# (Predavanje 8) Stabla

Stablo je struktura podataka kojom se modeliraju hijerarhijski organizacioni odnosi između objekata koji čine stablo.

Takvi odnosi se susreću u različitim područjima.

Na primjer, pomoću stabla se mogu prikazati organizacijski odnosi u nekoj kompaniji, rodbinski odnosi u nekoj porodici, organizacija različitih dijelova neke vojske, sadržaj neke knjige, itd.

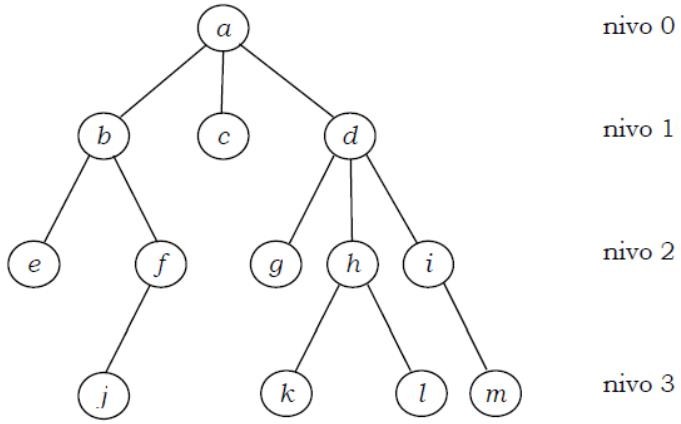
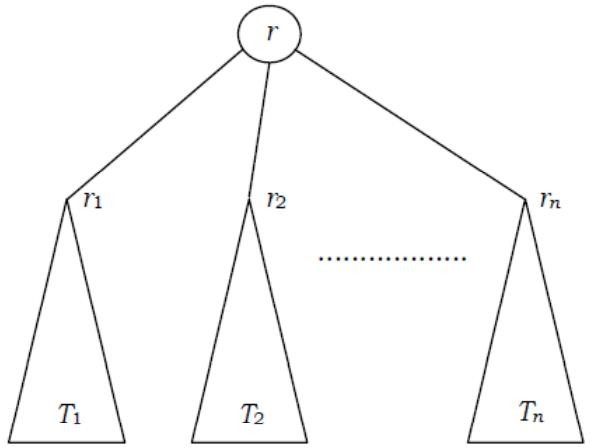
## Definicija stabla i terminologija:

Stablo T je konačan skup elemenata istog tipa koji se zovu čvorovi

Jedan je čvor istaknut i naziva se korijen stabla ostali čvorovi su podijeljeni u n disjunktnih skupova T1, T2, ...Tn, n≥0 – tako da svaki skup također predstavlja stablo.

Korijeni r1, r2, ...rn od stabala T1, T2, ...Tn su djeca od čvora r, a čvor r je njihov roditelj.

Stabla Ti, 1≤ i ≤n, se nazivaju podstabla korijena r.

Korijen r nema roditelja, dok svi ostali čvorovi imaju tačno jednog roditelja. Svaki čvor u stablu se može promatrati kao korijen nekog podstabla

Na slici je prikazano stablo T koje ima 13 čvorova T={a, b, c, d, e, f, g, h, i, j, k, l, m}, pri čemu je čvor a korijen stabla, a ostali čvorovi su podijeljeni u tri disjunktna skupa T1={b, e, f, j}, T2={c}, T3={d, g, h, i, k, l, m}. Svaki od prethodno navedena tri skupa je također stablo, jer zadovoljava oba uvjeta navedena pri definiranju stabla.

Korijen a ima tri podstabla, čiji su korijeni čvorovi b, c i d.

Ako pretpostavimo da su grane koje spajaju čvorove stabla usmjerene od korijena prema stablu, onda možemo definirati pojmove ulaznog i izlaznog stupnja čvora.

Ulazni stupanj predstavlja broj grana koje ulaze u čvor, dok je izlazni stupanj broj grana koje izlaze iz čvora.

To znači da korijen a ima ulazni stupanj 0, a svi ostali čvorovi imaju ulazni stupanj 1.

Za razliku od ulaznog stupnja, koji može imati samo dvije vrijednosti, izlazni stupanj čvora može imati različite vrijednosti za svaki pojedini čvor.

Stupanj stabla se definira kao najveći izlazni stupanj nekog od čvorova, koji čine to stablo.

Za primjer stabla na slici , stupanj stabla je 3, jer čvorovi a i d imaju maksimalni izlazni stupanj, a to je 3.

Čvorovi sa nultim izlaznim stupnjem se nazivaju terminalni čvorovi ili listovi, dok se ostali čvorovi stabla nazivaju neterminalni čvorovi.

Terminalni čvorovi stabla prikazanog na slici su: e, j, c, g, k, l i m.

Stablo je struktura podataka koja može sadržavati čvorove na različitim nivoima. Pod nivoom nekog čvora podrazumijevamo broj grana koje se nalaze na putu od korijena do dotičnog čvora.

Na primjer, čvor a je na nivou 0, dok je čvor f na nivou 2 (slika). Maksimalna vrijednost nivoa nekog od listova u stablu se naziva visina stabla. Većina algoritama u radu sa stablima svoje operacije započinju u korijenu r.

Definirajmo pojam puta (n1,...,nk) kao niz čvorova kod kojeg je čvor ni roditelj od čvora ni+1, 1<=i<=k. Dužina puta od čvora n1 do čvora nk je definirana brojem grana na tom putu, a to je za jedan manje od broja čvorova na putu.

Do svih čvorova u stablu se može doći slijedeći put od korijena, a taj put je jedinstven za svaki čvor

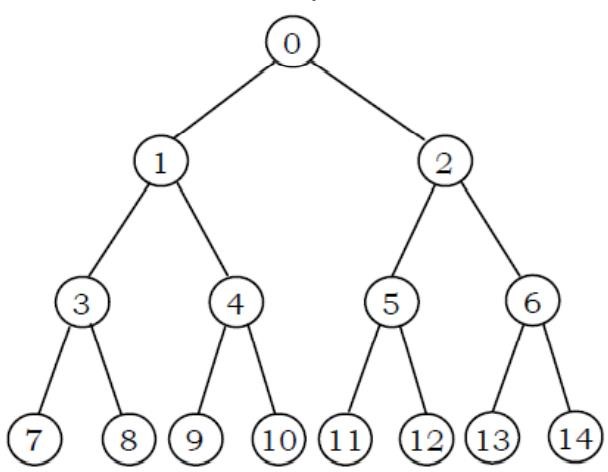
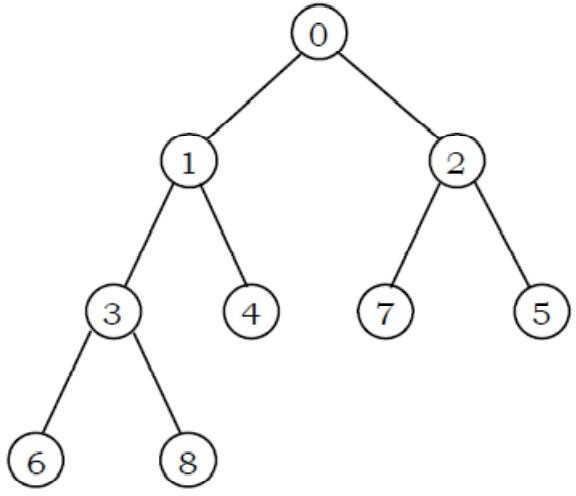
# Binarna stabla

## Definicija stabla i terminologija:

Binarno stablo je stablo kod kojeg svaki roditelj ima najviše dvoje djece. Svako dijete u binarnom stablu je označeno ili kao lijevo dijete ili kao desno dijete, pa svaki roditelj ima najviše jedno lijevo dijete i jedno desno dijete.

