (2011) Data Structures and Algorithms in C++. 2. izd. Hoboken: John Wiley & Sons, Inc. Dostupno na: https://o6ucs.files.wordpress.com/2012/11/data-structures-and-algorithms-in-c.pdf [10.7.2018.]
9. Manger, R. (2013) Strukture podataka i algoritmi. 4. izd. Zagreb. [15.7.2018.]
10. Marinović, M. (2004) Teorija grafova. Seminarski rad. Zagreb: Fakultet elektrotehinke i računarstva, dostupno na: http://www.zemris.fer.hr/predmeti/mr/arhiva/2002-2003/seminari/finished/pdf/grafovi.pdf [5.9.2018.]
11. Math.rs (2013) Algoritmi na grafovima. Dostupno na: http://www.math.rs/p/files/16-Operaciona_istra%C5%BEivanja_-_ve%C5%BEbe_CELOBROJNO_PROGRAMIRANJE_4.pdf [25.8.2018.]
12. Motik, B. i Šribar, J. (1997) Demistificirani C++. 4. izd. Zagreb [27.7.2018.]
13. Morin, P. (2014) Open Data Structures (in C++). 1. izd. Autoedición. Dostupno na: http://opendatastructures.org/versions/edition-0.1e/ods-cpp.pdf [7.7.2018.]
14. Ruđer Bošković Institute. Skupovi, dostupno na : http://lnr.irb.hr/soya/nastava/predavanje07-cb-5.pdf [12.8.2018.]
15. Skupovi brojeva. Zagreb: Element, dostupno na: https://element.hr/artikli/file/1301

16. Techiedelight (2018), dostupno na: http://www.techiedelight.com/find-maximum-width-given-binary-tree/ [1.9.2018.]
17. Tomašević, M (2004). Algoritmi i strukture podataka. Beograd. [9.7.2018.]
18. Veljan, D. (2003) Konačna matematika s teorijom grafova, Algoritam, Zagreb, [6.9.2018.]
19. Višić, F. (2011) Napredne strukture podataka. Završni rad. Varaždin: Fakultete organizacije i informatike, dostupno na: https://bib.irb.hr/datoteka/535674.fvisic.pdf [30.8.2018.]
20. Weiss, M. A. (2014) Data Structures and Algorithm Analysis in C++. 4. izd. Florida: Pearson Education, Inc. Dostupno na: http://iips.icci.edu.iq/images/exam/DataStructuresAndAlgorithmAnalysisInCpp_2014.pdf [29.6.2018.]
21. Živković, D. (2010) Uvod u algoritme i strukture podataka. 1. izd. Beograd: Singidunum. Dostupno na: https://www.vps.ns.ac.rs/Materijal/mat21058.pdf [22.7.2018.]

Popis slika

Popis tablica

Popis implementacija