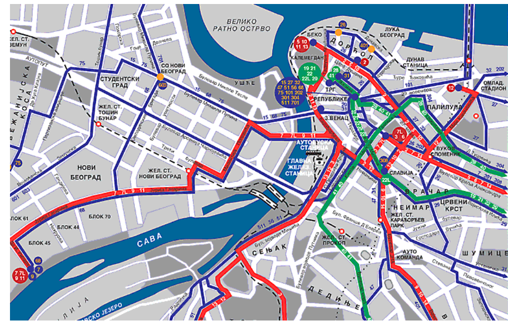
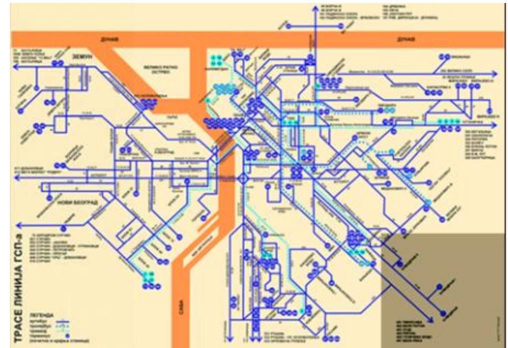
APSTRAKCIJE U PROGRAMIRANjU





Apstrakcije

-Osnovni problem u programiranju je složenost problema.

-Ne može se ceo problem posmatrati i rešavati odjednom.

-Složenost se rešava apstrakcijama, kontrolisanim uvođenjem detalja.

-Detalji se zanemaruju na nekom nivou apstrakcije kako bi se broj koncepata sa kojima se projektant suočava sveo na razumnu meru.

-*Primer* apstrahovanja je ***plan grada***

-Apstrakcije predstavlja opis nekog fenomena za neke potrebe u kome su svesno uklonjeni detalji nepotrebni sa aspekta datih potreba.

-Moguće je kreirati različite apstrakcije za dati fenomen, od kojih svaka uklanja, odnosno ostavlja različite detalje za različite potrebe.

-Primer: mapa gradskog prevoza, mapa noćnog prevoza

-Različiti pogledi ili aspekti

-Konkretizacija

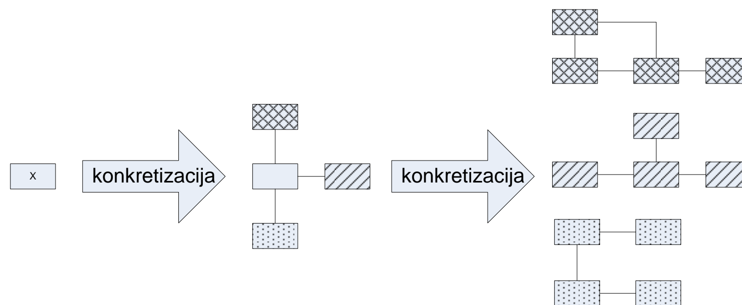
-Suprotno od asptrakcije, dobija se postupkom konkretizacije (detaljisanja)

-Apstrakcija/konkretizaija su relativni - ono što je na jednom nivou konkretizacija, na sledećem se može posmatrati kao apstrakcija

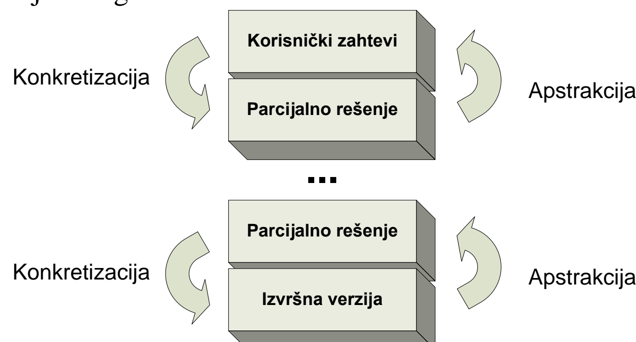
-Hijerarhijina apstrakcija - uzastopnom primenom postupka apstrahovanja/konkretizovanja

-Savremeni pristupi razvoju softvera baziraju softverski sistem na modelu realnog sveta u okviru koga on treba da se izvršava

-Postupak apstrakcije/konkretizacije se koristi kako bi se od korisničkih zahteva stiglo do izvršnog koda softverskog sistema koji zadovoljava date zahteve.



-Na vrhu hijerarhije su korisnički zahtevi kao krajnja apstrakcija softverskog sistema, a na dnu je izvršna verzija sistema kao krajnja konkretizaciju datog sistema



-**Specifikacija** je opis apstrakcije
-**Implementacija** neke apstrakcije je njena konkretizacija na nekom implementacionom sredstvu
-Neformalne i formalne specifikacije
-Specifikacije je formalna ako je data na jeziku čija je sintaksa i semantika formalno definisana

Proceduralne apstrakcije
-Proširenje skupa naredbi datog programskog jezika
-Uvode se ***zbog velikog semantičkog jaza*** između problema kolji se rešava i nivoa naredbi programskog jezika
-**Dekompozicijom** (top down) se svode na jednostavnije (na nižem nivou apstrakcije) proceduralne apstrakcije sve dok se ne dodje do primitivnih (operacija implementacionog sredstva)
-***Top-down stepwise refinement*** postupak razvoja programa
-Šta je primitivni nivo zavisi od imlementacionog sredstva
-Npr. *sin(x)* je primitivna operacije u višim programskim jezicima, a složena (apstraktna) u mašinskom jeziku

```
procedure izracunajPlatuRadnika(r)
  begin
    bruto = izracunajBrutoPrimanja(r);
    odbici = izracunajOdbitkeRadnika(r);
    plata = bruto - odbici;
end;
procedure izracunajBrutoPrimanja(r)
  begin
    bruto = brojBodova * vrednostBoda;
end;
procedure izracunajOdbitkeRadnika(r)
  begin
     odbici = sindikalnaClanarina(r) + rataKredita(r);
end;
```

Apstrakcije podataka
-**Podatak** je nosilac informacije. **Informacija** je protumačeni podatak.
-Konkretna vrednost podatka se može tretirati kao predstava nekog objekta iz realnog sistema.
-Npr:
5 - predstavlja temperaturu
("Pera Perić", "143/2009", 4) - predstavlja studenta

Vrste apstrakcije podataka
-Postoji tri načina kako se podaci mogu apstrahovati:
-Klasifikacija
-Agregacija i
-Generalizacija
-Ovi načine definišu tri osnovne vrste apstrakcija podataka

Klasifikacija
-Svi objekti istih osobina se klasifikuju u jedan tip
-npr. Ceo broj (integer), Student
-Umesto da se posmatraju pojedinačne instance, dovoljno je posmatrati tipove
-Tako se broj koncepata koji se razmatra značajno redukuje, tj. umanjuje složenost
-Tip predstavlja opšti, generički opis osobina koje svaka instanca ima

Tip podatka se definiše se kao skup objekata (vrednosti) i operacija nad njima.
-Int : Z, {+, -, *, }
-Student: S, {upisiSemstar, poloziIspit, …}
-Važno je uočiti da definicija obavezno uključuje i operacije
-tipovi podataka bez operacija nemaju mnogo smisla
-Tipovi se mogu posmatrati :
-Prema načinu implementacije
-Prema složenosti

-Tipovi prema načinu implementacije:

(1) ***Fizički tipovi***

-tipovi podržani od strane računara (hardvera);

-npr: int, char

(2) ***Jezički (virtuelni) tipovi***

-tipovi podataka koji su podržani od strane implementacionog jezika;

-npr: zapis, niz

(3) ***Apstraktni tipovi podataka (ATP)***

-proširenje skupa tipova implementacionog sredstva;

-npr: student, radnik

-Tipovi prema složenosti:

(1) ***Prosti (primitivni) tipovi***

-Vrednost prostog tipa je nedeljiva

-npr: int -> 5

(2) ***Složeni (strukturni) tipovi***

-Dobijaju se agregiranjem više drugih tipova

-Vrednost je složena, sastoji se od drugih vrednosti;

-npr: zapis -> ("Pera Perić", "143/2009", 4)

-Strukturni tipovi se nazivaju i Strukture podataka

Agregacija

-Dobija se agregiranjem više tipova u jedan apstraktniji složeni tip.

-Objekti u strukturi ne moraju biti istog tipa

-Npr. niz znakova (string) i ceo broj u prethodnom primeru

-Većina klasičnih programskih jezika podržava agregaciju

Generalizacija i specijalizacija

-Uočavanjem zajedničkih osobina i operacija tipovi se apstrahuju u generički apstraktniji tip

-Obrnut postupak je specijalizacija

-Uzastopnom primenom generalizacije i specijalizacije se dobija hirerarhija tipova

-Operacije i osobine se nasleđuju od nadtipa

-Podtip može imati dodatne operacije i osobine

-Primer

-Osoba

-Ime, Prezime, Datum rođenja

-Radnik - podtip od Osoba

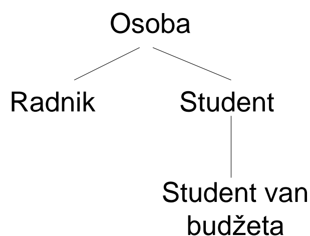
-Organizacija, Staž, Plata

-Student - podtip od Osoba

-Fakultet, godina studija, prosek

-Student van buxeta- podtip od Student

-Iznos školarine



POJAM STRUKTURA PODATAKA

-Strukture podataka su složeni tipovi podataka

-Nastaju primenom apstrakcije agregacije, tj. objedinjavanjem više jednostavnijih tipova podataka

-Svaka konkretna vrednost strukture podataka je jedna instanca odgovarajućeg tipa

-Npr. ("Pera Perić", "143/2009", 4) je jedna instanca strukture podataka kojom se predstavljaju studenti

Vrste struktura podataka

-Svaka konkretna vrednost strukture podataka se sastoji od jednostavnijih delova, koji predstavljaju njene elemente

-Elementi mogu biti ***prosti*** ili ***složeni***, tj. strukture podataka za sebe

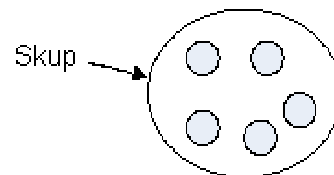
-Na taj način se mogu graditi strukture proizvoljno složene strukture

-Npr. zapis (“Pera Perić”, “143/2009”, 4) se sastoji od tri polja elementa:
-Ime - niz znakova (složen tip)
-Broj indeksa- niz znakova (složen tip)
-Godina studija - ceo broj (prost tip)

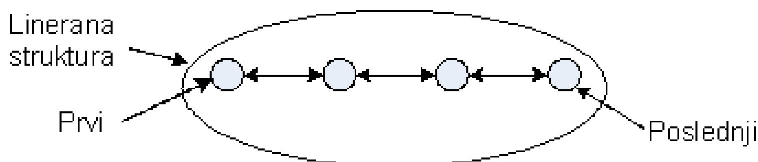
-Elementi u strukturi podataka mogu biti u nekom međusobnom odnosu (relaciji)
-Stoga se struktura podataka može formalno definisati kao **uređen par S =(E, r)**
-E je ***skup elemenata***
-r je ***binarna relacija*** definisana nad E kojom je uređen skup E
-Pojedine vrste struktura podataka se izvode na osnovu toga kakav je njihov međusobni odnos, tj. kakve su osobine relacije r
-Najvažnija osobina relacije r je **kardinalnost**, tj. koliko elemenata odgovara jednom elementu u datoj strukturi i obrnuto

-Na osnovu kardinalnosti relacije r mogu se definisati sledeće vrste struktura:
-**0:0** (ne postoji uređenje) - ***kolekcije i skupovi***
-**1:1** (linearno uređenje) - ***linearne strukture***
-**1:M** - ***stabla***
-**M:M** - ***grafovi i mreže***

Korelacije i skupovi
-Elementi u strukturi nemaju međusobni odnos, tj. kardinalnost relacije r je ***0:0***
-Samo se može znati da li je neki element u strukturi ili ne
-Bliske su pojmu **skupa** iz matematike, pa se tako i nazivaju
-***Skup nema duplikate*** (jedan isti element ne može biti dva puta član skupa)
-***Kolekcija može imati duplikate*** (jedan isti element može biti dva puta član skupa)

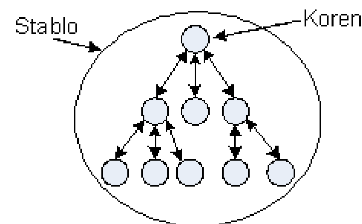


Linearne strukture
-Elementi u strukturi su linerano uređeni, , tj. kardinalnost relacije r je ***1:1***

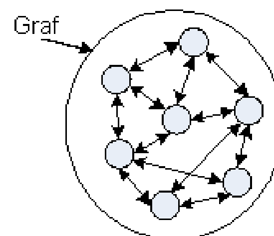


-Osim jednom posebnom elementu koji se naziva ***prvi***, svakom elementu ***prethodi tačno jedan element***
-Osim jednom posebnom elementu koji se naziva ***poslednji***, svakom elementu ***sledi tačno jedan element***

Stabla -hijerarhijske strukture
-Elementi u strukturi su hijerarhijski uređeni, tj. kardinalnost relacije r je ***1:M***
-Osim jednom posebnom elementu koji se naziva ***koren***, svakom elementu ***prethodi tačno jedan element***
-Svakom elementu ***može slediti više elementa***



Grafovi i mreže
-Elementi u strukturi su uređeni bez ograničenja, tj. kardinalnost relacije r je ***M:M***
-Svakom elementu može ***prethoditi više elemenata***
-Svakom elementu može ***slediti više elementa***



-Navedene opšte strukture podataka pokrivaju sve moguće slučajeve
-Svaka konkretna struktura podataka u praksi se ***svodi na neku od navedenih***

Pojam apstraktnih tipova
-***Apstraktni tipovi podataka (ATP)*** su korisnički tipovi podataka kod kojih je sakriven način implementacije
-definisani su od strane programera i predstavljaju proširenje skupa tipova datog programskog jezika
-Vidljiva je samo ***specifikacija ATP***
-Razdvajanje specifikacije od implementacije
-Način realizacije interne strukture podataka za predstavljanje i pamćenje vrednosti datog ATP i algoritama za specificirane operacije ATP-a je ***sakriveno***
-***Moguće su različite realizacije istog ATP!!!***

-U modernim OO jezicima (npr. Java, C#) specifikacija se ATP daje preko koncepta **interfejsa**
-***Interfejs*** definiše skup operacija koje se mogu pozivati nad vrednostima datog tipa
-Operacije se mogu odnositi na osobine (atribute) ili predstavljati neko ponašanje objekata iz realnog sveta

-Primer specifikacije ATP preko Java interfejsa
```
interface BankovniRacun {
        double Stanje();
        void UloziNovac(double iznos);
        void PodigniNovac(double iznos);
}
```

Interfejs
-Interfejs definiše samo sintaksu tj. način korišćenja (pozivanja) operacija
-Semantika (značenje) nije obuhvaćena interfejsom
-Prava potpuna specifikacija obuhvata i sintaksu i semantiku
-Specifikacija semantike je poseban problem

Klase
-U modernim OO jezicima (Java, C#) se ATP realizuje preko koncepta klase
-Klasa definiše skup metoda kojima se implementiraju operacije datog tipa
-Obuhvata i definisanje privatnih članova kojima se pamte vrednosti datog tipa
-Jedna klasa može implementirati više tipova (tj. interfejsa)

-Primer realizacije ATP preko Java klase
```
class BK implements BankovniRacun {
         private double stanje;
        double Stanje() {
                return stanje
        }
        void UloziNovac(double iznos) {
                stanje += iznos;
        }
        void PodigniNovac(double iznos){
                stanje -= iznos;
        }
}
```

Programiranje zasnovano na apstraktnim tipovima
-Pristup programiranju zasnovan na apstraktnim tipovima i apstrakcijama podataka se je ***suština objektno-orijentisanog programiranja***
-Klasično programiranje i jezici ne podržavaju ATP i ne podržavaju sve apstrakcije (npr. generalizaciju, tj. nasleđivanje tipova)

-Razvoj programa preko ATP se odvija u sledećim fazama
-***Definiše*** (tj. specificira) se skup ATP koji su pogodni za problem koji se rešava
-***Aplikacija*** (glavni program) ***se razvija*** pomoću operacija definisanih ATP
-Definisani ATP se ***implementiraju*** preko drugih postojećih tipova podataka (uključujući druge prethodno razvijene ATP)

-Prednosti korišćenja ATP su:
-ATP se može formalno tretirati kao ***algebra***
-Razvojem programa preko ATP se ostvaruje ***princip sakrivanja informacija***
-Postiže se ***modularnost*** što za posledicu ima olakšano organizaciju razvoja i održavanje programa
-Omogućava se ponovno korišćenje koda (eng. ***reuse***) u programima koji koriste iste ATP