Najčešće korištena vrsta stabala su upravo binarna stabla.

Puno binarno stablo je binarno stablo kod kojeg svaki roditelj ima tačno dvoje djece. Primjer punog

binarnog stabla je prikazan na slici 1

Slika 1 Slika 2

Puno binarno stablo kod kojeg su svi listovi na istom nivou se naziva kompletno binarno stablo.

Primjer kompletnog binarnog stabla je prikazan na slici 2

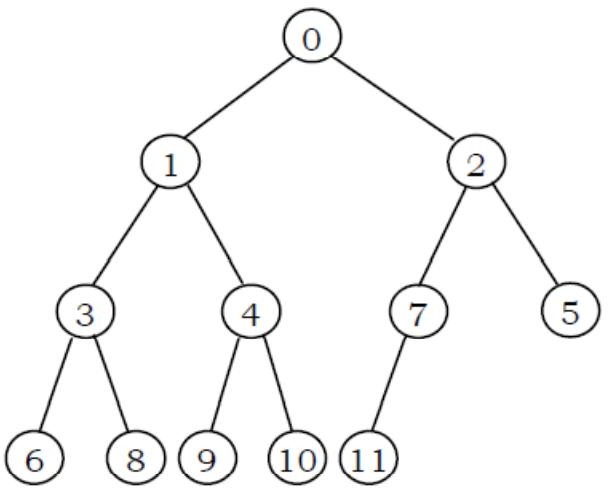
Na i-tom nivou kompletnog binarnog stabla se nalazi 2 𝑖 čvorova. Prema tome, ukupan broj čvorova

n u kompletnom binarnom stablu visine h je



Ako binarno stablo na zadnjem nivou nije u cjelosti popunjeno, već je stablo slijedno popunjeno počevši od krajnjeg lijevog lista, tada takvo stablo nazivamo skoro kompletno stablo.

Dakle, kod skoro kompletnog stabla broj čvorova na zadnjem nivou je manji od 2 ℎ .

Skoro kompletno stablo ne mora biti istovremeno i puno stablo. Na primjer, stablo na slici

Dalje će biti detaljnije opisana jedna posebna vrsta binarnog stabla koja je izuzetno važna u mnogim

primjenama.

Ta vrsta binarnog stabla se naziva binarno stablo pretraživanja.

Čvorovi binarnih stabala pretraživanja sadrže podatke, pri čemu se obično koristi specifično polje koje ima jedinstvenu vrijednost za čitavo stablo. To polje se naziva ključ i na temelju njega se definira organizacija stabla.

Na ovom mjestu ćemo razmatrati samo binarna stabla pretraživanja, kod kojih su svi ključevi u stablu različiti, jer se u većini stvarnih primjena najčešće izbjegava korištenje više istih vrijednosti u stablu, iako se u nekim specifičnim primjenama dozvoljava da više ključeva ima istu vrijednost

Binarno stablo pretraživanja za svaki čvor n u kojem je pohranjena vrijednost ključa k ima sljedeće

osobine:

sve vrijednosti smještene u njegovom lijevom podstablu su manje od neke vrijednosti k;

sve vrijednosti pohranjene u desnom podstablu su veće od k.

Značenje termina „manje od“ i „veće od“ ovisi o tipu vrijednosti pohranjenih u stablu.

Na primjer, kod znakovnih vrijednosti značenje navedenih termina možemo temeljiti na vrijednosti

ASCII koda pojedinog znaka

## Implementacija binarnog stabla

Binarna stabla se mogu implementirati na različite načine. Najjednostavniji način implementacije binarnog stabla je sekvencijalna implementacija pomoću nizova.

Pri implementaciji stabla pomoću niza na prvom mjestu se nalazi korijen. Svaki element, pored informacionog sadržaja, umjesto pokazivača na djecu sadrži indekse elemenata koji odgovaraju djeci datog čvora.

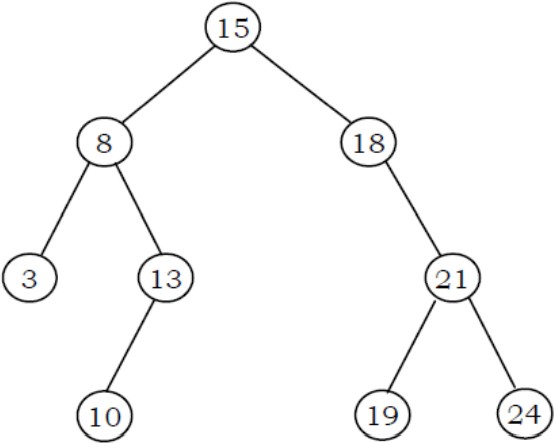
Vrijednost 0, slično kao i prazni pokazivači, označava da dotični čvor nema dijete

Na primjer, stablo na slici se može prikazati kao što je to prikazano u tabeli na slijedećem slajdu.

Na primjer, za čvor 15, djeca se nalaze na pozicijama 8 i 5. Ili, za čvor 21, djeca se nalaze na

pozicijama 4 i 2.

Oznaka 0 označava da dotični čvor nema dijete.

Na primjer, stablo na slici se može prikazati kao što je to prikazano u tabeli.

Na primjer, za čvor 15, djeca se nalaze na pozicijama 8 i 5. Ili, za čvor 21, djeca se nalaze na

pozicijama 4 i 2.

Oznaka 0 označava da dotični čvor nema dijete.

Ovakva implementacija je neprikladna u onim primjenama kod kojih je teško predvidjeti broj čvorova u stablu.

U tim situacijama se može dogoditi da je alocirani memorijski prostor premalen za zahtjeve koji se pojave u toku izvođenja programa, ili se pak može dogoditi da je alocirani prostor prevelik, što dovodi do nepotrebnog zauzeća memorijskih resursa.

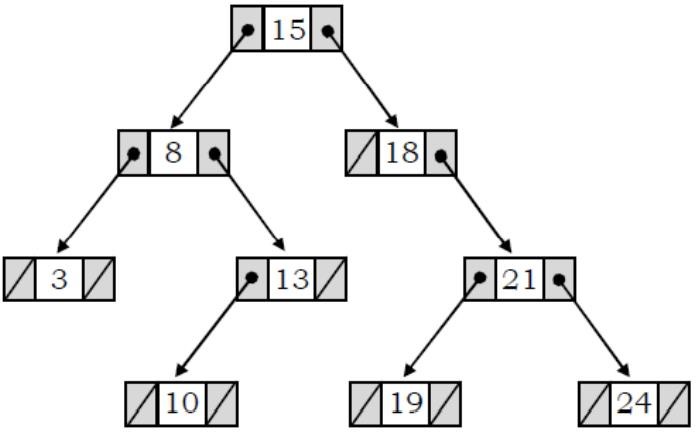
Postoje situacije kada je prikaz binarnog stabla pomoću niza opravdan i poželjan. Jedna od tih situacija će biti kasnije opisana kod implementacije posebnog tipa binarnog stabla koja se naziva gomila (eng. heap). Međutim, dinamičke strukture su ipak najčešće puno prikladnije za implementaciju stabla.

Najčešći način implementacije stabla je ulančana implementacija kod koje se koriste čvorovi koji sadrže informacioni sadržaj, kao i pokazivači na djecu.

Kod ovakve implementacije listovi sadrže prazne pokazivače, kao i čvorovi koji imaju izlazni stupanj manji od stupnja stabla. Prema tome, kod binarnih stabala se koriste dva pokazivača.