Realizacija strukture podataka na računaru
-Opšte operacije
-*Kreiranje* (inicijalzacija)
-Konstruktori (new)
-*Izbacivanje*
-Skupljač smeća
-*Ubaci*
-*Izbaci*
-*Pretraživanje*
-*Obilazak*

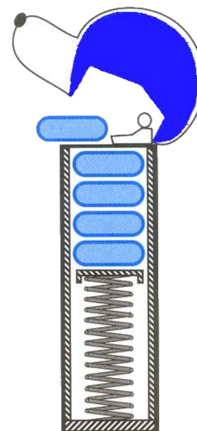
-Opšti načini realizacije
-Statička realizacija
-Nizovi
-Prednost: jednostavnost
-Nedostatak: neracionalno korišćenje memorije
-Dinamička realizacija
-Spregnute strukture
-Prednost: racionalno korišćenje memorije
-Nedostatak: složenost

LINEARNE STRUKTURE

Razlikuju se po mestu (lokaciji) gde se vrši ubacivanje i izbacivanje elemenata
(1) ***Stek*** (eng. stack) - ubaci i izbaci sa istog kraja
(2) ***Red*** (eng. queue) - ubaci na jednom, izbaci sa drugog kraja
(3) ***Dvostruki red*** (eng. double queue - Deque) - ubaci izbaci samo na krajevima
(4) ***Lista*** (eng. list) - ubaci/izbaci bilo gde

Stek

-**LIFO** struktura (Last In First Out)
-Primer: Držač za Pez bombone
-Osnovne operacije
-Ubacivanje (Push)
-Izbacivanje (Pop)
-Ostale operacije
-Vrati vrh (Peek)
-Prazan stak?
-Pun stak?
-Broj elemenata



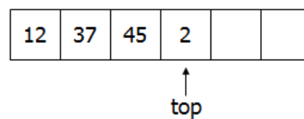
-Definicija preko ATP

```
public interface IStack{
        boolean isEmpty();
        void Push(int obj);
        int Pop();
        int Peek();
}
```

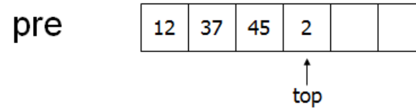
-Implementacija preko niza

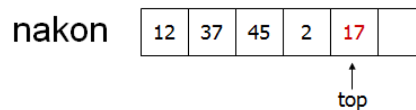
-Kao ***statička struktura***

-Primer:

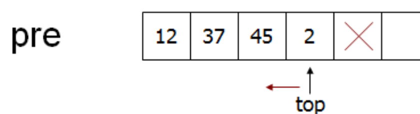


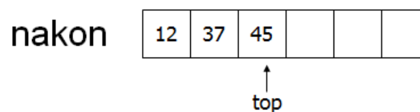
-Ubacivanje





-Izbacivanje

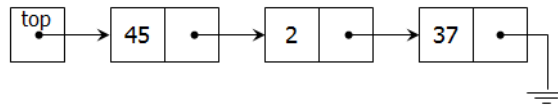




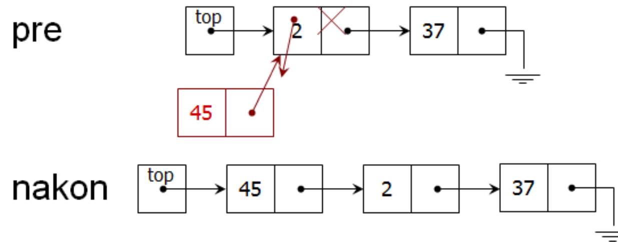

```
public class ArrayStack implements IStack {
        protected int[] data;
        protected int top;
        public ArrayStack(int capacity){
                data = new int[capacity];
                top = -1;
        }
        public boolean IsEmpty(){
                return top == -1;
        }
        public void Push(int obj){
                if(top < data.length - 1)
                        data[++top] = obj;
        }
        public int Pop(){
                if(IsEmpty())
                        return Integer.MIN_INT;
                int Obj = data[top--];
                return Obj;
        }
        public int Peek(){
                if(IsEmpty())
                        return Integer.MIN_INT;
                return data[top];
        }
}
```

-Implementacija preko dinamičkih struktura

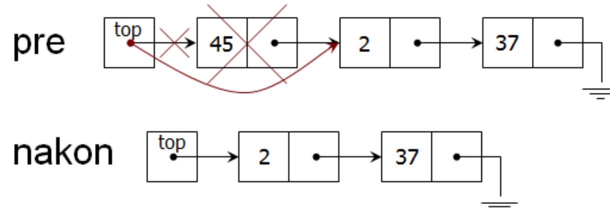
-Primer:



-Ubacivanje

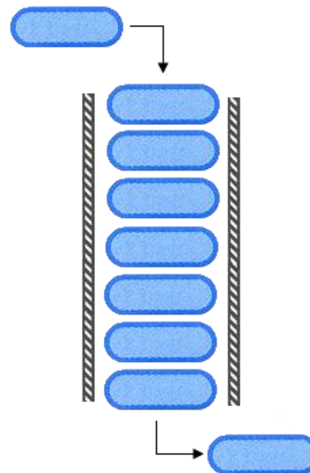


-Izbacivanje



Red

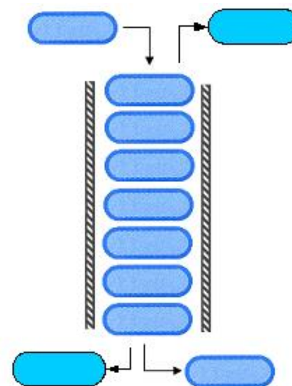
-**FIFO** struktura (First In First Out)
-Primer: Red u prodavnici



-Osnovne operacije
   -Ubacivanje (Enqueue)
   -Izbacivanje (Dequeue)
-Ostale operacije
   -Broj elemenata
   -Prazan red?
   -Pun red?
-Definicija preko ATP

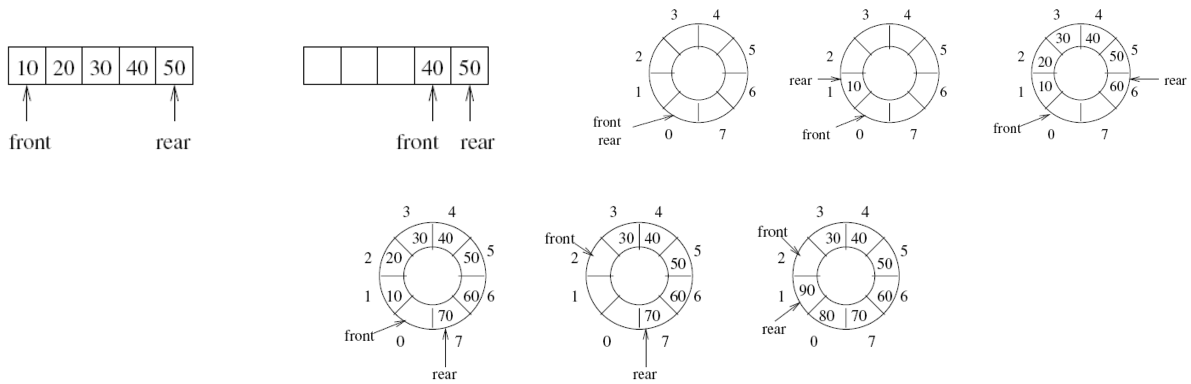
```
public interface IQueue{
        boolean isEmpty();
        void Enqueue(int obj);
        int Dequeue();
        int Peek();
}
```

Dvostruki red



-Osnovne operacije
   -Ubaci na kraj (append)
   -Ubaci na početak (prepend)
   -Izbaci sa kraja (delete last)
   -Izbacisa početka (delete first)
-Ostale operacije
   -Broj elemenata
   -Vrati Prvi
   -Vrati Zadnji

-Implementacija preko niza



Lista

-***Linerana struktura bez ograničenja*** na mesto ubacivanja ili izbacivanja

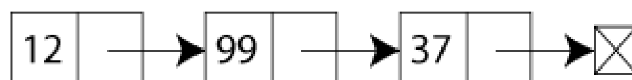
-Operacije

- -Ubacivanje
- -Izbacivanje
- -Broj elemenata
- -Brisanje svih elemenata

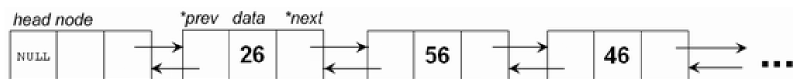
-Definicija preko ATP

```
public interface ILinkedList{
        void InsertBefore(int Data);
        void InsertAfter(int Data);
        int Remove();
        boolean MovePrevious();
        boolean MoveNext();
        void MoveTo(int index);
        void Clear();
        int Count();
}
```

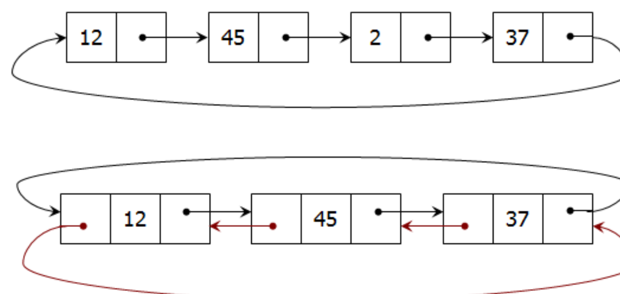
-Jednostruko spregnuta lista (singly-linked)



-Dvostruko spregnuta lista (doubly-linked)



-Ciklična lista (Circularly-linked list)



ANALIZA EFIKASNOSTI ALGORITAMA

-Problem upoređivanja dva algoritma koji obavljaju isti zadatak
-Apsolutno vreme nije objektivna mera upoređivanja
-Zavisi od brzine računara, jezika u kome je napisan algoritam, trenutnih okolnosti u kojima se izvršava dati program, itd.
-Potrebna je objektivna mera za upoređivanje
-Problem analize efikasnosti algoritama je nalaženje objektivne mere

Pojam kompleksnosti
-Vremenska i prostorna kompleksnost
-Potrebno vreme i prostor nekom algoritmu da obavi zadatak
-Ovde se daje fokus na vremensku kompleksnost
-Iskazuje se kao ***funkcija veličine problema*** koji se rešava
-Primeri
-Sortiranje niza dimenzije N je funkcija koja zavisi od N
-Množenje matrice dimenzije MxN je funkcija koja zavisi od parametara M i N.

Analiza kompleksnosti
-Načini za utvrđivanje vremenske kompleksnosti
-Broj operacija
-Broj koraka
-Broj promašenih pogodaka u keš memoriji

Broj operacija
-Broj operacija koje treba da se izvrše
-Odabira se skup pogodnih operacija (npr. sabiranja, množenja, ubacivanja, poređenja, sl) i utvrđuje se koliko se tih operacija obavi bi algoritam uradio dati zadatak
-Primer: nalaženje pozicije (indeksa) najvećeg broja u nizu celih brojeva

```
int max(int [] a, int n){
        if (n < 1) return -1; // no max
        int positionOfCurrentMax = 0;
        for (int i = 1; i < n; i++)
                if (a[positionOfCurrentMax] < a[i]) positionOfCurrentMax = i;
        return positionOfCurrentMax;
}
```

-Broj poređenja: n-1 za n>1

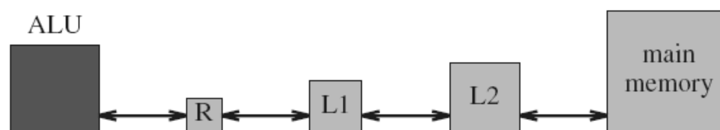
Broj koraka
-Broj koraka (instrukcija) koje treba da se izvrše
-***Korak*** je jedinica koja je nezavisna od veličine problema
-***Računa se tako što se analizira***:
-Koliko se koraka izvrši za svaku naredbu
-Koliko se svaka naredba izvršava puta

```
0    →   int max(int [] a, int n) {
1    →           if (n < 1) return -1; // no max
1    →           int positionOfCurrentMax = 0;
n-1  →           for (int i = 1; i < n; i++)
1    →                   if (a[positionOfCurrentMax] < a[i]) positionOfCurrentMax = i;
1    →           return positionOfCurrentMax;
         }
```

Broj koraka: 1+1+(n-1)*1 + 1 = n+2

Broj promašaja
-Merenje broja operacija ili koraka je opravdano ***kada je vreme izvršavanja neuporedivo veće od vremena potrebnog za pribavljanje podataka***
-Danas je u mnogim slučajevima obrnuto
-Podaci se nalaze na disku (10^9 sporiji pristup od operativne memorije)

-Koristi se ***keš memorija***
-Uvodi se ***hijerarhija keš memorija***



-Na brzinu algoritma najviše utiče ***koliko podataka je već u kešu***
-Redosled i način pristupa podacima je bitan
-Npr. da li se u matrici elementima pristupa redovima ili kolonama
-***Meri se broj promašaja***, tj. koliko puta se mora ići po podatke koji nisu u kešu

-***Efikasnost algoritma nije uvek tačno zavisna samo od dimenzije problema***
-Npr. u slučaju sortiranja utiče i koliko je prethodno niz već uređen
-Stoga se ***traži najbolji, najgori i prosečan slučaj efikasnosti algoritma***
-Primer: naći prvo pojavljivanje broj x u nizu
-Najbolji slučaj: 1
-Najgori: n
-Prosek: (n+1)/2

Asimptotska kompleksnost
-p(n) je ***asimptotski veće od*** q(n), akko:

$$\lim_{n\to\infty} q(n)/p(n) = 0$$

-Kaže se i da je q(n) je asimptotski manje od p(n)
-Kaže se da su asimptotski jednaki ako ni p(n) niti q)n) je asimptotski veće od onog drugog

-Primer 1: $(10n+7)$ ***i*** $(3n^2 + 2n + 6)$

$$\lim_{n\to\infty} (10n+7)/(3n^2 + 2n + 6) = (10/n+7/n2)/(3 + 2/n + 6/n2) = 0/3 = 0$$

$(3n^2 + 2n + 6)$ je asimptotski veće od $(10n+7)$, tj. $(10n+7)$ je asimptotski manje od $(3n2 + 2n + 6)$

-Primer 2: $(10n+7)$ ***i*** $(2n + 6)$

$$\lim_{n\to\infty} (10n+7)/(2n + 6) = \infty/\infty$$

$(10n+7)$ *i* $(2n + 6)$ su asimptotski jednaki

-Veliko O notacija (Big O notation)
- Notacija $f(n) = O(g(n))$ (čita se "***f(n) je veliko o od g(n)***") znači da je $f(n)$ asimptotski manje ili jednako od $g(n)$.
-U asimptotskom smislu, ***g(n) je gornja granica za f(n)***

10n + 6 = O(n)
6n2 + 3n + 12 = O(n2)

-Uzima se u obzir najveći član
-Asimptotska uređenost:

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

Term	Naziv
1	konstanta
$\log n$	logaritmaska
n	linerana
$n \log n$	$n \log n$
n^2	kvadratna
n^3	kubna
2^n	eksponencijalna
$n!$	faktorijel

PRETRAŽIVANJE LINEARNIH STRUKTURA

Problem pretraživanja

-Pronaći element u nekoj strukturi podataka na osnovu njegovog ključa

-Svi elementi obično imaju neki atribut čija je vrednost jedinstvena. Ovaj atribut se naziva ključ ili identifikator

-Vrlo čest problem u praksi

-U većini slučajeva je to najčešća operacija

-Algoritam pretraživanja dosta zavisi od načina organizacije elemenata unutar strukture

-Vrlo često se određena struktura podataka i bira zbog pogodnosti pretraživanja

Sekvencijalno pretraživanje

-***Jednostavan algoritam***

-Porede se elementi redom počev od početka pa do kraja strukture

-Primenljiv i u slučaju realizacije preko niza i liste

-U slučaju niza se prelazi na sledeći element povećavanjem indeksa

-U slučaju liste, prelazak se ostvaruje preko pokazivača na sledeći element

```
int sekPret(int [] A, int k, int N) {
        for (int i=0; i < N; i++) {
                if (A[i] == k) return i;
        }
        return -1; // nije nadjen
  }
```

-Efikasnost pretraživanja

-***Najbolja***: ***O(1)*** – kada je element na početku

-***Najgora***: ***O(n)*** - kada je element na kraju

-***Prosečna***: ***O((n+1)/2)***

-$E = 1*P_1 + 2*P_2 + 3*P_3 + \dots n*P_n$

-Verovatnoće $P_1 \dots P_n$ su iste, tj. jednake $1/n$

-$E = 1* 1/n + 2* 1/n + 3* 1/n + \dots n* 1/n$

-$E = (n+1)/2$

-Poboljšanja algoritma

-Uređivanje niza prema verovatnoći pretraživanja

-Varijante algoritma:

-Metoda transpozicije

-prebacivanje na početak elementa koji je pretražen

-Metoda zamene

-zamena elementa koji je pretražen sa prethodnikom, tj. pomeranje za 1 mesto unapred

-***Metoda transpozicije daje bolje rezultate***

-Ali u slučaju niza zahteva pomeranje svih elemenata niza – O(n) efikasnost

Binarno pretraživanje

-***Primenljiv samo u slučaju sortiranih nizova***

-Ideja algoritma:

-Ispitati element koji se nalazi ***na sredini i na osnovu toga odrediti deo niza*** koji treba dalje pretraživati

-***Dalje pretraživanje*** polovine niza se pretražuje na isti način ***rekurzivno***

```
int binPretRek(int [] A, int k, int dg, int gg) {
          int s;
    if (gg < dg)
       return -1; // nije nadjen
    s = dg+ ((gg - dg) / 2);
    if (A[s] > k)
       return binPretRek(A, k, dg, s-1);
```

```
        else if (A[s] < k)
            return inPretRek(A, value, s+1, gg);
        else
            return s; // nadjen
    }

int binPretIter(int [] A, int k, int N) {
            int s,
            int dg = 0;
            int gg = N-1;
        while (gg < dg) {
                    s = dg+ ((gg - dg) / 2);
            if (A[s] > k)
                        gg = s-1;
            else if (A[s] < k)
                    dg= s+1;
            else        return s; // nadjen
                    }
                    return -1; // nije nadjen
    }
```

-Algoritam u svakoj iteraciji deli niz na dva jednaka dela

-Uzastopno delenje je ***log*** funkcija

-Efikasnost binarnog pretraživanja

-***Najgori slučaj – O(log N)***

-***Prosečni – O(log(N) -1)***

Interpolaciono pretraživanje

-Primenljivo samo u slučaju sortiranih nizova

-Simulira postupak koji koriste ljudi kod pretraživanja npr. ***telefonskog imenika***

-Ako se traži neko ko se preziva "Antić" onda će se imenik pokušati otvoriti stranica imenika koja je bliža početku

-Ako se traži neko ko se preziva "Džaković" onda će se imenik pokušati otvoriti stranica imenika koja je bliža kraju imeneika

-Pretpostavka da ako je početno slovo prezimena bliže početku azbuke, onda se dato prezime nalazi bliže početku imenika

-Ideja algoritma:

-Postoji veza između pozicije vrednosti ključa u skupu vrednosti ključeva i pozicije u nizu zapisa sa tim ključem, pa se može izračunati (interpolirati) pozicija traženog elementa.

-Ispitati element koji se nalazi na interpoliranoj poziciji i na osnovu toga odrediti deo niza koji treba dalje pretraživati

-Dalje pretraživanje polovine niza se pretražuje na isti način ***rekurzivno*** (slično kao kod binarnog pretraživanja)

$$\frac{P_k - P_{\min}}{P_{\max} - P_{\min}} = \frac{K - MIN}{MAX - MIN}$$

- K vrednost ključa
- Pk pozicija zapisa sa ključem
- Pmin (Pmax) pozicija zapisa sa minimalnom (maksimalnom) vrednošću ključa
- MIN (MAX) minimalna (maksimalna) vrednost ključa

$$P_k = P_{\min} + \frac{K - MIN}{MAX - MIN} * (P_{\max} - P_{\min})$$

-Pod pretpostavkom da su ključevi uniformno raspoređeni, interpolaciono pretraživanje zahteva prosečno ***$O(\log_2 \log_2 N)$***

-Ako ključevi nisu uniformno raspoređeni tad performanse algoritma mogu biti znatno pogoršane: ***O(N)***

-**<u>Robusno interpolaciono pretraživanje</u>** ili **brzo (fast search**)

-uvođenje promenljive **razmak (R)** tako da je uvek **Pk - Pmin** i **Pmax - Pk** veće od **R.**

-Početna vrednost

$$R = \sqrt{P_{\max} - P_{\min}} + 1$$

$$Pom = P_{\min} + \frac{K - MIN}{MAX - MIN} * (P_{\max} - P_{\min})$$

$$P_k = Minimum(P_{\max} - R, Maximum(Pom, P_{\min} + R)$$

-Robusno interpolaciono pretraživanje ***u proseku*** zahteva ***O(log2(log2N))***

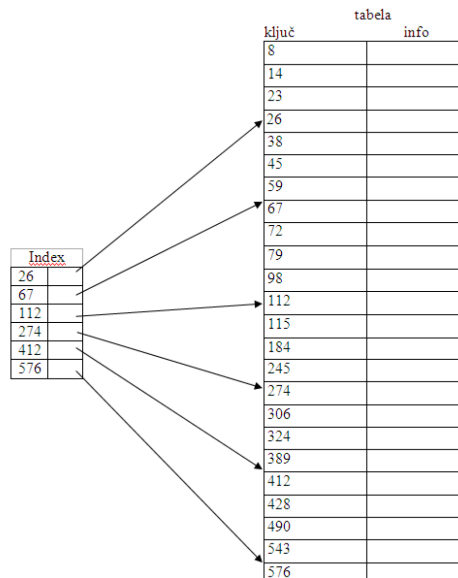
-U ***najgorem*** slučaju ***O((log2N))***^{2} poređenja.

<u>Indeks-sekvencijalno pretraživanje</u>

-Koristi dodatni pomoćni niz (tabelu) zvani **indeks**

-Pretraživanje ***počinje od indeksa*** traženjem ***prvog ključa koji je veći ili jednak*** traženom.

-Kada se naiđe na takav ključ onda se preko pokazivača ***pristupa bloku u kome se zapis pronalazi*** posle najviše *m* poređenja.



-**Prosečan broj poređenja** prilikom pretraživanja datoteke je jednak prosečnom broju poređenja za pretraživanje indeksa plus prosečnom broju poređenja za pretraživanje bloka:

$$P = \frac{k}{2} + \frac{m}{2} = \frac{k + \frac{n}{k}}{2} \quad \text{jer je} \quad m = \frac{n}{k}$$

-Optimalna vrednost za k se dobija iz uslova:

$$\frac{\partial P}{\partial k} = 0 \Rightarrow \frac{1}{2} + \frac{n}{2k^2} = 0 \Rightarrow k = \sqrt{n}$$

-**Dva nivoa indeksa** (slika dole)

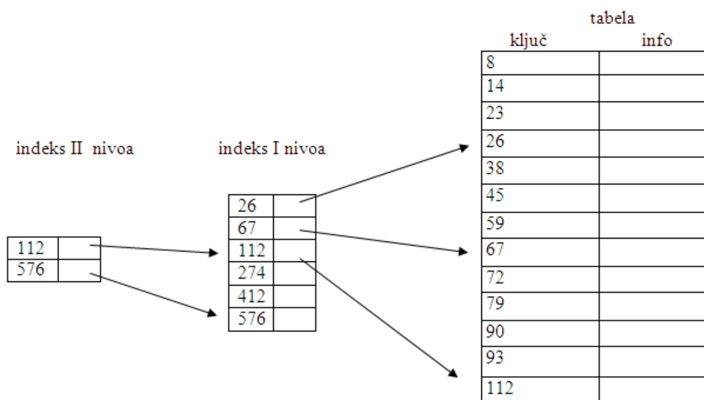
-**Problem ažuriranja** – naknadno ubacivanje u blok koji je popunjen nije moguće

-**Rešenje**

-Kod inicijalnog formiranje indeksa se ***ostavlja prazan prostor*** u bloku koji se koristio za naknadna ubacivanja

-Kada se blok napuni, onda se novi zapisi (tzv. prekoračioci) ***pamte u posebnoj memorijskoj zoni*** gde se olančavaju u 1-struko spregnutu listu

-Kada se zona za prekoračioce napuni, vrši se ***ponovno formiranje indeksa*** (vrlo skupa operacija)



SORTIRANJE

Problem sortiranja

-***Sortiranje*** je operacija koja uređuje sve elemente niza u rastućem ili opadajućem redosledu

a[i] <= a[i+1] za svako i=1,n - **rastući**

a[i] => a[i+1] za svako i=1,n - **opadajući**

-U mnogim situacijama su potrebni sortirani nizovi

-Primer algoritam za binarno pretraživanje

Algoritmi za sortiranje

-Metoda izbora - ***Selection sort***
-Metoda zamene -***Bubble (Sink) sort***
-Metoda umetanja - ***Insertion sort***
-Šelova metoda - ***Shell sort***
-Metoda gomile - ***Heap sort***
-Metoda spajanja - ***Merge sort***
-Brza metoda - ***Quick sort***
-…

Selection sort

-Sortiranje element po element

-Prolazak kroz nesortiran deo i pronalaženje najmanjeg/največeg elementa

-***Jednostavan*** za implementaciju

-Efikasan na malom broju podataka

-Efikasnost $O(n^2)$

-$n^2/2$ poređenja i n zamena.

```
public void SelectionSort(int[] aArray){
    int i, j;
    int min, temp;
    for (i = 0; i < aArray.length - 1; i++) {
        min = i;
        for (j = i + 1; j < aArray.length; j++) {
            if (aArray[j] < aArray[min])
                min = j;
        }
        temp = aArray[i];
        aArray[i] = aArray[min];
        aArray[min] = temp;
    }
}
```

8	2	4	9	5	6

2	8	4	9	5	6

2	4	8	9	5	6

2	4	5	9	8	6

2	4	5	6	8	9

Metoda zamene

-Upoređuje susedne elemente i zamenjuje ih ukoliko nisu u odgovarajućem redu

-Postoje dve verzije algoritma

-***bubble sort***

-***sink sort***

-$n^2/2$ poređenja i $n^2/2$ zamena u proseku i u najgorem slučaju

-**Najsporiji** od svih $O(n^2)$ algoritama

Bubble sort

-Počev od kraja niza upoređuje susedne elemente i zamenjuje ih ukoliko nisu u odgovarajućem redosledu

-Najmanja vrednosti "**isplivava na površinu**"

-Podseća na izranjanje **mehurića** iz vode

Sink sort

-Počev od početka niza upoređuje susedne elemente i zamenjuje ih ukoliko nisu u odgovarajućem redosledu
-Najveća vrednosti “**potone na dno**”

```
public void SinkSort(int[] aArray){
	int i, j, temp;
	for ( i = 0; i < aArray.length -1; i++ ){
		for ( j = 0; j < aArray.length -2 - i; j++ ){
			if ( aArray[j] > aArray[j+1] ){
				temp = aArray[j];
				aArray[j] = aArray[j+1];
				aArray[j+1] = temp;
			}
		}
	}
}
```

8	2	2	2
2	8	4	4
4	4	8	8
9	9	9	5
5	5	5	9
6	6	6	6

Insertion sort

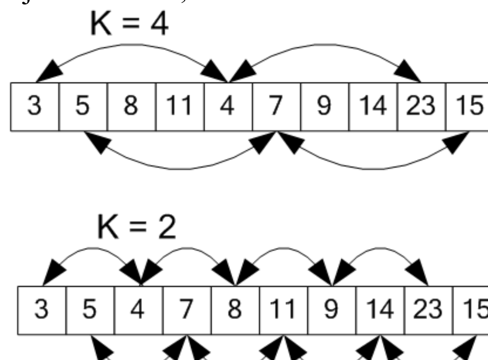
-Zasnovan na tehnici koja se koristi **u igrama sa kartama**
-Prolazak kroz nesortiran deo i ubacivanje sledećeg elementa u sortiran deo
-**Jednostavan** za implementaciju
-Efikasan na malom broju podataka
-Kompleksnost ***$O(n^2)$:***
 -***$n^2/4$*** poređenja i ***$n^2/4$*** pomeranja u ***proseku***, a ***duplo više u najgorem slučaju***

```
public void InsertionSort(int[] aArray){
	int i, j, index;
	for ( i = 1; i < aArray.length; i++ ){
		index = aArray[i];
		j = i;
		while ( (j > 0) && (aArray[j-1] > index) ){
			aArray[j] = aArray[j-1];
			j--;
		}
		aArray[j] = index;
	}
}
```

8	2	4	9	5	6
2	8	4	9	5	6
2	4	8	9	5	6
2	4	5	8	9	6

Shell sort

-**Metoda umetanja** (insertion sort) je spor jer se zamene odnose **samo elemente koji su fizički susedi**
 -ako je najmanji element na kraju niza, potrebno je n koraka da dodje na početak
-Šelova metoda je ***proširenje metode umetanja koja dozvoljava da se udaljeni elementi zamenjuju***
-Ideja je da svi elementi koji su međusobno k-udaljeni budu sortirani
-Ceo niz se sastoji od k podnizova koji su sortirani, ali su međusobno izmešani



-Ako je ***K veliko***, tada ce ***elementi sa manje koraka brže biti pomerani ka svojim pravim pozicijama***

-Uzastopno sortiranje umetanjem sa smanjivanjem K do 1 je suština Šelove metode
-**Problem** je ***kako odrediti k i inkremente njegovog smanjenja*** tj. kako generisati sekvencu rastojanja
-dobro je generisana sekvenca rastojanja bude logaritamska, tj. da od k do 1 bude logaritamski broj koraka
-Šelova metoda je ***jednostavna za implementaciju***, a daje ***dobre rezultate i za velike nizove***

```
static void shell(int[] a, int l, int r) {
   int k;
   for (k = 1; k <= (r-l)/9; k = 3*k+1);
   for ( ; k > 0; k /= 3)
     for (int i = l+k; i <= r; i++)
       { int j = i;  int v = a[i];
         while (j >= l+k && v < a[j-k])
           { a[j] = a[j-k]; j -= k; }
         a[j] = v;
       }
 }
```

Merge sort

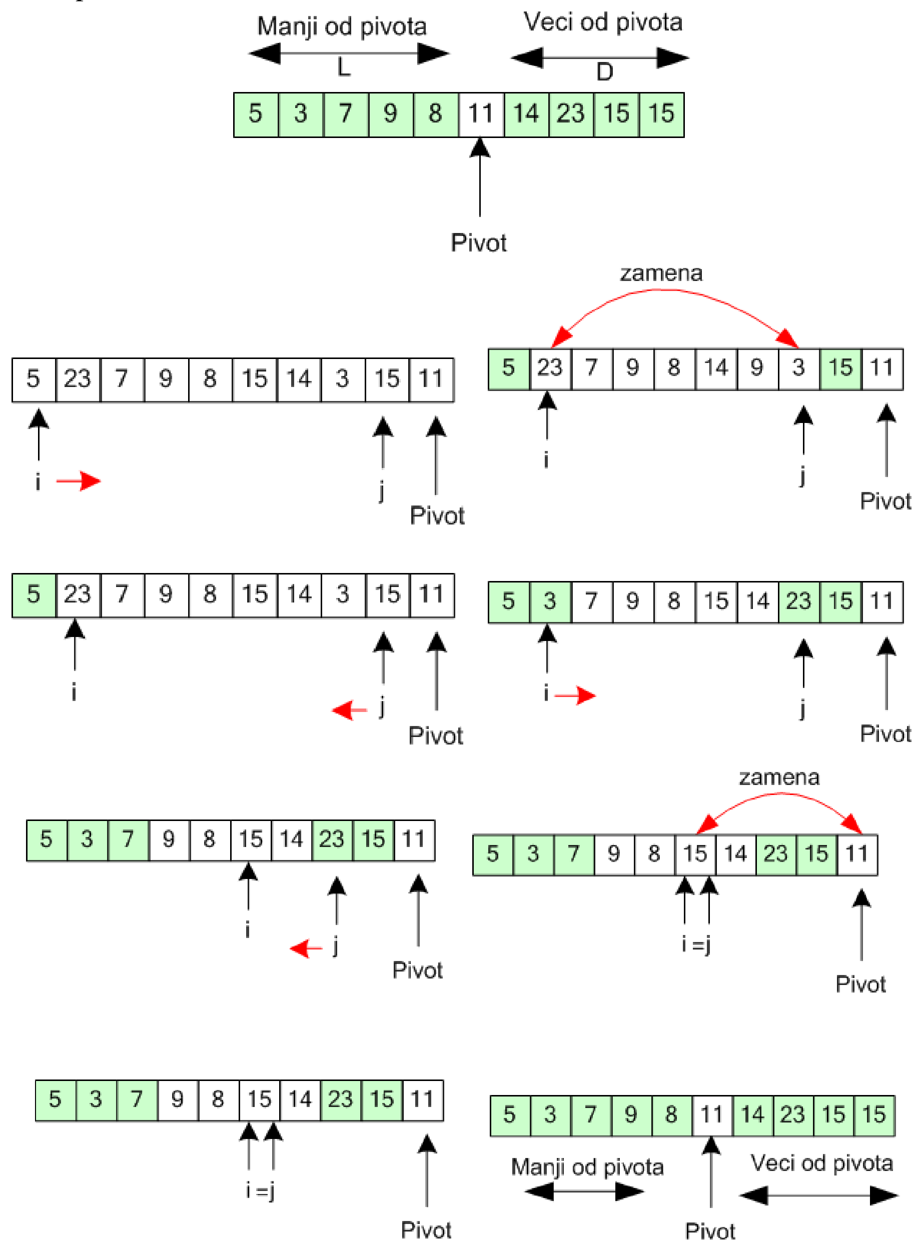
-Pristup "**podeli pa vladaj**" (Divide-and-Conquer)
-Podeli niz na pola u dva podniza
-Podnizovi se sortiraju na isti način rekuzivno
-Spoj podnizove u sortiran niz
-***Kompleksnost O(n log n)***

```
public static void MergeSort(int[] aArray){
		mergeSort(aArray, 0, aArray.length - 1);
}
private static void mergeSort(int[] aArray, int aLeft, int aRight){
	if (aRight > aLeft){
		int middle = (aRight + aLeft) / 2;
		mergeSort(aArray, aLeft, middle);
		mergeSort(aArray, middle + 1, aRight);
		merge(aArray, aLeft, middle + 1, aRight);
	}
}

private static void merge(int[] aArray, int aLeft, int aMiddle, int aRight){
	int i, left_end, num_elements, tmp_pos;
	num_elements = aRight - aLeft + 1;
	int[] temp_array = new int[aArray.length];
	left_end = aMiddle - 1;
	tmp_pos = aLeft;
	while ((aLeft <= left_end) && (aMiddle <= aRight)){
		if (aArray[aLeft] <= aArray[aMiddle]){
			temp_array[tmp_pos] = aArray[aLeft];
			aLeft++;
		}
		else{
			temp_array[tmp_pos] = aArray[aMiddle];
			aMiddle++;
		}
		tmp_pos++;
	}
	while (aLeft <= left_end){
		temp_array[tmp_pos] = aArray[aLeft];
		tmp_pos++;
		aLeft++;
	}
```

```
        while (aMiddle <= aRight){
                temp_array[tmp_pos] = aArray[aMiddle];
                tmp_pos++;
                aMiddle++;
        }
        for (i = 0; i < num_elements; i++){
                aArray[aRight] = temp_array[aRight];
                aRight--;
        }
}
```