Jedan pokazivač pokazuje na korijen lijevog podstabla, a drugi pokazivač pokazuje na korijen desnog

podstabla.

Na slici je prikazana ulančana

implementacija binarnog stabla

Stabla se mogu prikazati i tako da se koriste nizovi povezanih listi (slika).

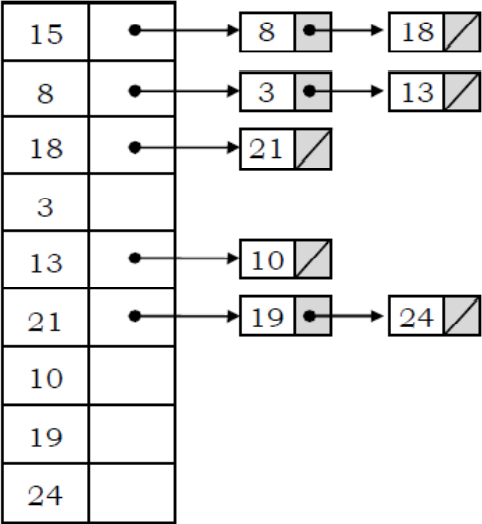
U nizu su na pojedinim lokacijama pohranjena zaglavlja listi, pri čemu se za svaki čvor stabla kreira

po jedna povezana lista.

Prva lokacija niza odgovara korijenu stabla.

U listi se nalaze sva djeca dotičnog čvora.

Ovaj prikaz je prikladniji za stabla kod kojih se puno razlikuju izlazni stupnjevi čvorova



## Operacije sa binarnim stablima pretraživanja

Uobičajene operacije sa binarnim stablima pretraživanja su:

pretraživanje;

pronalaženje minimalnog i maksimalnog elementa;

pronalaženje prethodnika i sljedbenika;

umetanje novog čvora;

brisanje čvora;

obilazak čvorova stabla

## Pretraživanje:

Zbog uređenosti binarnog stabla pretraživanja, operacija pretraživanja je efikasna.

Algoritam pretraživanja je opisan funkcijom TRAZI-BST

Funkcija za neku zadatu vrijednost ključa k i pokazivač na korijen stabla korijen vraća pokazivač na čvor sa ključem k, ako je taj ključ prisutan u stablu.

U suprotnom, funkcija vraća prazan pokazivač NIL.

**TRAZI-BST (korijen, k)**

**if(korijen = NIL) or k = kljuc(korijen)) then**

**return korijen**

**end\_if**

**if (k<kljuc(korijen)) then**

**return TRAZI-BST (lijevi(korijen),k)**

**else**

**return TRAZI-BST (desni(korijen),k)**

**end\_if**

Pretraživanje započinje u korijenu i rekurzivno se nastavlja spuštanjem kroz stablo prema podstablu koje sadrži zadati ključ k ako je taj ključ prisutan u stablu (uspješno pretraživanje), ili prema podstablu koje bi trebalo sadržavati zadati ključ k kada bi on bio prisutan u stablu (neuspješno

pretraživanje).

Ako je vrijednost k manja od vrijednosti kljuc u čvoru, rekurzivno spuštanje se nastavlja u lijevom

podstablu (linije 4-5).

U suprotnom, ako je ključ k veći od vrijednosti kljuc u čvoru, rekurzivno spuštanje se nastavlja u

desnom podstablu (linija 7).

Postupak rekurzivnog spuštanja se nastavlja sve dok se ne pronađe zadata vrijednost k ili dok se ne dođe do praznog podstabla što se identificira ispitivanjem uvjeta korijen = NIL or k=kljuc(korijen) (linija 1).

**TRAZI-BST-I(korijen,k)**

**while(korijen /= NIL) or k /= kljuc(korijen)) do**

**if ( k< kljuc(korijen)) then**

**korijen** 🡨 **lijevi(korijen)**

**else**

**korijen** 🡨 **desni(korijen)**

**end\_if**

**end\_while**

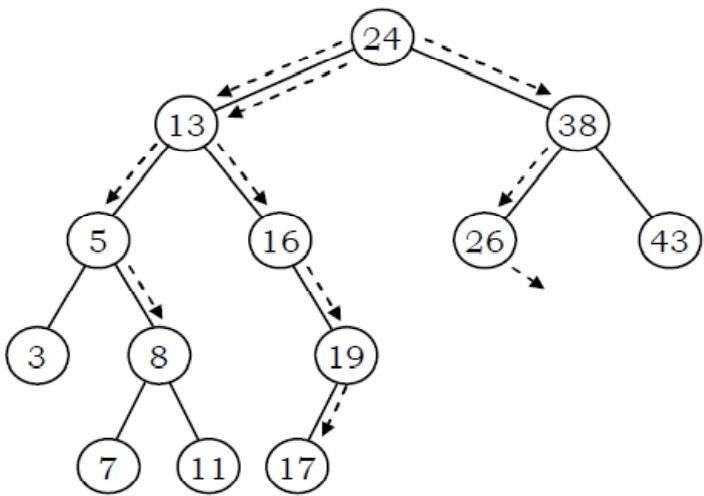
**return korijen**

Ovdje je opisana i iterativna varijanta pretraživanja binarnog stabla.

I ovdje imamo sličan način spuštanja kroz stablo kako je to prethodno opisano za rekurzivnu varijantu, ali je u iterativnoj varijanti to spuštanje realizirano unutar while petlje (linije 1-7).

Nakon završetka while petlje, vraća se pokazivač korijen, koji pokazuje na čvor koji sadrži ključ k, ili prazan pokazivač korijen=NIL ako ključ k nije prisutan u stablu (linija 8).

Na slici su ilustrirani primjeri uspješnog (8, 17) i neuspješnog pretraživanja (29). Pri pretraživanju ključa 8, slijedi se put od korijena: 24<-13<-5<-8.

Nadalje, pri pretraživanju ključa 17, put koji se slijedi je: 24<-13<-16<-19<-17. Ova dva pretraživanja su uspješna, jer završavaju

pronalaženjem traženog ključa.

S druge strane, pretraživanje ključa 29 završava neuspješno, jer se slijeđenjem puta 24<-38<-26<-NIL dolazi do čvora u kojem se proces pretraživanja usmjerava prema praznom podstablu, u kojem bi se

inače da je traženi čvor prisutan u stablu taj čvor trebao nalaziti

SLOŽENOST O(h)

## Minimalni element

Minimalni element u binarnom stablu pretraživanja se može pronaći tako da se od korijena slijedi lanac pokazivača na lijevo podstablo, sve dok se ne dođe do praznog podstabla.

Pronađeni čvor zapravo predstavlja najljevlji čvor u stablu.

Na primjer, u stablu prikazanom na slici, minimalni čvor 3 se pronalazi tako da od korijena slijedimo lanac pokazivača koji pokazuju na lijevu djecu, što u ovom slučaju generira put 24<-13<-5<-3.

Algoritam pronalaženja minimalnog ključa u binarnom stablu pretraživanja je prikazan u obliku funkcije MIN-BST

**MIN-BST (korijen)**

**1 while (lijevi(korijen) /= NIL) do**

**korijen 🡨 lijevi(korijen)**

**end\_while**

**return korijen**

## Maksimalni element:

Na sličan način se može pronaći i čvor koji sadrži maksimalan element u stablu. Taj čvor je najdesniji čvor i pronalazi se tako da se krene od korijena i slijedi lanac pokazivača na desno podstablo, sve dok se ne dođe do praznog podstabla.

Na primjer, u stablu prikazanom na slici, maksimalni čvor 43 se pronalazi tako da od korijena slijedimo lanac pokazivača koji pokazuju na desnu djecu, što u ovom slučaju generira put 24<-38<- 43.

Algoritam pronalaženja maksimalnog ključa u stablu je prikazan u obliku funkcije MAXBST

**MAX-BST(korijen)**

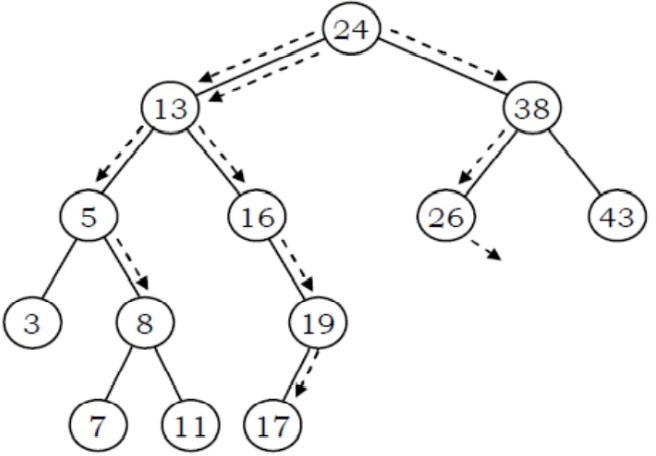
**1 while (desni(korijen) /= NIL) do**

**2 korijen 🡨 desni(korijen)**

**3 end\_while**

**return korijen**

## Sljedbenici i prethodnici:

Ponekada je u stablu binarnog pretraživanja važno pronaći sljedbenika i prethodnika nekog zadatog čvora.

Sljedbenik zadatog čvora je čvor sa najmanjim ključem koji je veći od ključa dotičnog čvora.

Taj ključ se, u slučaju kada zadati čvor ima desno podstablo, pronalazi tako da se traži najmanji

ključ u tom desnom podstablu

Na primjer, u stablu prikazanom na slici,

sljedbenik čvora 24 je čvor 26, jer je taj čvor najmanji (najljevlji) u desnom podstablu čvora 24

## Sljedbenik:

Pronalaženje sljedbenika zadatog čvora na kojeg pokazuje pokazivač r u binarnom stablu pretraživanja je opisano funkcijom SLJEDBENIK-BST

Funkcija vraća pokazivač na sljedbenika zadatog čvora, na koji pokazuje pokazivač r , ako sljedbenik postoji. U suprotnom, ako zadati čvor sadrži najveću vrijednost u stablu, funkcija vraća NIL

**SLJEDBENIK-BST (r)**

**1 if (desni(r) /= NIL) then**

**2 return MIN-BST (desni(r))**

**3 end if**

**4 q** 🡨 **roditelj(r)**

**while (q /= NIL and r = desni(q)) do**

**r** 🡨 **q**

**q** 🡨 **roditelj(q)**

**8 end\_while**

**return q**

Prikazana funkcija SLJEDBENIK-BST posebno obrađuje dvije moguće situacije.

Prva situacija je jednostavnija i prepoznajemo je po tome što zadati čvor ima desno podstablo. Tada je sljedbenik zapravo najmanji ključ u tom desnom podstablu, koji se pronalazi pozivom funkcije MINBST(desni(r)) (linije 1-3).

S druge strane, ako zadati čvor nema desno podstablo, pronalaženje njegovog sljedbenika je nešto složenije. Naime, u tom slučaju se sljedbenik zadatog čvora pronalazi tako da se identificira najmanji predak zadatog čvora čije je lijevo dijete također predak zadatog čvora.

U funkciji SLJEDBENIKBST ovaj slučaj se obrađuje u linijama 4 do 9. Tada se koristi pokazivač roditelj, koji treba postojati u čvoru i pomoću kojeg se od zadatog čvora krećemo prema korijenu, sve dok se ne nađe čvor koji je lijevo dijete svog roditelja ili dok se ne dođe do korijena (linije 5-8).

Nakon završetka while petlje pokazivač q pokazuje na sljedbenika zadatog čvora i funkcija vraća taj pokazivač (linija 9).

Bitno je istaknuti da se, zahvaljujući strukturi i organizaciji binarnog stabla pretraživanja, sljedbenik zadatog čvora pronalazi bez operacija poređenja ključeva.

Isto tako, napomenimo da samo najdesniji čvor, koji sadrži maksimalan ključ, nema svog sljedbenika,

pa tada funkcija SLJEDBENIK-BST vraća NIL

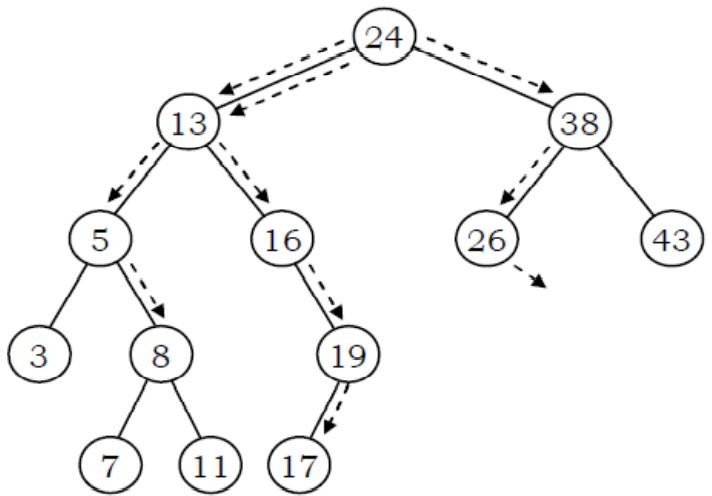
## Prethodnik:

Prethodnik zadatog čvora u binarnom stablu pretraživanja definiramo kao čvor sa najvećim ključem koji je manji od ključa zadatog čvora.

Taj ključ se pronalazi tako da se traži najveći ključ u lijevom podstablu čvora kojem tražimo

prethodnika.

Na primjer, u stablu prikazanom na slici prethodnik čvora 13 je čvor 11, jer je taj čvor najveći (najdesniji) u lijevom podstablu čvora 13



**PRETHODNIK (r)**

**If lijevi (q) /= NIL) then**

**return MAX-BS (lijevi(r))**

**end\_if**

**q 🡨 roditelj(r)**

**while (q /= NIL and r=lijevi(q)) do**

**r 🡨 q**

**q 🡨 roditelj(q)**

**end\_while**

**return q**

Postupak pronalaženja možemo opisati funkcijom PRETHODNIK-BST.

Pošto je taj postupak vrlo sličan pronalaženju sljedbenika, reći ćemo samo da se ta funkcija razlikuje u odnosu na funkciju SLJEDBENIK-BST u tome što je umjesto poziva MIN-BST(desni(r)) potrebno pozvati funkciju MAX-BST(lijevi(r)) u liniji 2, te umjesto dijela uvjeta r=desni(q) u liniji 5 treba da stoji r=lijevi(q).

Samo najljevlji čvor, koji sadrži minimalnu vrijednost ključa, nema svog prethodnika, pa tada funkcija

PRETHODNIK-BST vraća NIL

## Rekurzivne definicije

**(Predavanje 9) Rekurzija**

Primjer1: U cilju razumijevanja pojma rekurzije treba razmotriti sljedeću definiciju pretka neke

osobe:

roditelji neke osobe su njeni predci;

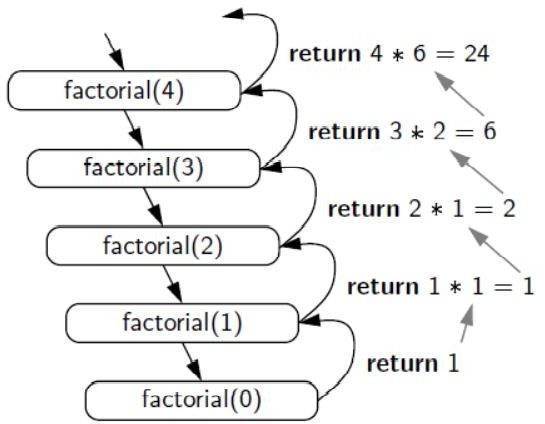
roditelji bilo kojeg pretka neke osobe su također njeni predci.