Quick sort

-"**Podeli pa vladaj**" strategija **sa sortiranjem u mestu**
 -Podeli niza na dva dela L i D po "***pivotu***"
 -Rekurzivno sortirati podnizove L i D.




```
static void quicksort(int[] a, int l, int r) {
   if (r <= l) return;
   int i = partition(a, l, r);
   quicksort(a, l, i-1);
   quicksort(a, i+1, r);
```

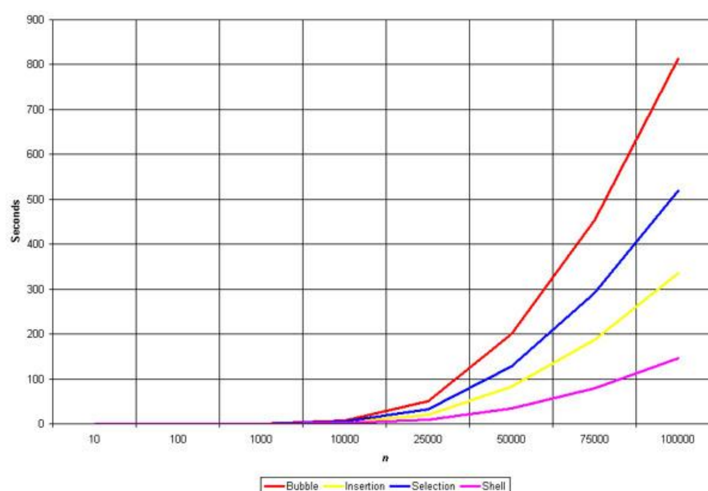
```
}
static int partition(int a[], int l, int r) {
int i = l-1, j = r; int v = a[r];
  for (;;){
     while (a[++i]< v) ;
     while (v < a[--j]) if (j == l) break;
     if (i >= j) break;
     zameni(a, i, j);
    }
  zameni(a, i, r);
  return i;
 }
```

Poređenje algoritama za sortiranje

-O(n^2) algoritmi

-selection sort,
-bubble sort
-insertion sort,
-Shell sort

Vrsta	Prosek	Najgori slučaj
Selection sort	$n^2/2$	$n^2/2$
Bubble sort	$n^2/2$	$n^2/2$
Insertion sort	$n^2/4$	$n^2/2$
Shell sort	-	$\log^2 n$

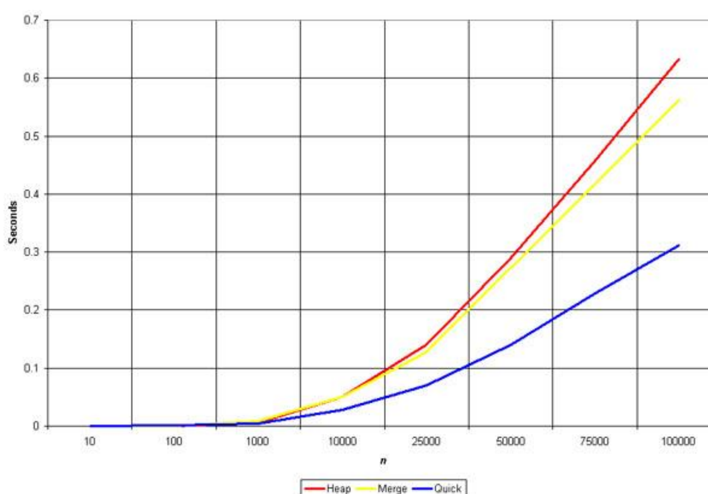


-O(n log n) algoritmi

-merge sort
-heap sort
-qick sort

-O(n log n) je teoretski najbolja moguća prosečna efikasnost algoritma za sortiranje.

Vrsta	Prosek	Najgori slučaj
Merge sort	n log n	n log n
Heap sort	n log n	n log n
Quick sort	n log n	$n^2/2$



STABLA

-***Stabla*** su nelinearne strukture podataka
-Predstavljaju ***najvažnije nelinearne strukture*** koje se vrlo često koriste u računarstvu
-Odnos između elemenata ***nije linearan***
-Imaju razgranatu ili ***hijerarhijsku strukturu elemenata***
-Njihov naziv implicira vezu sa stablima (drvećem) u prirodi ili porodičnim stablima
-Većina terminologije potiče iz ovih izvora

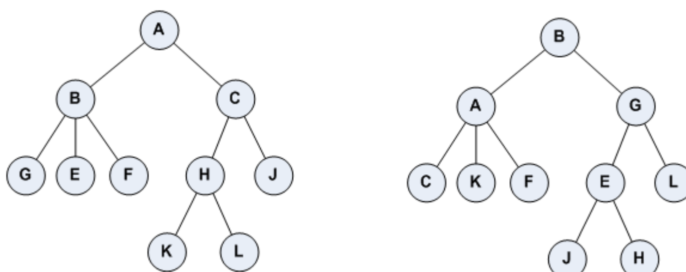
Definicija 1:
-Struktura podataka B=(K,R); K-skup čvorova i R - binarna relacija prethođenja nad skupom K, predstavlja ***stablo*** ako R zadovoljava sledeće uslove:
(1) Postoji samo jedan čvor r, koga nazivamo koren, kome ne prethodi ni jedan drugi čvor
(2) Svaki čvor, izuzev korena, ima samo jednog prethodnika
(3) Za svaki čvor k, k≠r, postoji niz čvorova $k_0, k_1, k_2, \ldots k_n = k$ (n≥1 i=1,n) koji predstavljaju listu, tj. (k_{i-1}, k_i) € R i=1,n

Definicija 2:
-Stablo se može definisati kao poseban oblik ***nelinearnog grafa***:
-***Linearni graf*** je skup čvorova i skup relacija (nazivaju se linije grafa) koji opisuju veze između čvorova. Linearni graf je povezan ako je svaki par čvorova u grafu povezan linijom.
-Stablo se onda može definisati kao ***povezan graf*** koji ne sadrži petlje, odnosno ako važi:
-Bilo koja dva čvora u stablu su povezana linijom
-Stablo sa n čvorova ima n-1 linija

Definicija 3:
-Stablo se može ***rekurzivno*** definisati kao:
-***Stablo*** sa korenom je konačan skup K od jednog ili više čvorova takvih da je:
(1) Postoji jedan specijalni čvor koji se naziva koren stabla
(2) Preostali čvorovi (isključujući koren) se mogu podeliti u m ≥ 0 skupova K_1, K_2, …K_m, čiji je presek prazvan skup, a koji svaki predstavlja stablo za sebe. Stabla K_1, K_2, …K_m se nazivaju podtsbla korena
-Ova rekurzivna definicija je veoma pogodna za predstavljanje stabala u memoriji računara i obavljanje operacija nad stablima

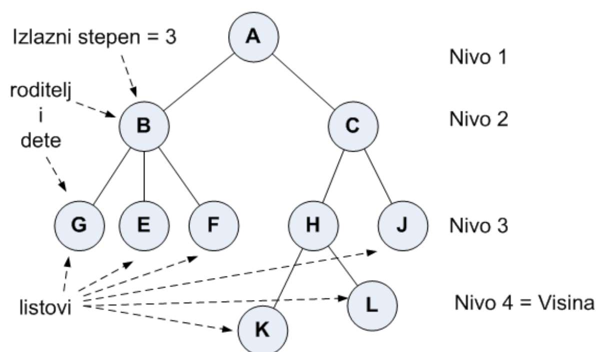
-***Slična stabla***: Za dva stabla se kaže da su slična ako imaju istu strukturu, tj. tačnije ako su oba prazna ili su sva njihova podstabla slična

-***Ekvivalentna stabla***: Dva stabla su ekvivalentna ako su slična i imaju identičan informacioni sadržaj u odgovarajućim čvorovima



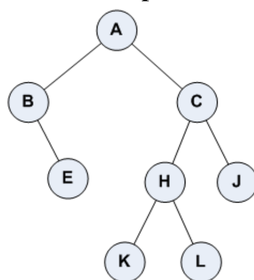
Termini vezani za stabla

-***Terminalni čvor*** ili list je čvor koji nema podstabla
-Čvor d je ***dete*** čvoru k ako je d koren podstabla od čvora k. Čvor k se naziva ***roditelj***.
-***Stepen*** čvora je broj podstabala datog čvora
-***Šuma*** je skup stabala koja ne preklapaju
-***Nivo čvora*** je 1 ako je koren ili jednak broju čvorova koji se prodju na putu od korena do datog čvora.
-***Visina*** (ili dubina) stabla je maksimalni nivo na kome se nalazi neki čvor stabla

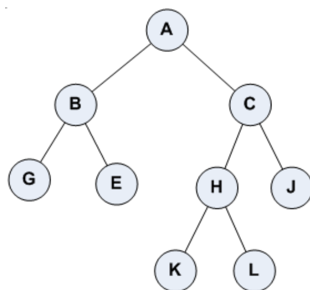


<u>*Binarna stabla*</u>

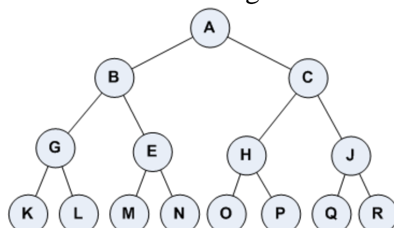
-***Binarno stablo*** se rekurzivno definiše kao konačan skup elemenata koji je ili prazan ili se sastoji od korena i dva binarna podstabla koja se ne preklapaju (tzv. levo i desno podstablo).



-***Striktno binarno stablo*** je binarno stablo kod koga svaki čvor nema nijedno podstabla ima tačno dva

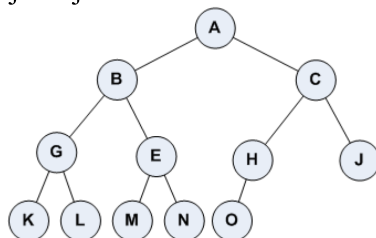


-***Kompletno binarno stablo*** je striktno binarno stablo kod koga su svi listovi na istom nivou



-*Broj čvorova* u kompletnom stablu je: $\mathbf{n = 2^h - 1}$, gde je h visina stabla

-***Skoro kompletno binarno stablo*** je binarno stablo za koga važi da su svi nivoi u stablu popunjeni, osim eventualno zadnjeg, i to tako da se popunjavanje vrši s leva u desno



Prolazi kroz stablo

-Način kako se mogu obići čvorovi datog stabla a da se pri tome posete tačno jednom

-Tri su standardna načina

(1) **Prefiks prolaz (K-L-D)**: Prvo se poseti koren, zatim prefiks prolazom svi čvorovi levog podstabla, a zatim svi čvorovi desnog podstabla

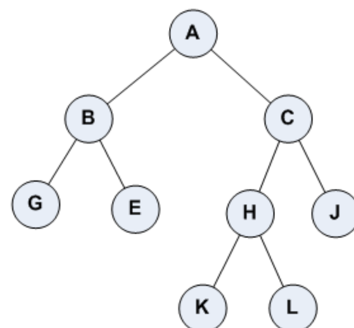
(A, B, G, E, C, H, K, L, J)

(2) **Infiks prolaz (L-K-D)**: Prvo se posete infiks prolazom svi čvorovi levog podstabla,
zatim se poseti koren, a zatim svi čvorovi desnog podstabla

(G, B, E, A, K, H, L, C, J)

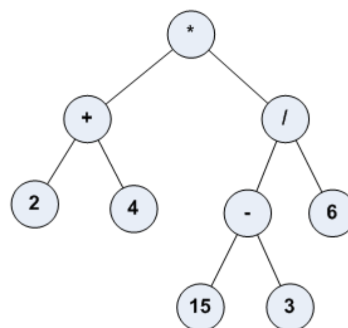
(3) **Postfiks prolaz (L-D-K)**: Prvo se posete postfiks prolazom svi čvorovi levog podstabla,
zatim svi čvorovi desnog podstabla, a zatim se poseti koren

(G, E, B, K, L, H, J, C, A)



Predstavljanje izraza preko stabla

-Matematički izrazi sa binarnim operatorima se mogu predstaviti preko striktnog binarnog stabla i to: listovi su operandi, unutrašnji čvorovi su binarni operatori (operacije).