Primjer 2: Rekurzivna funkcija faktorijel

n! = n\*n-1\*n-2\*...\*2\*1

Ili

## Funkcija FAKTORIJEL

Funkcija kao argument uzima neki broj n i izračunava faktorijel tog broja

**FAKTORIJEL (n)**

**if(n=0) then**

**return 1**

**else**

**return n+FAKTORIJEL (n-1)**

**end\_if**

Rekurzivne definicije se obično sastoje od dva dijela:

osnovni slučaj;

rekurzivni slučaj.

Prvi dio rekurzivnih definicija je tzv. osnovni slučaj, koji predstavlja element definicije nekog pojma ili nekog rješenja na temelju kojeg se grade sva ostala rješenja.

Drugi dio je tzv. rekurzivni slučaj, koji sadrži pravila pomoću kojih se konstruiraju novi objekti na temelju osnovnih objekata ili na temelju objekata koji su prethodno konstruirani. Na primjer, pri definiciji pojma predak, osnovni slučaj bi bio prvi dio definicije, koji tvrdi da je roditelj neke osobe njen predak.

Rekurzivni slučaj definicije je drugi dio, koji tvrdi da su roditelji bilo kojeg pretka neke osobe također

predci te osobe

Osnovni slučaj u definiciji faktorijela je prvi dio definicije, koji se koristi kada se izračunava faktorijel

broja 0.

Rekurzivni slučaj je drugi dio definicije, koji se koristi za izračunavanje faktorijela svih brojeva većih

od 0

Bitno je istaknuti da svaki rekurzivni algoritam mora imati uvjet završetka da bi se izbjegao

beskonačni niz rekurzivnih poziva.

Na primjer, kod funkcije FAKTORIJEL, uvjet kojim se kontrolira završetak rekurzivnih poziva je n = 0. Naime, rekurzivne pozive funkcije možemo promatrati kao spuštanje za jedan nivo dublje u algoritamskom procesu, pa moramo koristiti uvjet završetka rekurzije da bismo nakon određenog nivoa omogućili podizanje u nivoima rekurzije sve do nivoa prvog poziva dotične funkcije

## Rekurzija – podijeli pa vladaj

Posebno bitna primjena rekurzivnih algoritama je pri korištenju jedne važne strategije rješavanja problema koja se naziva „podijeli-pa-vladaj“ (eng. divide and conquer).

Primjer: Neka je potrebno oblikovati algoritam za potenciranje broja x s brojem n, gdje je n pozitivni

cijeli broj, pri čemu se mogu koristiti samo dvije operacije:

množenje dva broja;

oduzimanje.

Ako nemamo operaciju koja rješava 𝑠 = 𝑥 𝑛 moramo riješiti slijedeće probleme:

potencirati x s brojem 1: s = x;

potencirati x s brojem 2: s = x\*x.

Za n>2 imamo dva problema:

Problem P1: potencirati x s brojem i. Neka je S1 rješenje za problem P1.

Problem P2: potencirati x s brojem n-i. Neka je S2 rješenje za problem P2.

Nakon toga S=S1\*S2

Na koji način riješiti probleme P1 i P2

Oni su problemi istog tipa kao i originalni problem, tako da možemo primijeniti istu strategiju, koju

smo primijenili na originalni problem.

Provjerimo da li se problemi mogu riješiti direktno. Naime, ako je i=1 ili i=2, tada možemo P1 riješiti

direktno.

Nadalje, ako je n-i=1 ili n-i=2, tada možemo P2 riješiti direktno.

Probleme koje ne možemo riješiti direktno, rješavat ćemo rekurzivno...

Npr. 𝑠 = 3 7

P1 potencirati 3 sa 2 (S(1)=9)

P2 potencirati 3 sa 5

Sada treba podijeliti problem P2 na:

P2-1 potencirati 3 sa 1 (S(2-1) = 3)

P2-2 potencirati 3 sa 4

Dalje treba podijeliti problem P2-2 na:

P2-2-1 potencirati 3 na 2 (S(2-2-1)=9)

P2-2-2 potencirati 3 na 2 (S(2-2-2)=9) → S2-2=81\*S2-1\*S1=2187

Strategija „podijeli pa vladaj“ se sastoji od:

Podijeliti početni problem P na nekoliko (m) podproblema: P1, P2,...Pm;

riješiti m podproblema, što rezultira dobivanjem rješenja S1, S2,...Sm;

kombinirati S1, S2, ...Sm - da se konstruira rješenje za početni problem P

Za rješavanje podproblema se koristi rekurzivni postupak jer podrazumijeva istu strategiju

Ako je P1 veći od limita šta bi se dešavalo

Na primjer, ako je veličina ulaza za problem P1 veća od limita, onda taj problem možemo dalje podijeliti na podprobleme (P1-1, P1-2, ..., P1-p), te svaki od tih problema rješavati posebno. Dobivena rješenja (S1-1, S1-2, ..., S1-p) treba kombinirati, da bi se dobilo rješenje S1 za problem P1

## Funkcija PODIJELI-PA-VLADAJ

Općenito govoreći, algoritmi se mogu klasificirati ili kao iterativni ili kao rekurzivni.

Rekurzivnim algoritmima rješavamo probleme tako što ih dijelimo na manje podprobleme, a zatim

algoritam primjenjujemo na svaki od podproblema.

Generički algoritam PODIJELI-PA-VLADAJ prvo provjerava da li je veličina početnog problema n dovoljno mala, da se može direktno dobiti rješenje nekim jednostavnim nerekurzivnim algoritmom (linija 1). Ako se provjerom utvrdi da je problem dovoljno mali, onda se na temelju osnovnog slučaja direktno dobija rješenje (linija 2).

**PODJELI-PA-VLADAJ()**

**if(n<=n0)then**

**direktno rijesiti pocetni problem P**

**else**

**podijeliti pocetni problema P na m podproblema Pi i=1…m**

**for i=1 to m do**

**kreirati rjesenja Si za problem Pi rekurzije pozivom algoritma PODJELI-PA-VLADAJ**

**end\_for**

**kreirati rjesenje S kombiniranjem m rjesenja Si**

**end\_if**

S druge strane, ako je problem prevelik, on se na neki način dijeli na m manjih podproblema Pi, i=1,2,...m (linija 4). Ti manji podproblemi mogu imati istu veličinu, ili pak mogu imati različite, a ponekada i drastično različite veličine.

Algoritam PODIJELI-PA-VLADAJ se zatim rekurzivno poziva za svaki od manjih podproblema Pi, da bi

se za te podprobleme kreirala rješenja Si, i=1,2,...m (linije 5-7).

Na kraju se rezultati rekurzivnih poziva kombiniraju, da bi se dobilo rješenje S na pojedinim nivoima

rekurzije (linija 8).

## Funkcija PODIJELI-PA-VLADAJ za FAKTORIJEL

Poređenjem algoritma FAKTORIJEL-M i generičkog algoritma vidimo da je limit veličine u ovom

slučaju 1, kada je moguće dobiti direktno rješenje koje iznosi 1. U svim ostalim slučajevima se koristi

blok naredbi u sklopu else klauzule (linije 4-7).

Prva faza u generičkom algoritmu je podijeliti ulaz na manje dijelove. U slučaju funkcije FAKTORIJEL- M imamo izračunavanje i dodjelu ulaz<-n-1. Sljedeći korak u generičkom algoritmu je rekurzivno pozivanje istog algoritma s ciljem rješavanja manjih podproblema.

Kod funkcije FAKTORIJEL-M imamo samo jedan rekurzivni poziv sa veličinom problema koja je za

jedan manja od originalnog (linija 5). Na kraju, kao zadnji korak generičkog algoritma imamo kombiniranje rješenja. Kod funkcije FAKTORIJEL-M kombiniranje rješenja je implementirano množenjem n\*s1 (linija 6)

**FAKTORIJEL-M(n)**

**if(n=0)then**

**return 0;**

**else**

**ulaz<--n-1**

**s1<--FAKTORIJEL-M(ulaz)**

**s<--n\*s1**

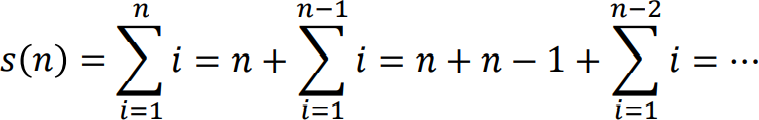
**return s**

**end\_if**

## Funkcija PODIJELI-PA-VLADAJ za SUMU

Prema tome, ključni dio kod strategije „podijeli-pa-vladaj“ je razbijanje problema na podprobleme, pri čemu u rekurzivnom koraku moramo smanjiti veličinu početnog problema.

Nadalje, proces razbijanja problema na sve manje podprobleme mora završiti osnovnim slučajem.

Tek kada je veličina podproblema manja od nekog limita n0, tada imamo tzv. osnovni slučaj i primjenjujemo direktno rješenje. Na primjer, razmotrimo sljedeću sumu brojeva

odnosno vrijedi s(n)=1 za n=1 i s(n)=n+s(n-1) za n>1.

Linija 2 se izvršava za elementarni slučaj

Linija 4 se izvršava kao rekurzija

**SUMA(n)**

**if(n=1)then**

**s<--1**

**else**

**s<--n+SUMA(n-1)**

**end\_if**

**return s**

## HANOJSKI TORNJEVI (PRSTENOVI)

Tri su tornja (S-izvorni, T-pomoćni, D-odredišni), ali i pravila:

sa bilo kojeg štapa se može prebacivati samo disk sa vrha;

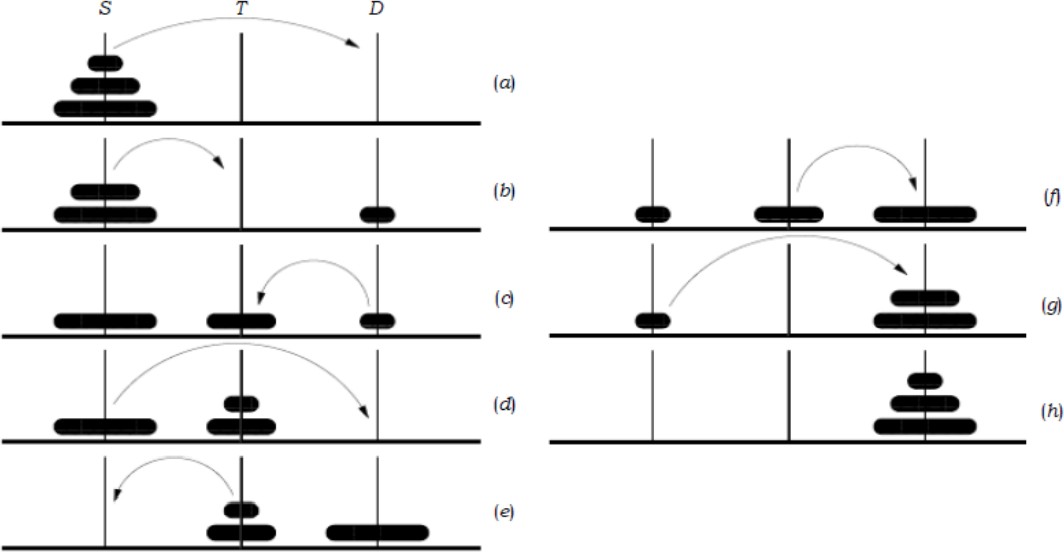
niti u jednom trenutku nije dozvoljeno da se veći disk postavlja na manji



Ako se na izvornom štapu S nalazi samo jedan disk, onda je rješenje vrlo jednostavno i sastoji se u direktnom prebacivanju diska na odredišni štap D.

Rješenje je jednostavno i ako imamo dva diska. U tom slučaju, gornji manji disk se premjesti na štap

T, a donji veći disk na štap D. U sljedećem koraku se disk sa štapa B premjesti na štap D

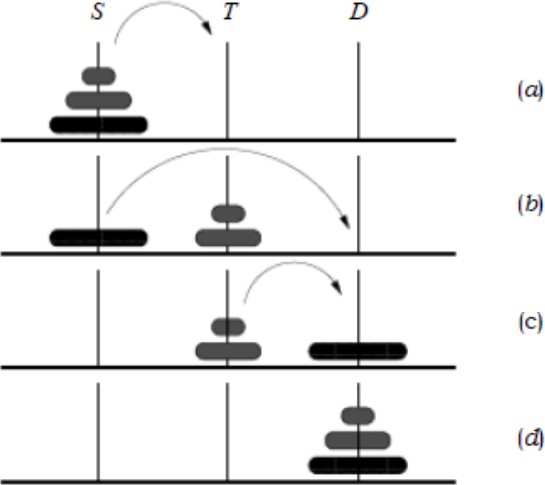


## HANOJSKI TORNJEVI (PRSTENOVI) – rješenje „podijeli pa vladaj“

Na slici su ilustrirane samo ključne faze rješavanja problema hanojskih tornjeva strategijom „podijeli- pa-vladaj“. Naime, na slici a je prikazan početni raspored štapova, dok je na slici b prikazan raspored štapova koji se dobije nakon dijeljenja početnog problema na dva manja podproblema P1 i P2:

problem P1: prenijeti disk sa štapa S na štap D;

problem P2: prenijeti 2 diska sa štapa T na štap D koristeći štap S kao pomoćni.



Problem P1 možemo riješiti direktno, dok je problem P2 problem istog tipa kao početni problem, ali

je manji.

Primijetimo da je raspored štapova na slici b jednak rasporedu štapova na slici d, pa možemo

konstatirati da su koraci kojima se početni problem dijeli na dva manji podproblema prikazani na

slikama a-d.

Osim toga, primijetite da su ti koraci zapravo istog tipa kao i početni problem. Prema tome, faza

dijeljenja početnog problema se sastoji u tome da 2 diska sa štapa S premjestimo na štap T, koristeći kao pomoćni štap D.

Drugim riječima, strategiju rješavanja problema hanojskih tornjeva, za slučaj kada je broj diskova n=3, možemo opisati na sljedeći način:

premjestiti 2 diska sa štapa S na štap T, koristeći D kao pomoćni štap (slike a-b);

premjestiti disk 3 sa štapa S na štap D (slike b-c);

premjestiti 2 diska sa štapa T na štap D, koristeći S kao pomoćni štap (slike c-d).