-Korišćenjem pojedinih prolaza se dobijaju poznate notacije za predstavljanje izraza

-Prefiks prolaz: *, +, 2, 4, /, -, 15, 3, 6 → Prefiks notacija

-Postfiks prolaz: 2, 4, +, 15, 3, -, 6, /, * → Postfiks (Inverzna poljska) notacija

-Infiks prolaz: 2,+,4, *, 15, -, 3, /, 6 → Infiks notacija – uobičajena (Neophodne zagrade zbog prioriteta operatora (2+4)*(15-3)/6)

Definicija stabla kao ATP

-Binarno stablo se može definisati kao **apstraktni tip** na sledeći način:

```
public interface binStablo {
    void ubaci(int a);
    void izbaci(int a);
    int visina();
    void prefiks();
    void postfiks();
    void infiks();
}
```

-Binarna stablo se mogu **implementirati na dva načina**:

(1) ***Dinamička implementacija*** (preko dinamičko spregnute strukture)

-Vrlo fleksibilna implementacija, lako dodavanje i izbacivanje

-Nema ograničenja na broj čvorova i broj nivoa stabla

-Moguća primena za sve slučajeve i oblike binarnih stabala

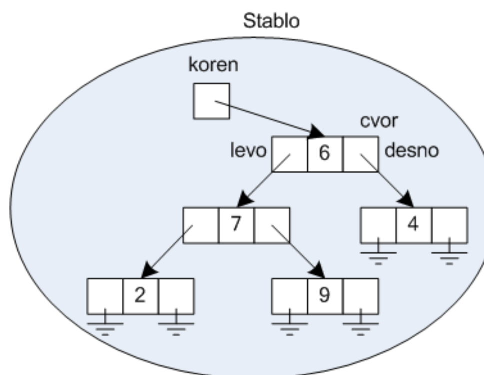
(2) ***Implementacija preko niza***

-Efikasna implementacija moguća samo za specijalni slučaj – skoro kompletno binarno stablo

-Broj čvorova i nivo stabla ograničen dimenzijom niza

Dinamička implementacija

```
public class Stablo implements binStablo {
private class Cvor {
Int info;
Cvor levo;
Cvor desno;
void poseti() {..}
}
        private Cvor koren;
        int visina() {...}
        void prefiks() {...};
        void postfiks() {...};
        void infiks() {...};
}
```

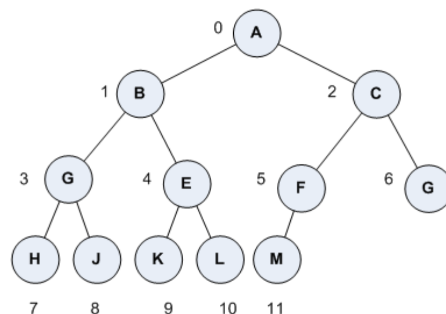



```
public int visina() { return VisinaSt(koren); }
private int visinaSt(Cvor k) {
        if (k == null)
                return 0;
        else
                return  1+ Math.max(visinaSt(k.levo), visinaSt(k.desno)
        }
}
```

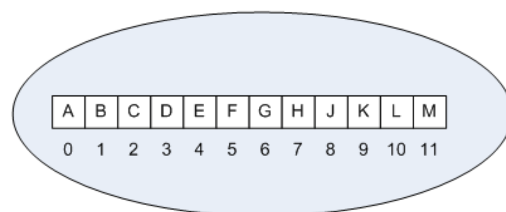
Statička implementacija

-***Efikasna implementacija preko niza moguća samo za skoro kompletno binarno stablo*** (SKBS)
-Kod SKBS je moguće numerisati čvorove da odgovaraju indeksima u nizu i to tako da za svaki čvor važi:

-Indeks levog deteta = Indeks * 2
-Indeks desnog deteta = Indeks * 2 + 1
-Indeks roditelja = Indeks / 2



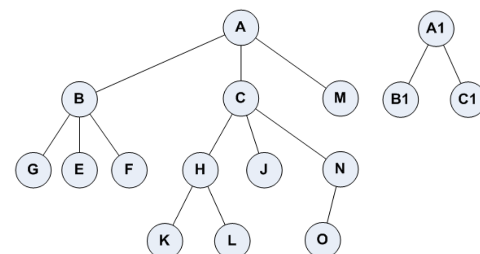

```
public class NizStablo implements binStablo {
private class Cvor {
Int info;
void poseti() {..}
}
        Cvor [] cvorovi;
    int n;
        int visina() {…}
        void prefiks() {…};
        void postfiks() {…};
        void infiks() {…};
}
```

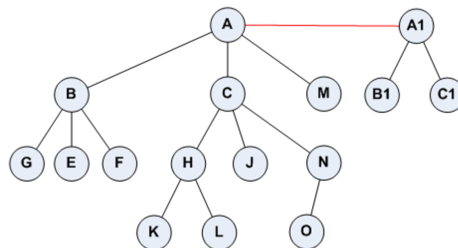


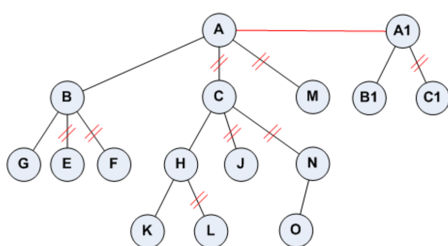

```
public int visina() {
 return  floor (log(n)) + 1;  // n = 2^h-1
}

public void prefiks() {
        if (n > 0)
          return prefiksSt(0); // pocni prefiks prolaz od korena
}
private void prefiksSt(int k) {
        if (k < 0 or k>= n) return;
        cvorovi[k].visit();
        prefiksSt (k*2+1);  // prefiks levo
        prefiksSt (k*2+2); // prefiks desno
}
```

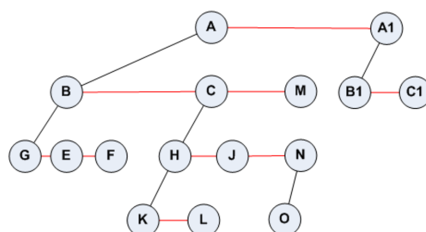
Knutova transformacija
-Transformiše šumu stabala u jedno binarno stablo

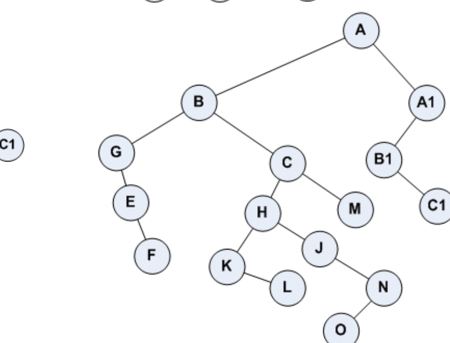
-***Koraci:***
(1) Povežu se koreni svih stabala
(2) Prekinu se sve veze između roditelja i dece izuzev krajnje leve veze
(3) Povežu se sva deca istog roditelja
(4) Dobijena slika se rotira za 45 stepeni udesno











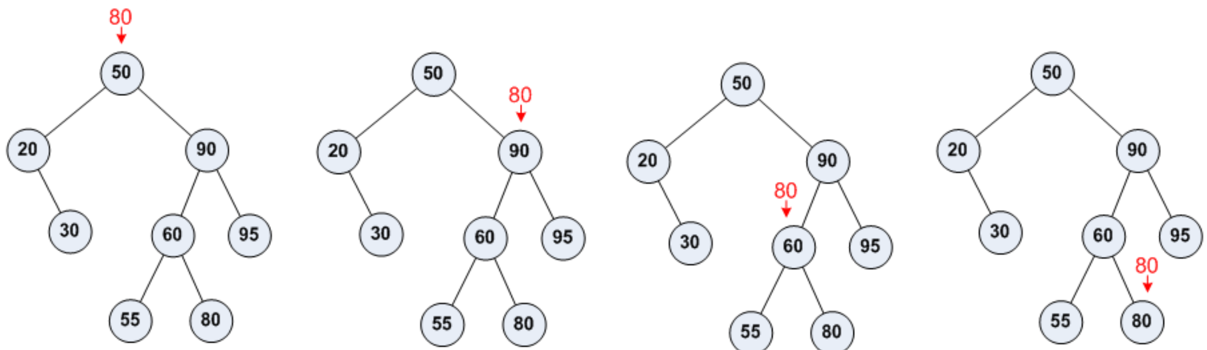
-Rekonstrukcija stabla originalnog stabla
-pokazivač nalevo je pokazivač na prvo dete
-pokazivač na desno je pokazivač na brata

Transformacija u striktno binarno stablo
-Transformiše bilo ***koje stablo*** u striktno binarno stablo
-***Koraci:***
(1) Prekinu se sve veze između roditelja i dece izuzev krajnje leve veze
(2) Povežu se sva deca ista roditelja
(3) Dobijena slika se zarotira za 45st udesno

<u>Stablo za binarno pretraživanje (BST)</u>

-Kod običnog binarnog stabla pretraživanje se zasniva na nekom od prolaza i ima **O(n) efikasnost**.
-BST se zasniva se na ideji da se binarno stablo organizuje tako da omogući pretraživanje slično binarnom pretraživanju niza
-BST je binarno stablo kod koga **za <u>svaki</u> čvor važe sledeća 2 uslova**:
(1) Svi čvorovi ***u levom*** podstablu imaju sadržaj koji je ***manji ili jednak*** sadžaju datog čvora
(2) Svi čvorovi ***u desnom*** podstablu imaju sadržaj koji je ***veći ili jednak*** sadžaju datog čvora

Postupak pretraživanja BST
-Ako je sadžaj korena ***veći*** od traženog ključa onda se pretražuje ***levo*** podstablo rekurzivno
-Ako je sadžaj korena ***manji*** od traženog ključa onda se pretražuje ***desno*** podstablo rekurzivno
-Ako je ***jednak*** po sadržaju onda je ***koren*** traženi čvor



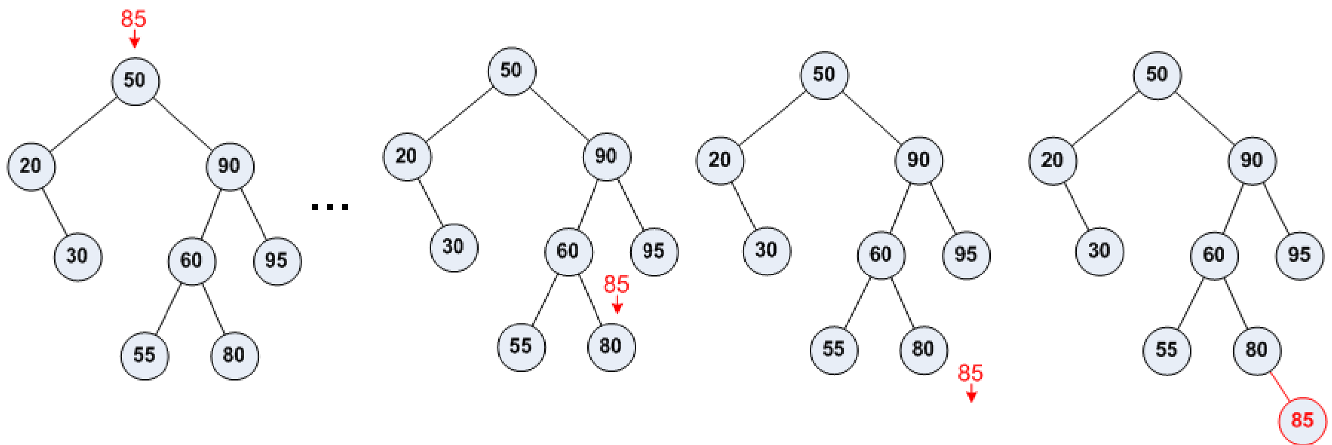

```
public Cvor pretrazi (Cvor k, int v) {
        if (k == null)
                return null;
        if (k.info > v)
                return pretrazi(k.levo, v)
        if (k.info < v)
                return pretrazi(k.desno, v)
  else
          return k;
}
public Cvor pretraziIterativno (Cvor k, int v) {
        while(k != null) {
                if (k.info > v)
                  k = k.levo;
                else if (k.info < v)
                  k = k.desno;
                else if (k.info == v)
                  return k;
   }
        return null;
}
```

Ubacivanje u BST

-Postupak ubacivanja u BST

(1) Prvo se ***pretraži*** BST

(2) Ubaci se novi čvor ***na mesto nula pokazivača*** gde se pretraživanje završilo



Izbacivanje iz BST

-Postupak izbacivanja u BST

(1) Prvo se ***pretraži*** BST da bi se našao traženi čvor za izabacivanje

(2) zatim se ***nadjeni čvor izbucuje*** na nači koji zavisi od njegove pozicije

(3) Moguće su ***tri situacije*** gde se nalazi čvor:

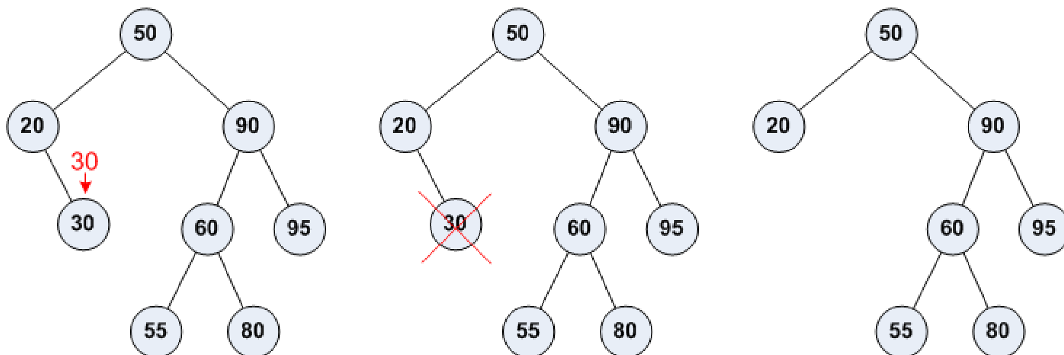
a) *Čvor je list*

b) *Čvor je polu-list* (Ima samo jedno dete)

c) *Čvor je unutrašnji* (Ima oba deteta)

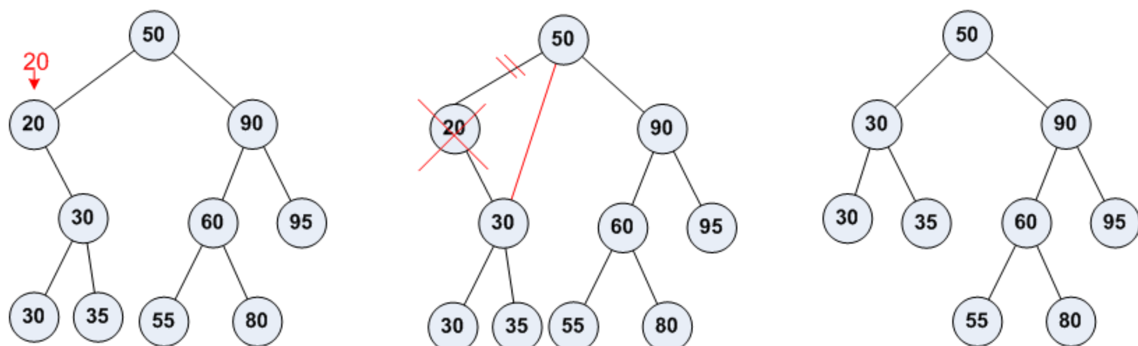
-**Situacija 1: Čvor je list** - prosto se izbaci čvor iz stabla i ažurira pokazivač njegovog roditelja da sadrži <u>null</u> vrednost

-Primer: izbaciti 30



-**Situacija 2: Čvor je polu-list** - izbaci čvor iz stabla, a pokazivač njegovog roditelja se ažurira da sadrži pokazivač na dete čvora koji se izbacuje

-Primer: izbaciti 20



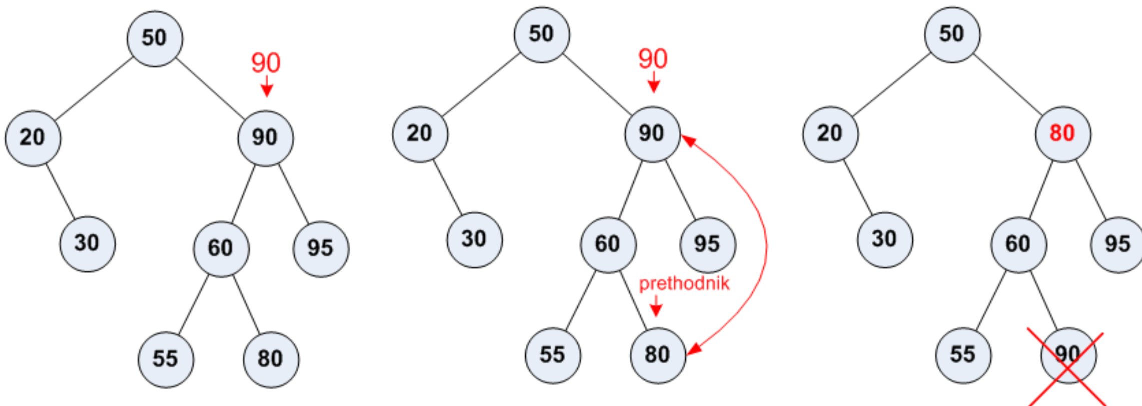
-Situacija 3: Čvor je unutrašnji

(1) Pronađe se njegov prvi sledbenik (ili prethodnik) koji mora biti na krajnjoj levoj (tj. desnoj) poziciji u desnom (tj. levom) podstablu. Sledbenik (tj. prethodnik) mora biti ili list ili polu-list

(2) Zameni se sadržaj sledbenik (tj. prethodnik) sa sadržajem traženog čvora.

(3) Izbacuje se čvor sa pozicije sledbenika (tj. prethodnika). Ovo se svodi na slučajeve 1. i 2. izbacivanja

-Primer: Izbaciti 90



-Ako je stablo **dobro balansirano** ***efikasnost je O(log n)***

-Ako stablo **nije balansirano**, tada je ***efikasnost lošija***.