Na temelju prethodno opisanih koraka pri rješavanju problema hanojskih tornjeva za slučaj kada je broj diskova n=3, možemo poopćiti strategiju rješavanja problema hanojskih tornjeva na slučaj kada je broj diskova n, na sljedeći način:

premjestiti n-1 diskova sa štapa S na štap T, koristeći D kao pomoćni štap;

premjestiti disk n sa štapa S na štap D;

premjestiti n-1 diskova sa štapa T na štap D, koristeći S kao pomoćni štap

Prema tome, zaključujemo da se rješenje problema za n diskova reducira na rješavanje problema istog tipa za n-1 diskova, pa se kao rješenje nameće rekurzivni pristup. Na slici je prikazana procedura HANOJSKI-TORNJEVI, koja ima sljedeće ulazne argumente:

n broj diskova;

S izvorni štap;

D odredišni štap;

T pomoćni štap.

**HANOJSKI-TORNJEVI (n, S, D, T)**

**1 if (n = 1) then**

**2 PRINT(„Preseliti disk 1 sa štapa S na štap D“)**

**3 return**

**4 end if**

**5 HANOJSKI-TORNJEVI (n-1, S, T, D)**

**PRINT („Preseliti disk n sa štapa Sna štap D“)**

**HANOJSKI-TORNJEVI (n-1, T, D, S)**

**return**

## Linearna i repna rekurzija

Strategija „podijeli pa vladaj“ se sastoji od:

Podijeliti početni problem P na nekoliko (m) podproblema: P1, P2,...Pm;

riješiti m podproblema, što rezultira dobivanjem rješenja S1, S2,...Sm;

kombinirati S1, S2, ...Sm - da se konstruira rješenje za početni problem P

Za rješavanje podproblema se koristi rekurzivni postupak jer podrazumijeva istu strategiju • Ako je P1 veći od limita šta bi se dešavalo

Na primjer, ako je veličina ulaza za problem P1 veća od limita, onda taj problem možemo dalje podijeliti na podprobleme (P1-1, P1-2, ..., P1-p), te svaki od tih problema rješavati posebno. Dobivena rješenja (S1-1, S1-2, ..., S1-p) treba kombinirati, da bi se dobilo rješenje S1 za problem P1

Sve rekurzivne definicije funkcija sadrže poziv funkcije koja se definira. Međutim, postoje različiti načini kako se ti pozivi mogu implementirati.

Na primjer, rekurzivna funkcija može samo jednom pozivati samu sebe, ali isto tako, rekurzivna funkcija može pozivati samu sebe više puta.

Zatim, rekurzivna funkcija može biti jednostavna ili implementirana na složen način.

Rekurzija kod koje se funkcija rekurzivno poziva samo jednom se naziva linearna rekurzija. Primjer

takve rekurzivne funkcije je već prethodno prikazan funkcijom FAKTORIJEL

Poseban slučaj linearne rekurzije je tzv. repna rekurzija, koju karakterizira svojstvo da se koristi

jedan rekurzivni poziv na samom kraju implementacije funkcije.

Drugim riječima, nakon rekurzivnog poziva u funkciji nema više naredbi koje se trebaju izvršiti, te

osim tog poziva u funkciji nema drugih rekurzivnih poziva.

Repnu rekurziju ćemo ilustrirati na još jednom primjeru funkcije za izračunavanje faktorijela

## Repna rekurzija funkcije FAKTORIJEL

Funkcija FAKTORIJEL se može riješiti upravo repnom rekurzijom, jer je zadnja naredba zapravo poziv

te funkcije.

Za izračunavanje faktorijela broja n potrebno je napraviti poziv TAILFAKTORIJEL (n,1).

Dakako, pri izračunavanju faktorijela broja n, sintaksno bi bilo jasnije koristiti kao ulazni argument

samo taj broj n.

Da bismo prikrili korištenje pomoćnog parametra rezultat, možemo npr. napraviti jednostavnu pomoćnu funkciju FAKTORIJEL-POM koja je prikazana na slici b

**TAIL-FAKTORIJEL (n, rezultat )**

**1 if (n = 0) then**

**2 return rezultat**

**3 else**

**4 return TAIL-FAKTORIJEL (n-1, n\*rezultat)**

**5 end if**

**FAKTORIJEL-POM in)**

**1 return TAIL-FAKTORIJEL (n, 1)**

## Binarna rekurzija

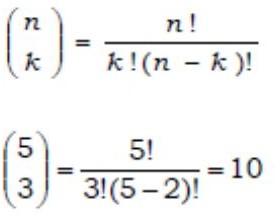
Rekurzivne funkcije ponekada sadrže više od jednog rekurzivnog poziva. Ako se izvršavanjem

funkcije ostvaruju dva rekurzivna poziva, onda takve funkcije nazivamo binarne rekurzivne funkcije.

Primjer binarne rekurzije možemo ilustrirati, na primjer, određivanjem podskupova sa k elemenata

(k kombinacija) iz skupa sa n elemenata.

Broj takvih kombinacija je u slučaju skupa od 5 elemenata 10

Odnosno (0,1,2), (0,1,3), (0,1,4), (0,2,3), (0,2,4), (0,3,4), (1,2,3),

(1,2,4), (1,3,4), (2,3,4)

Razmotrimo sada mogućnost da problem izračunavanja broja kombinacija pokušamo riješiti rekurzivnim pristupom.

Rekurzivni pristup nam sugerira da je potrebno pronaći jednostavniju

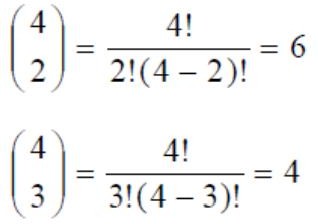
verziju istog problema.

Jednostavnija verzija istog problema bi moglo biti rješenje za manju

vrijednost n ili manju vrijednost k.

Analizirajmo kombinacije za skup A tako, da sve moguće kombinacije particioniramo u dvije grupe:

G1 i G2.

U prvu grupu svrstajmo kombinacije koje sadrže element 0, dok su u

drugu grupu svrstajmo sve ostale kombinacije.

Pošto je u prvoj grupi element 0 prisutan u svim kombinacijama, a

druga dva elementa zapravo predstavljaju sve moguće kombinacije od 2 elementa iz skupa od 4 elementa, to znači da je ukupan broj kombinacija u grupi G1 (gornja slika).

Druga grupa G2 sadrži kombinacije koje ne sadrže element 0, pa sva

tri elementa zapravo sadrže sve kombinacije iz skupa {1, 2, 3, 4}, koji ima četiri elementa. Prema

tome, u drugoj grupi broj kombinacija je (donja slika)

Drugim riječima, ako pozivom funkcije IZBOR(n, k) izračunavamo broj kombinacija sa k elemenata iz skupa od n elemenata, onda smo dobili da je: IZBOR (5, 3)= IZBOR (4, 2)+ IZBOR (4, 3)

Prema tome, dobili smo jednostavnije verzije istog problema jer su argumenti u jednostavnijim verzijama umanjeni

**IZBOR (n, k)**

**1 if (k = 0 or n = k) then**

**2 return 1**

**3 else**

**4 return IZBOR (n-1, k)+IZBOR (n-1, k-1)**

**5 end\_if**

Prethodna razmatranja na konkretnom primjeru možemo poopćiti tako da na sličan način

razmotrimo izbor k elemenata iz skupa od n elemenata. Naime, opet sve moguće kombinacije možemo particionirati na dva skupa.

U prvi skup možemo svrstati sve kombinacije koje sadrže prvi element e1, a u drugi skup su svrstane sve kombinacije koje ne sadrže element e1. Pošto je u prvoj grupi element e1 prisutan u svim kombinacijama, ostalih k-1 elemenata predstavljaju sve moguće kombinacije od k-1 elemenata iz skupa od n-1 elemenata. Drugim riječima, broj kombinacija u prvoj grupi je IZBOR (n-1, k-1).