-***Najgori slučaj je O(n)*** - kada se ubacuje sortirani niz brojeva, BST se degeneriše u listu

Visinski balansirana stablo – AVL

-BST stablo kod koga za svaki čvor važi da se visina njegovog levog i desnog podstabla ne razlikuje za više od 1 se naziva ***visinski balansirano stablo*** ili ***AVL stablo***

-AVL po imenima naučnika ***Addison, Velsky i Landin*** koji su predložili ovakvo stablo

-Rešava problem najgoreg slučaja kod BST

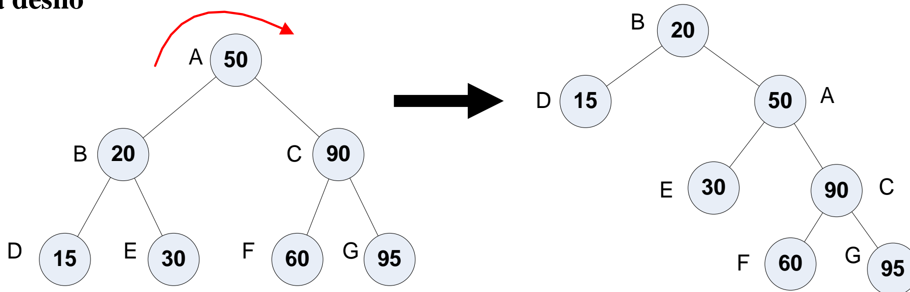
-Pretraživanje garantovano ima ***efikasnost O(log n)***

-U odnosu na BST ima ***modifikovani algoritme*** za ubacivanje i izbacivanje

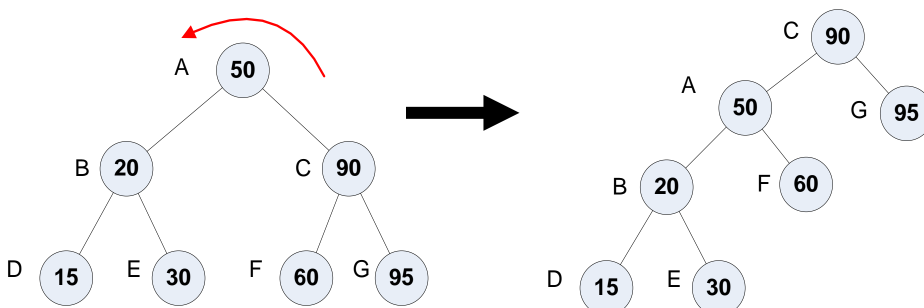
-Zasniva se na rotacijama

-Rotacije mogu biti na levo i na desno

-Rotacije na desno



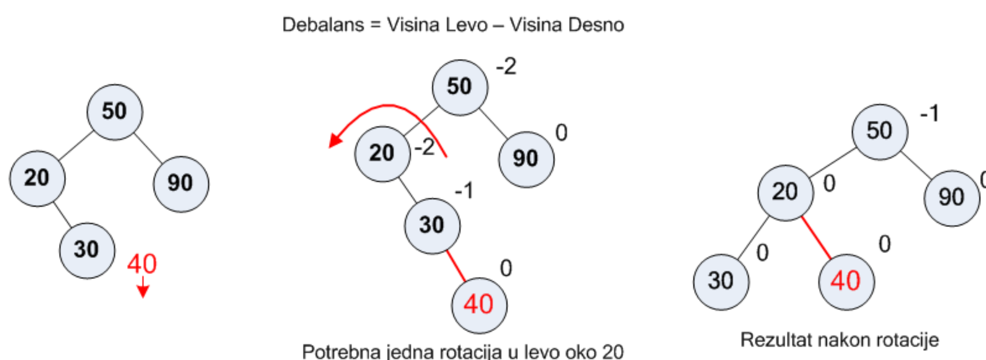
-Rotacije na levo



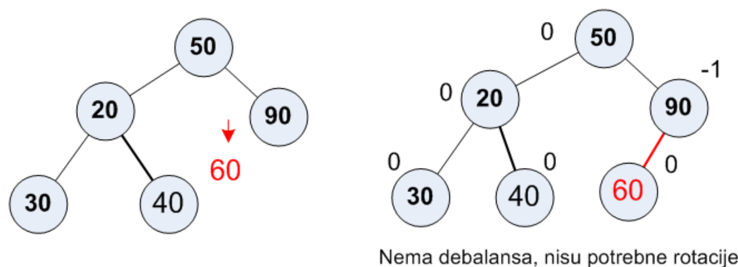
Balansiranje stabla
-Rotacije menjaju visinski balans stabla
   -Rotacija na desno povećava debalans u korist desnog podstabla
   -Rotacija na levo povećava debalans u korist levog podstabla
-Ovo se koristi kod algoritama za ubacivanje i izbacivanje
   -Kada operacija naruši visinski balans, tj. stvori se debalans na jednoj strani, primenjuje se odgovarajuća rotacija da koriguje debalans
   -Rotacija je uvek suprotna debalansu kako bi izjednačila balans. Npr, ako je debalans na desno, koristi se leva rotacija i obrnuto

Ubacivanje u AVL
   (1) Ubacuje se čvor na isti način kao u slučaju BST
   (2) Proveri se za svaki čvor u stablu njegov debalans, tj. razlika visina između levog i desnog podstabla
   (3) Ako postoji debalns, onda se vrši odgovarjuća rotacija oko čvora k koji je najbliži mestu ubacivanja
   (4) Ako je oznaka debalansa deteta čvora k suprotna, onda se pre rotacije iz koraka 3, vrši suprotna rotacija oko tog deteta čvora k. U ovom slučaju su potrebne dve rotacije!
      -DEBALANS = Visina Levo – Visina Desno

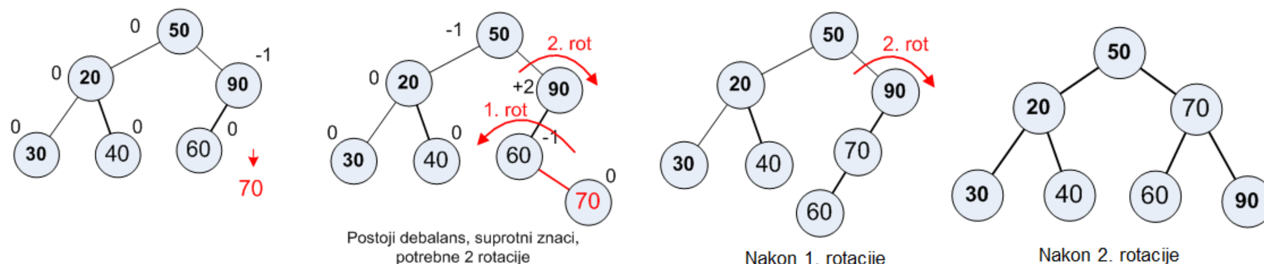
-Ubacuje se broj **40**:



-Ubacuje se broj **60**:

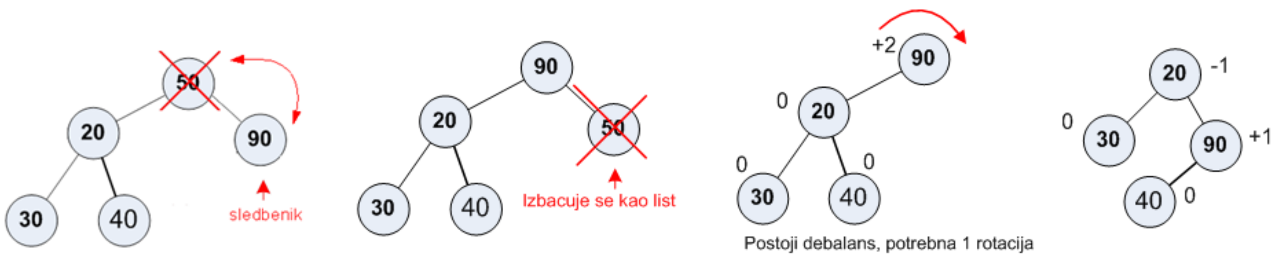


-Ubacuje se broj **70**:



Izbacivanje iz AVL stabla
-Izbacivanje iz AVL stabla se vrši na sledeći način:
   (1) ***Izbaci se čvor na isti način kao kod BST***. 3 slučaja:
      a) Čvor list – samo se izbaci
      b) Čvor polu-list – preveže se dete na roditelja
      c) Čvor unutrašnji – vrši se zamena sa prethodnikom (sledbenikom) i svodi se na prva dva slučaja
   (2) ***Proveri se debalans*** i vrše potrebne (jedna ili dve) rotacije kao i kod ubacivanja

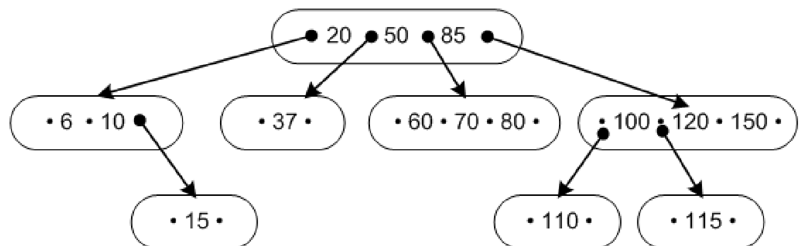
-Izbacuje se broj **50**:



B, B*, B+ stabla

Višegranska stabla
-Uopštenje binarnih stabala pretraživanja
-Postoji ***više ključeva*** u čvoru
-Svakom ključu pridružena ***dva podstabla*** - jedno podstablo je istovremeno levo desno za dva susedna ključa
-Svi ključevi u levom podstablu su manji, a u desnom podstablu veći od datog ključa.
-Red stabla određuje koliko može svaki čvor imati maksimalno ključeva. ***Maximalni broj podstabala je n+1***

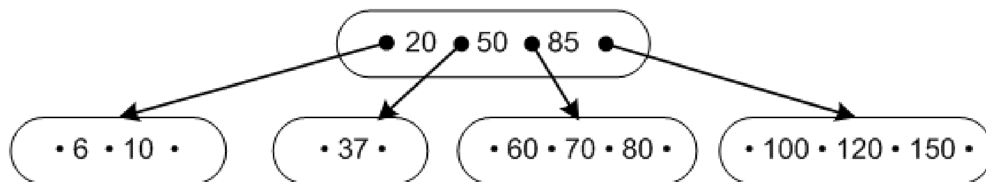
```
TreeNode PretVST(TreeNode Root, int kljuc){
        TreeNode node = RootVST;
        if (node == null){
                Poz = 0;
                return null;
        }
        int i = PretCvor(node, kljuc);
        if (kljuc == k(node, i)){
                Poz = i;
                return node;
        }
        return PretVST(s(node, i), kljuc);
}
```



B-stabla

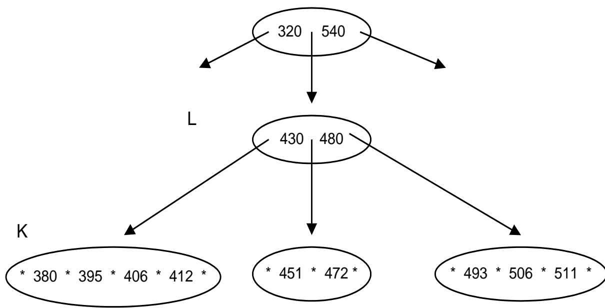
-B-Stablo reda n je **višegransko stablo reda n kod koga važi**:
(1) ***Svi listovi su na istoj visini***
(2) ***Svaki čvor, izuzev korena, ima minimalno n/2 ključeva***

-B-Stablo reda 3:

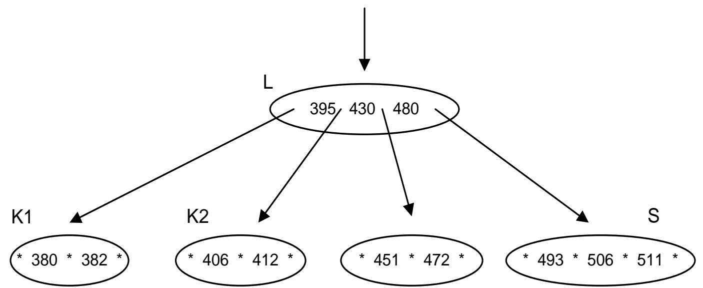


Ubacivanje u B-Stablo
-Pronaći čvor gde se vrši ubacivanje (Koristi se algoritam za pretraživanje)
-Ubaciti novi ključ
(1) Ako **ima mesta**, ključ se prosto ubaci
(2) Ako **nema mesta**, onda:
-Pocepa se dati čvor na dva, i sortirani niz postojećih ključeva (uključujući i novi), osim srednjeg ključa (sredina sortiranog niza) se raspodeljuje u nove čvorove
-Srednji ključ se ubacuje u nadređeni čvor
-Postupak ubacivanja se rekurzivno ponavlja

-Primer ubacivanja **382**



Posle ubacivanja **382** (čvor K se pocepao)



Izbacivanje iz B-Stabla

-**Dva slučaja** izbacivanja:

(1) ***Izbacivanje iz čvora koji nije list***

-Svodi se na izbacivanje iz čvora koji jeste list

-Pronalazi se prethodnik (ili sledbenik) datog ključa

-Prethodnik (sledbenik) se nalazi na krajnjoj desnoj (levoj) poziciji u krajnjem desnom (levom) čvoru u levom (desnom) podstablu datog ključa

(2) ***Izbacivanje iz čvora koji je list***

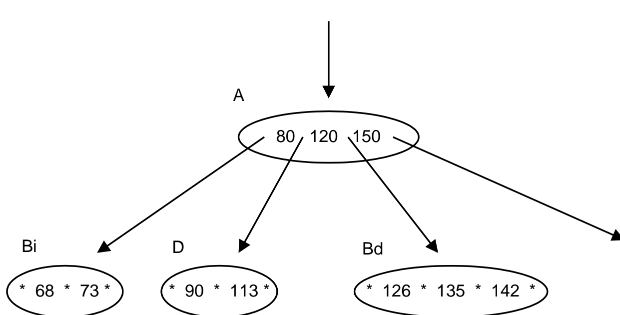
-Problem se javlja ako nakon izbacivanja u listu ostaje manje od n/2 ključeva. U tom slučaju:

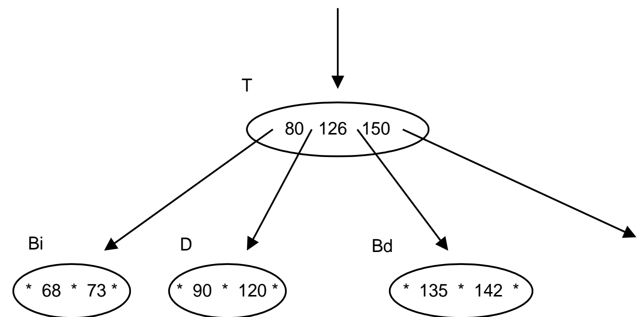
-Pozajmjuje se od levog ili desnog brata ključevi

-Ako nijedan brat nema dovoljno ključeva tada se dati čvor spaja sa nekim od svoje braće. Tom prilikom se spušta njihov nadređeni ključ iz nadređenog čvora

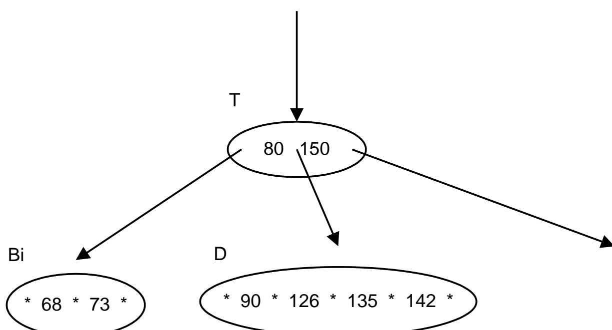
-Ako spuštanje nadređenog ključa narušava integritet onda se postupak izbacivanja rekurzivno ponavlja

-Primer izbacivanja ključa **113**

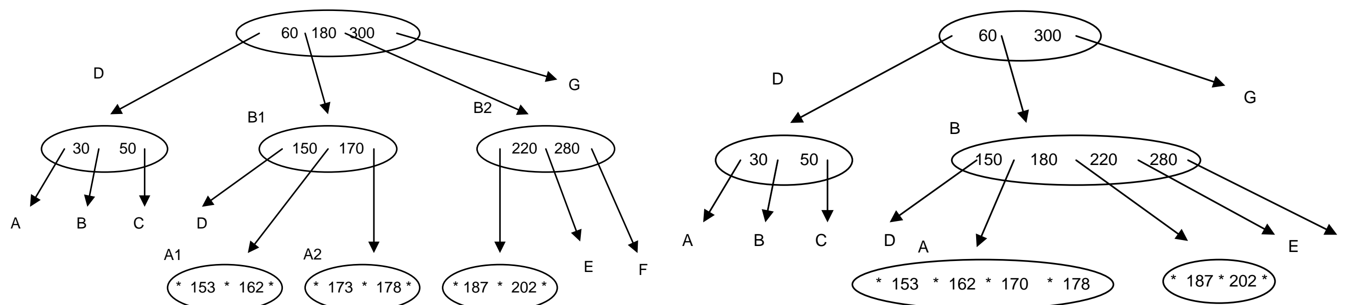




Nakon izbacivanja 120



-Nakon izbacivanja 173



Efikasnost B-stabala

-Neka je red stabla **m-1**
-n je ukupan broj ključeva u stablu.
-Svaki čvor sadrži najmanje
q = (m-1) div 2 ključeva.
-Iz minimalnog broja ključeva pronalazimo da je **maksimalna visina** (d) B-stabla:

$$\mathbf{n = 2(q+1)^{d-1}}.$$

-Logaritmujući levu i desnu stranu dobijamo:

$$\mathbf{\log_{q+1} n = \log_{q+1} 2(q+1)^{d-1}}.$$

-Sređivanjem dobijamo:

$$\mathbf{\log_{q+1} n - \log_{q+1} 2 = (d-1) \log_{q+1}(q+1)},$$

-a iz toga sledi

$$\boxed{\mathbf{d = \log_{q+1}(n/2) + 1}}$$

Visina B-stabla	Minimum		Maksimum	
	čvorova	ključeva	čvorova	ključeva
1	1	1	1	m-1
2	2	2q	m	(m -1)m
3	2(q+1)	2q(q+l)	m^2	(m -l)m^2
⋮	⋮	⋮	⋮	*. ⋮
i	$2(q+1)^{i-2}$	$2q(q+1)^{i-2}$	m^{i-1}	(m -i)m^{i-1}
⋮	⋮	⋮	⋮	⋮
ukupno za visinu d	$\frac{2(q+1)^{d-1}-1}{q}+1$	$2(q+1)^{d-1}$	$\frac{m^d-1}{m-1}$	m^d-1

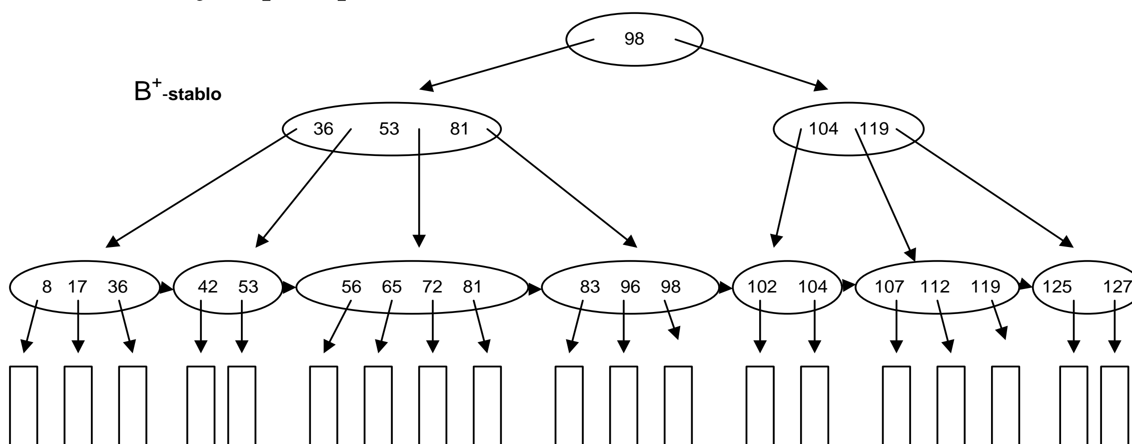
B* stablo

-B* - Poboljšan algoritam ubacivanja

-Umesto da se čvor u kome nema mesta za novi ključ odmah cepa, vrše se prethodno **pozajmica** viška ključeva njegovoj braći

-Tek ako pozajmica nije moguća (ako su braća popunjena) se vrši cepanje

-Omogućava i **sekvencijalni pristup**



Hashing – funkcija transformacije

-Skupovi kao strukture podataka

-Skup kao struktura podataka

(1) Ne postoji uređenje elemenata u skupu, tj.operacije *sledeći*, *prethodni*, *prvi*, *zadnji* nemaju smisla

(2) Samo se zna da je dati element član skupa, tj. *nadjiPoKljuču* jedino ima smisla

(3) Potrebne operacije *ubaci* i *izbaci*

Specifikacija skupa - Definicija preko ATP

```
public interface ISkup
{
  void izbaci(int obj);
  void ubaci(int obj);
  int nadjiPoKljucu(int k);
...
}
```

Implementacija skupa pomoću transformacije ključa u adresu (hashing-a)

-Niz (tabela) može da pamti elemente skupa

-Transformacijom ključa u adresu (indeks) se određuje pozicija elementa u nizu

-Funkcija koja vrši transformaciju se haziva **hash** funkcija, a ceo ovaj pristup **hashing**

-Potrebno je da hash funkcija generiše jedinstvene adrese

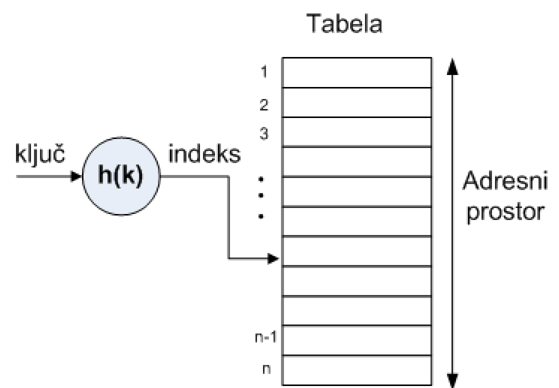
-Prilikom ubacivanje se pomoću hash funkcije odredi adresa i element se smešta na datu adresu

-Prilikom pretraživanja po ključu hash funkcijom se određuje adresa gde se on nalazi

Organizacija zasnovana na hashingu

-Teoretski dovoljan samo 1 pristup

-Efikasnost O(1), potencijalno brže od svih drugih metoda pretraživanja



Problemi hashing organizacije

(1) ***Kako pronaći idealnu hash funkciju h(k)***

-Treba da bude jednoznačna transformacija, tj. bez kolizije (sudaranja) ključeva

-**Za svako k1 ≠ k2 → h(k1) ≠ h(k2)**

(2) ***Ako se dogodi kolizija kako je rešiti***

-Problem ubacivanja: šta uraditi sa elementom koji dobiju adresu koja je već zauzeta

-Problem pretraživanja: kako naći traženi element ako nije na adresi dobijenoj pomoću hash funkcije

Hashing funkcija h(k)

-***Idealna (perfektna) h(k)***

-Moguće je konstruisati samo ako se unapred znaju svi mogući ključevi i adrese

-U praksi ovaj uslov često nije ispunjen

-***Dobre osobine h(k)***

-Da se brzo računa

-Da ima skoro slučajnu (uniformnu) raspodelu adresa, tj. da nema tačke nagomilavanja (sudaranja)

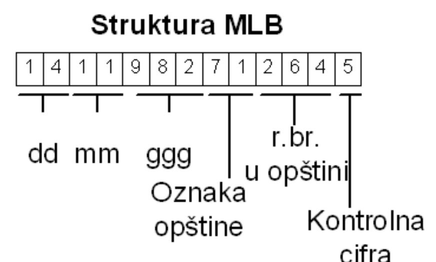
Metode za realizaciju neperfektne h(k)

-Selekcija cifara

-Obično je ključ numerički, pa se mogu neke cifre iz ključa uzeti kao adresa (Važan odabir cifara)

-<u>Npr. (slika)</u> skup od 50.000 studenata. Adresni prostor je 0-50.000, dakle, potrebno je da indeks ima pet cifara.

-Ključ studenta je MLB – 13 cifara. Npr. 1411981712645. Odabrati pet cifara od 13



-Ako se uzmu dd,mm,g svi rodjeni istog dana i meseca u istom veku će imati istu adresu

-Bolje je ako se uzmu 2 cifre opštine i tri cifre rednog broja

-Ostatak od celobrojnog deljenja

h(k) = k mod m = b ; 0<= b <= m

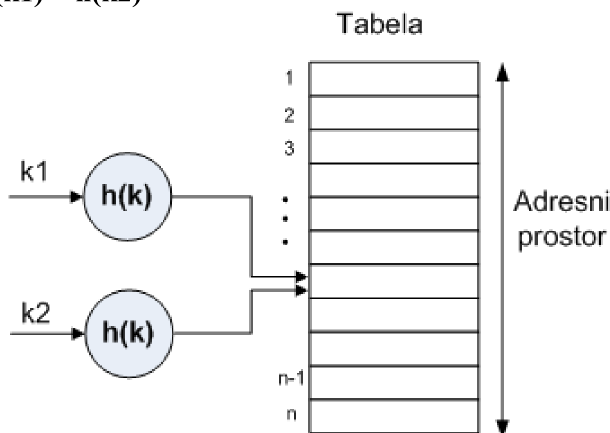
-***Adresni prostor od 0 – m generiše iste adrese ako je k faktor od m*** (Npr. ako je m = 25, svi ključevi koji su deljivi sa 5 će se preslikavati na adrese 0, 5, 10, 15 i 20)

-*Bolje ako k i m nemaju zajednički faktor*. Stoga za m treba uzeti prost broj blizak veličini adresnog prostora (Npr. ako je n = 100, m = 103)

Problem kolizija

-Transformacijom ključa u adresu (indeks) se određuje pozicija elementa u nizu: h(k) = a

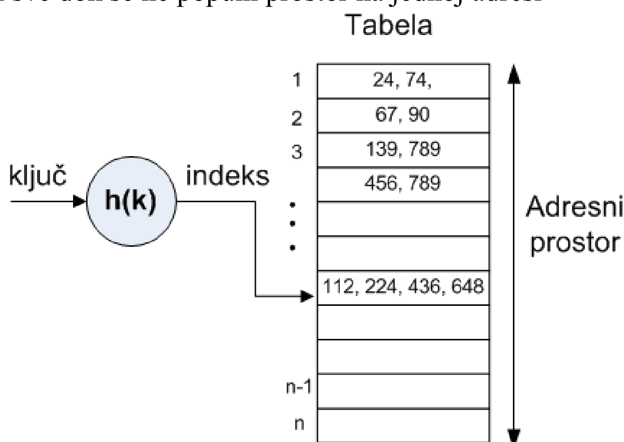
-Problem kolizije: **k1 ≠ k2, h(k1) = h(k2)**



Prihvatanje više zapisa na jednu adresu

-Jedna adresa može da prihvati više

-Problem smeštanja ne postoji sve dok se ne popuni prostor na jednoj adresi



-Ako se dogodi kolizija kako je rešiti

(1) ***Problem ubacivanja***: šta uraditi sa elementom koji dobiju adresu koja je već zauzeta

(2) ***Problem pretraživanja***: kako naći traženi element ako nije na adresi dobijenoj pomoću hash funkcije

-Prihvatanje više zapisa na jednu adresu

-Ublažava se problem kolizije, tj. problem smeštanja

-Dva standardna načina rešavanja kolizije

(1) **Otvoreno adresiranje**

-Otvoreno adresiranje se bazira na ideji da se primeni druga hash funkcija r(k) na adresu dobijenu originalnom h(k)

-Uzastopna primena r(k) dok se ne dobije slobodna adresa (n – veličina tabele):

$a_0 = h(k)$

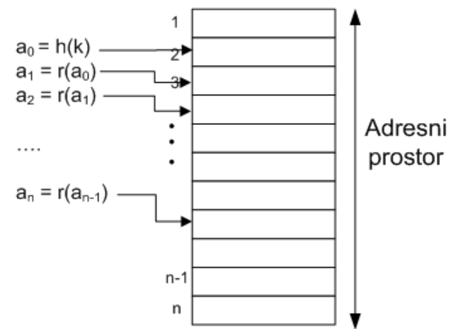
$a_1 = r(a_0)$ mod n //da ne da veću adresu od n

$a_2 = r(a_1)$ mod n

….

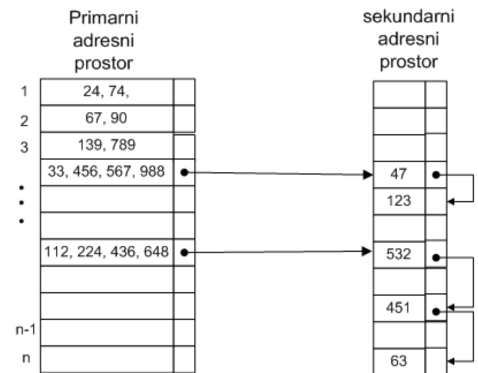
$a_n = r(a_{n-1})$ mod n

a) Linearno probanje
-Problem izbora r(k)
-Uvećati prethodno dobijenu adresu za 1: $\mathbf{r(a_{i+1}) = r(a_i) \bmod n}$



b) Sekundarna kolizija
-Primarna kolizija je kada različiti ključevi imaju istu adresu
-Sekundarna kolizija se javlja kada se sudare ključevi koji imaju različite h(k)
-Otvoreno adresiranje generiše sekundarnu koliziju - Uvećava se broj kolizija

c) Kvadratno probanje
-Problem izbora r(k)
-Sukcesivno uvećati prethodno dobijenu adresu za kvadrat prethodnog rastojanj: $r(a_{i+1}) = r(a_i) \pm j^2$; j=1,2,…n
-Delimično se rešava problem sekundarne kolizije



(2) Metoda olančavanja
-Problem primarne kolizije se rešava uvođenjem posebne zone za zapise koji dobiju istu adresu
-Zapisi koji su u koliziji se olančavaju

Poređenje metoda rešavanja kolizije

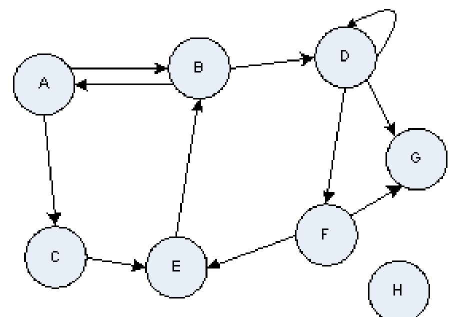
Metoda	Prednost	Nedostatak
Otvoreno adresiranje	Nema dodatnog memorijskog prostora	Problem sekundarne kolizije
Metoda olančavanja	Nema sekundarne kolizije	-Potreban dodatan memorijski prostor -Sekvencijalno pretraživanje jednostruko spregnute liste

Grafovi i mreže

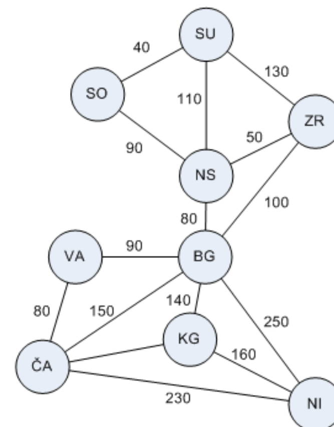
Grafovi su nelinearne strukture podataka
-Predstavljaju najopštije strukture podataka
-Odnos između elemenata nije linearan; jedna element može imati više sledbenika i više prethodnika
-Nema ograničenja u pogledu povezivanja elemenata
-Elementi grafa se nazivaju **čvorovi**
-Veze između čvorova se nazivaju **lukovi**

Definicija orjentisanih grafova
-Struktura podataka B=(K,R); K-skup čvorova i R - binarna relacija nad skupom K i to tako da svaki par čvorova $(k_1, k_2) \in R$ kažemo da obrazuje luk grafa
-Luk $l = (k_1, k_2)$ povezuje čvor k_1 sa čvorom k_2 i to tako da čvor k_1 prethodi čvoru k_2 binarna;

-Za graf kažemo da je ***neorjentisan*** ukoliko važi $(k_1, k_2) \in R \rightarrow (k_2, k_1) \in R$
-Za luk $l = (k_1, k_2)$ kažemo da je k_1 ***početni čvor*** luka l, a za k_2 da je ***završni čvor*** luka (Kaže se i da luk izlazi iz čvora k_1 i da ulazi u čvor k_2).
-Za dva ***čvora*** se kaže da su ***susedni*** ukoliko postoji luk koji ih spaja
-Za dva ***luka*** se kaže da su ***susedni*** ukoliko imaju jedan zajednički čvor
-Čvor koji nema susednih čvorova se naziva ***izolovani čvor***.
-Luk kod koga su identični završni i početni čvor se naziva ***petljom***



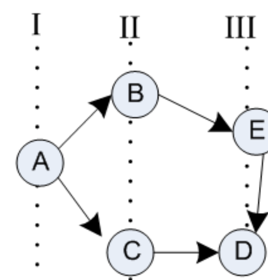
-Lukovi čiji završni čvor jednog je identičan početnom čvoru drugog se naziva ***put grafa***
-Put kod koga su početni čvor prvog luka i završni čvor poslednjeg luka identični se naziva ***ciklus***
-Broj lukova koji obrazuje put se naziva ***dužinom puta*** (Kod petlje dužina puta je 1!)
-Kod neorjentisanog grafa se umesto termina put koristi termin ***lanac***
-Za graf kažemo da je ***povezan*** ukoliko između svaka dva čvora postoji lanac
-Za graf kažemo da je ***stogo povezan*** ukoliko između svaka dva čvora postoji put



<u>Mreže</u>
-Grafovi kod kojih se likovima pridružuje neki podatak se naziva mreža
-<u>Primer:</u> *putna mreža* (neorijentisan graf)

Prolazi kroz graf
-Obilazak svih čvorova grafa tačno jednom
-Potrebni za rešavanje raznih problema:
  -Da li su dva čvora povezana?
  -Nalaženje najkraćeg rastojanja
-Postoji više algoritama za obilazak grafa

(1) Prolaz po širini



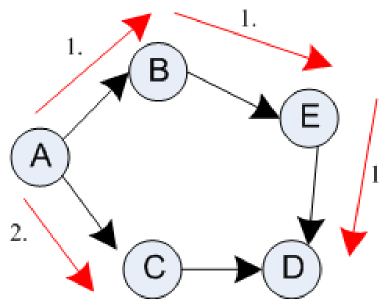
-Čvorovi se obilaze po slojevima (nivoima)
-Kada se jedna čvor obiđe, onda se obilaze svi njegovi susedi, pa tek onda susedi njegovih suseda, itd.
-<u>Primer (slika)</u>: ***A, B, C, E, D***
-Algoritam za prolaz po širini
  -zahteva korišćenje ATP Red (queue)
  -Svaki čvor ima status: Čeka, spreman i obrađen
  -Obilaze se svi čvorovi grafa i ubacuju u red ako već nisu posećeni ili ubačeni u red (tj. one sa statusom da čekaju)
  -Uzima čvorove iz reda, posećuje ih i menja status da su posećeni, a zatim ubacuje sve njihove susede u red ako već nisu posećeni ili ubačeni u red i menja im status u speman (tj. da su ubačeni u red)

```
public void prolazPoSirini(Graf g){
        // inicijalizacija
        for_each (cvor c u grafu g) {
                c.status = čeka;
        }
        // glavna obrada
         for_each (cvor c u grafu g) {
                if (c.status == čeka)
                        posetiCvor(c);
        }
}

private void posetiCvor (Cvor c) {
        static Red r = new Red();  // red u kome se ubacuju cvorovi grafa

        r. Enqueue(c); // ubaci cvor u red
        while (!r.Empty()) {
                k = r. Dequeue();  // uzmi cvor iz reda
                k.visit(); // poseti ga
                k.status == obradjen;  // promeni mu status
                // ubaci u red sve susede od k koji čekaju
                for_each (cvor s koji je sused cvora k) {
                        if (s.status = čeka)
                                r. Enqueue(s);
                                s.status == spreman;
                }
        }
}
```

(2) Prolaz po dubini

-Kada se jedna čvor obiđe, onda se obilazi <u>jedan</u> njegov sused, pa tek kad se on i njegovi susedi obiđu rekurzivno po dubini, prelazi se na obilazak ostalih suseda

-<u>Primer (slika)</u>: ***A, B, E, D, C***



-Algoritam za prolaz po dubini

-zahteva korišćenje ATP Stek (stack)

-Svaki čvor ima status: Čeka, spreman i obrađen

-Obilaze se svi čvorovi grafa i ubacuju u stak ako već nisu posećeni ili ubačeni u red (tj. one sa statusom da čekaju)

-Uzima čvorove iz staka, posećuje ih i menja status da su posećeni, a zatim ubacuje sve njihove susede u stak ako već nisu posećeni ili ubačeni u stak i menja im status u speman (tj. da su ubačeni u red)

```
public void prolazPoDubini(Graf g){
        // inicijalizacija
        for_each (cvor c u grafu g) {
                c.status = čeka;
        }
        // glavna obrada
         for_each (cvor c u grafu g) {
                if (c.status == čeka)
                        posetiCvor(c);
        }
}

private void posetiCvor(Cvor c) {
        static Red s = new Stak();  // stak u kome se ubacuju cvorovi grafa

        s.Push(c); // ubaci cvor u stak
        while (!sEmpty()) {
                k = r.Pop();  // uzmi cvor iz staka
                k.visit(); // poseti ga
                k.status == obradjen;  // promeni mu status
                // ubaci u stak sve susede od k koji čekaju
                for_each (cvor q koji je sused cvora k) {
                        if (q.status = čeka) {
                                q.Push(s);
                                // označi da je spreman, tj. da je ubacen u stak
                                q.status == spreman;
                        }
                }
}
```

Implementacija grafova

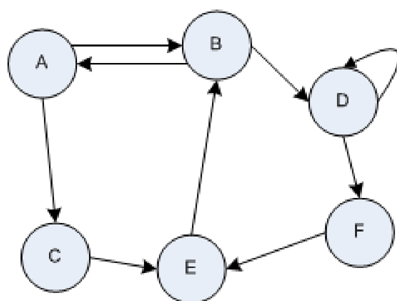
(1) Matrice susedstva (statička implementacija)

-Matrica susedstva grafa G sa n čvorova je kvadratna matrica A [i, j] reda n čiji elementi:

a) 1, ako je čvor i susedan čvoru j

b) 0, ako čvor i i j nisu susedni

-Ako je graf neorijentisan tada je njegova matrica susedstva simetrična: $\mathbf{A} = \mathbf{A}^{\mathbf{T}}$



	A	B	C	D	E	F
A	0	1	1	0	0	0
B	1	0	0	1	0	0
C	0	0	0	0	1	0
D	0	0	0	1	0	1
E	0	1	0	0	0	0
F	0	0	0	0	1	0

-Implementacija matrice susedstva

-Matrica je dvodimenzioni niz

-Prostorna kompleksnost $O(n^2)$

-Ako je broj čvorova grafa veliki, a sadrži mali broj lukova, onda se dobija tzv. <u>retko posednuta</u> matrica, tj. matrica sa mnogo 0

-Neefikasno korišćenje memorije

-Koristi se bit-matrice

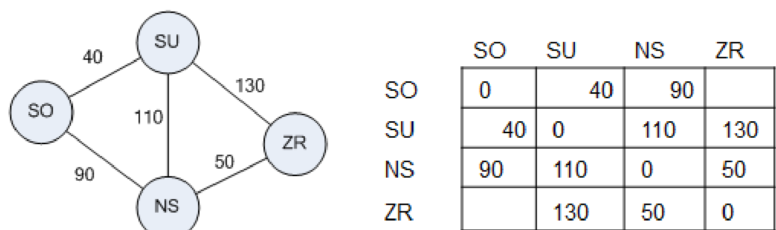
-Za pamćenje elementa Aij se koristi samo jedan bit

-Potrebno je manipulisati bitovima

-Predstavljanje mreže

-Matrica susedstva se može iskoristiti za predstavljanje mreža

-Elementi a_{ij} matrice A sadrže vrednost luka



	SO	SU	NS	ZR
SO	0	40	90	
SU	40	0	110	130
NS	90	110	0	50
ZR		130	50	0

(2) Liste susedstva (dinamička implementacija)

-Lista susedstva je multi-lista – tj. lista listi, tj. lista čiji elementi su liste

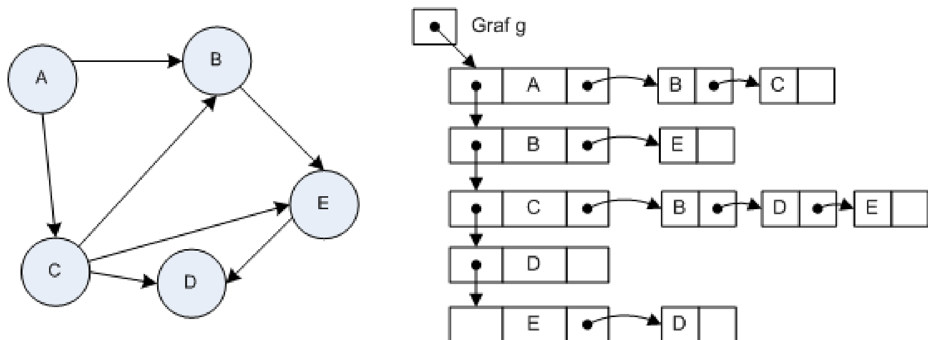
-Lista čvorova grafa od kojih svaki sadrži listu svojih suseda

-Prostorna kompleksnost O(n)

-Za broj čvorova O(n)

-Za broj lukova O(n)

-Efikasno korišćenje memorije



-Poređenje metoda:

(1) ***Matrice susedstva***

-Jednostavnije za manipulaciju

-Prostorna kompleksnost $\boldsymbol{O(n^2)}$

-Neefikasno korišćenje memorije kod retko posednutih matrica

(2) ***Liste susedstva***

-Složenije za manipulaciju

-Prostorna kompleksnost $\boldsymbol{O(n)}$

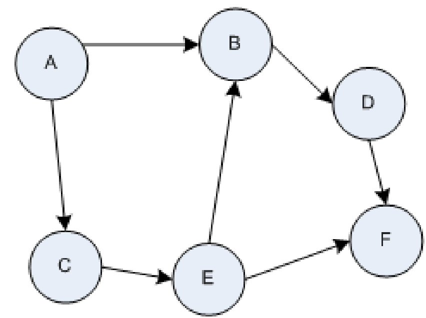
-Efikasnije korišćenje memorije

Algoritmi nad grafovima

(1) Topologijski sort

-Linearno uređenje čvorova grafa tako da nijedan čvor i koji prethodi nekom čvoru j ne bude u lineranoj listi posle čvora j

-Primenljivo samo na orijentisane aciklične grafove



-Topologijski sort:
(1) Nadji čvor bez sledbenika
(2) Ubaci ga na listu
(3) Izbaci iz grafa nađeni čvor i sve njegove lukove
(4) Ponovi korake 1-3 sve dok graf ne ostane prazan

-<u>Primer (slika)</u>: Toplogijski sort: **A, C, E, B, D, F**

-Topologijski sort zahteva dve osnovne procedure:
-Nalaženje čvora koji nema sledbenika
-Izbacivanje čvora i svih njegovih lukova

```
public int NoSuccessors() {
        bool isEdge;
        for(int row = 0; row <= numVertices-1; row++) {
                isEdge = false;
                for(int col = 0 col <= numVertices-1; col++)
                        if (adjMatrix[row, col] > 0) {
                                isEdge = true;
                                break;
                        }
                }
                if (!(isEdge))  return row;
        }
        return -1;
}
public void DelVertex(int vert) {
        if (vert ! = numVertices-1) {
                for(int j = vert; j <= numVertices-1; j++)
                        vertices[j] = vertices[j+1];
                for(int row = vert; row <= numVertices-1; row++)
                        moveRow[row, numVertices];
                for(int col = vert; col <= numVertices-1; col++)
                        moveCol[row, numVertices-1];
        }
}
private void MoveRow(int row, int length) {
        for(int col = 0; col <= length-1; col++)
                adjMatrix[row, col] = adjMatrix[row+1, col];
}
private void MoveCol(int col, int length) {
        for(int row = 0; row <= length-1; row++)
                adjMatrix[row, col] = adjMatrix[row, col+1];
}
public void TopSort() {
        int origVerts = numVertices;
        while(numVertices > 0) {
                int currVertex = noSuccessors();
                if (currVertex == -1) {
                        Console.WriteLine("Error: graph has cycles.");
                return;
                }
                gStack.Push(vertices[currVertex].label);
                DelVertex(currVertex);
        }
        Console.Write("Topological sorting order: ");
        while (gStack.Count > 0)
                Console.Write(gStack.Pop() + " ");
}
```

(2) Tranzitivni zatvarač

-Graf T je tranzitivni zatvarač grafa G akko postoji luk (i,j) u grafu T ako postoji puitanja bilo koje dužine u grafu G između čvorova i i j.

-P – matrica susedstva ---- Putanje duzine 1

-P^2=PxP ---- Putanje dužine 2

-P^n=PxPx ..P ---- Putanje dužine n

-Algoritam koji stepenuje matricu susedstva na n-ti stepen nije efikasan

-**Warshall-ov algoritam**

-Pk(i,j) = 1 ako postoji putanja od i do j koja prolazi kroz čvorove koji su numerisani do k

-Pk+1(i,j) = 1 akko

-P(i,j) = 1

-Pk(i,k+1) and Pk(k+1,j)

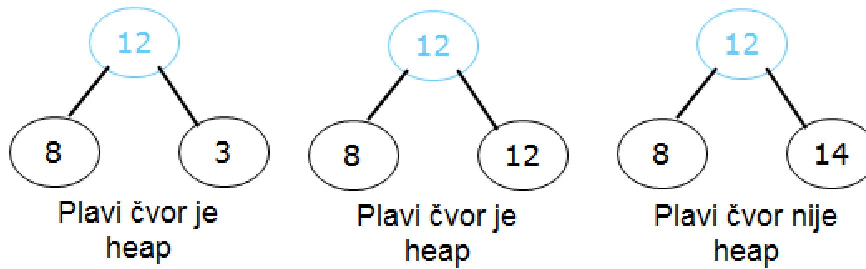
-Efikasnost je $O(n^3)$

```
void Warshal() {
  for(int k = 0; k<= numVertices-1; k++) {
     for(int i = 0; i<= numVertices-1; i++) {
        if (adjMatrix[i, k]) {
           for(int j= 0; j<= numVertices-1; j++)
            adjMatrix[i, j] = adjMatrix[i, k] || adjMatrix[i, k]
        }
     }
  }
}
```

Heap sort

-Efikasnost algoritma je ***O(n log n)***

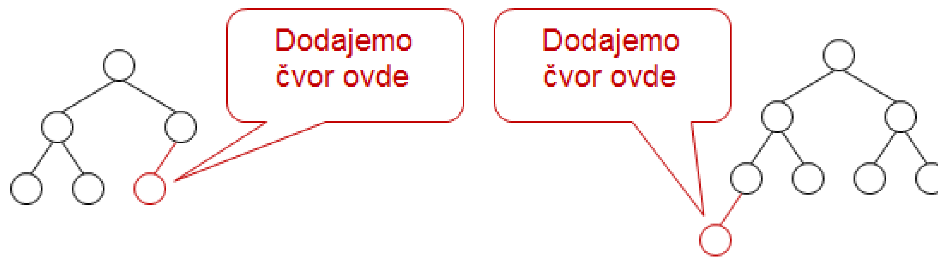
-Quicksort algoritam ima prosečnu efikasnot ***O(n log n)*** ali je najgori slučaj mnogo lošiji ***O(n²)***



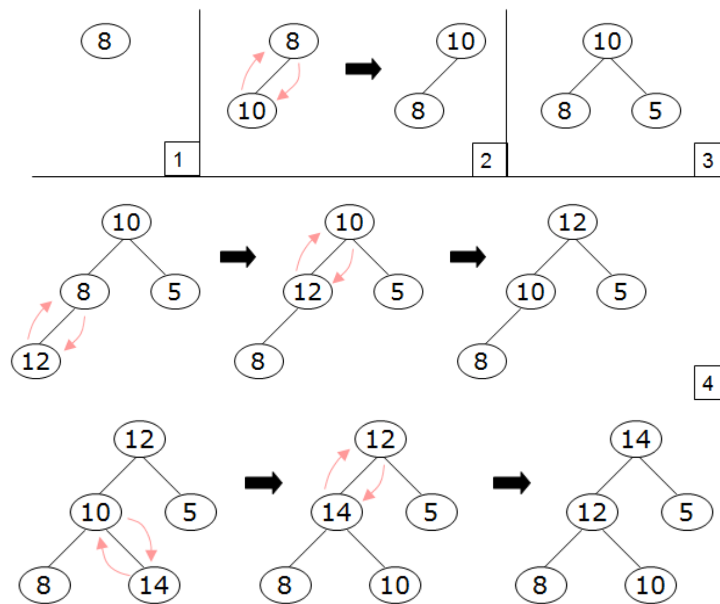
-Listovi su heap čvorovi

-Binarno stablo je heap ako su svi čvorovi heap

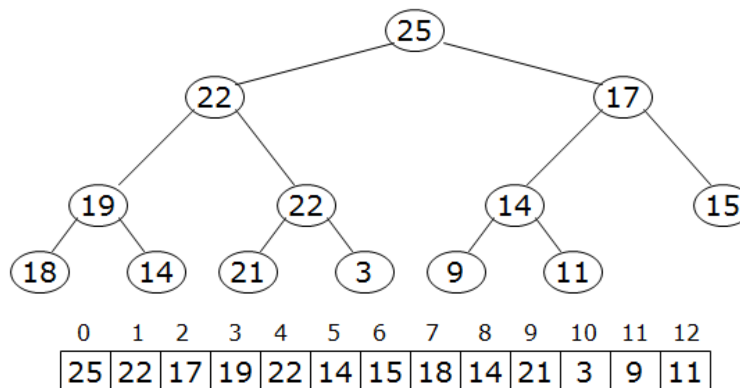
-Dodavanje čvora



-Konstrukcija stabla



-Primer preko niza

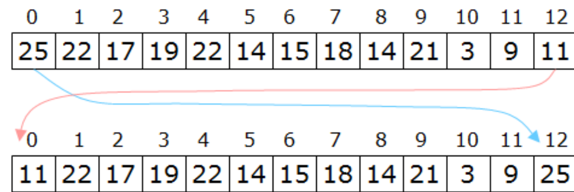


0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

Levo dete: 2*i+1

Desno dete: 2*i+2

-Najveći element odlazi na kraj, a nakon toga se kreira stablo od (dužina niza - broj sortiranih elemenata) elemenata



-I tako se ponavlja dok se se sortira ceo niz