Druga grupa sadrži sve kombinacije koje ne sadrže element e1, pa ta grupa zapravo sadrži svih k

elemenata iz skupa sa n-1 elemenata.

Prema tome, broj kombinacija u drugoj grupi je IZBOR (n-1, k).

Dakle, ukupan broj kombinacija izbora k elemenata iz skupa sa n elemenata je: IZBOR (n, k) = IZBOR (n-1, k-1)+IZBOR (n-1, k)

Na temelju prethodnog izraza ćemo dizajnirati rekurzivnu funkciju IZBOR.

Još je samo potrebno da odredimo uvjet završetka rekurzije. Iz prethodnog izraza vidimo da će se u

dijelu sume IZBOR (n-1,k) argument n rekurzivno smanjivati za 1.

Međutim, ako se n smanji tako da je ispunjeno n=k, tada zapravo imamo situaciju da se izračunava broj kombinacija sa k elemenata iz skupa sa k elemenata, što je trivijalan slučaj, jer tada imamo samo jednu kombinaciju.

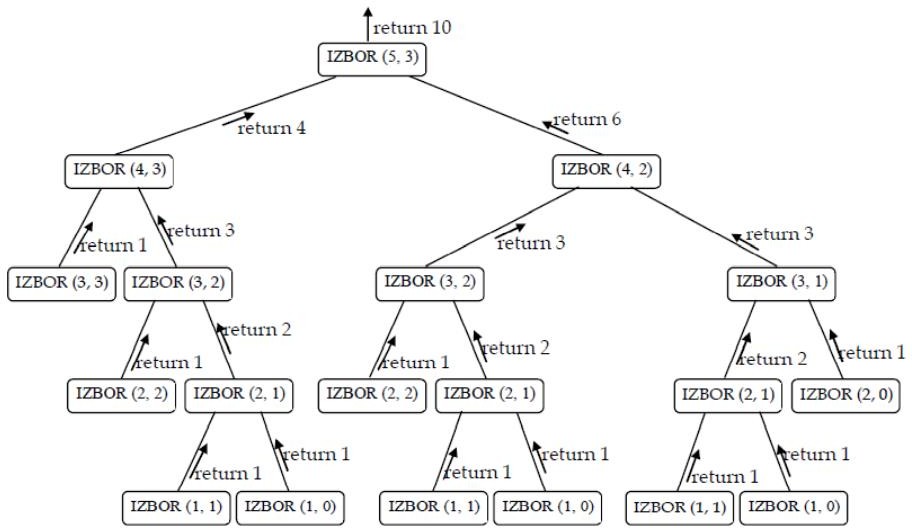
Prema tome, prvi uvjet završetka rekurzije je n=k, te funkcija u tom slučaju treba vratiti vrijednost 1.

Isto tako, analizirajmo drugi dio sume IZBOR (n-1, k-1). Ovdje imamo rekurzivno smanjivanje i vrijednosti n i vrijednosti k za 1.

Na određenom nivou rekurzije, pri pozivu funkcije IZBOR će prenesena vrijednost argumenta k

poprimiti vrijednost 0.

Međutim, određivanje broja kombinacija na koji se može izabrati 0 elemenata iz nekog skupa je također trivijalan slučaj i taj broj je 1. Prema tome, funkcija treba vratiti vrijednost 1 ako je ispunjeno: k=0 ili n=k

IZBOR (5,3)

## Eksponencijalna rekurzija

Ako se u funkciji nalazi više od jednog poziva rekurzivne funkcije, onda takve funkcije nazivamo eksponencijalne rekurzivne funkcije. Kod ovakvih funkcija broj rekurzivnih poziva raste eksponencijalno.

Eksponencijalnu rekurziju ćemo ilustrirati rješavanjem relativno čestog problema pronalaženja

permutacija nekog skupa.

Permutacije skupa od n elemenata su mogući uređeni razmještaji elemenata ili mogući uređeni

izbori elemenata iz tog skupa.

Broj permutacija skupa od n elemenata je n!.

Pođimo od jednog konkretnog primjera. Pretpostavimo da je zadat skup koji je prikazan u obliku cjelobrojnog niza A[0..3]={1, 2, 3, 4}. Neka je potrebno pronaći sve permutacije elemenata tog skupa.

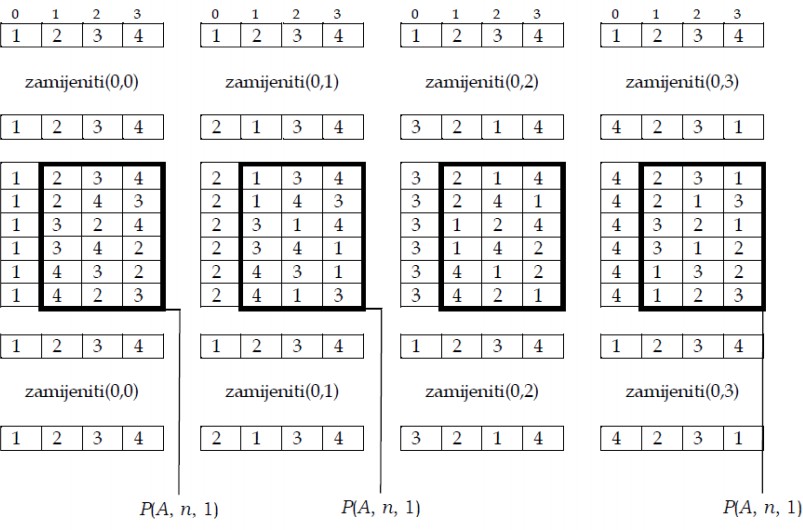
Označimo početni problem sa P0, a skup svih permutacija koji predstavlja njegovo rješenje sa S0.

Na slici je ilustriran jedan od mogućih pristupa rješavanju problema P0.

Skup S0 možemo podijeliti na četiri disjunktna podskupa Si (i=0, 1,2, 3).

Svaki od podskupova Si je kreiran tako da se na prvoj poziciji nalazi element A[i], a na ostalim

pozicijama j (j≠i) se nalaze permutacije preostala 3 elementa.



Na primjer, skup S1 sadrži sve permutacije koje na prvoj poziciji sadrže element A[1]=2, dok skup S3 označava sve permutacije koje na prvoj poziciji sadrže element A[3]=4.

Nadalje, u skupu S1 sve permutacije na preostale tri pozicije zapravo sadrže permutacije podskupa

{1, 3, 4}, dok u skupu S3 sve permutacije na preostale tri pozicije sadrže permutacije podskupa {1, 2,

3}.

Prema tome, početni problem P0 pronalaženja permutacija elemenata niza A možemo podijeliti na

četiri manja podproblema: P0-0, P0-1, P0-2 i P0-3 koji se svode na pronalaženja permutacija sljedeća

4 podniza:

A0-0[1..3]={2, 3, 4}, A0- 1[1..3]={1, 3, 4}, A0- 2[1..3]={1, 2, 4}, A0- 3[1..3]={1,2, 3}.

A0-0[1..3]={2, 3, 4}, A0- 1[1..3]={1, 3, 4}, A0- 2[1..3]={1, 2, 4}, A0- 3[1..3]={1,2, 3}.

Indekse u A0-0 označimo sa Ak-i. Prvi indeks k označava pozicije 0..k u nizu A na kojima su elementi fiksirani, dok indeks i označava poziciju elementa A[i], k ≤ i ≤ 3, koji razmjenjuje poziciju sa elementom A[k], te se fiksira na poziciji k

Nadalje, rješavanju problema P0-0 pronalaženja permutacija podniza A0-0[1..3]={2, 3, 4} možemo pristupiti na isti način kao i početnom problemu.

Naime, problem P0-0 dijelimo na manje podprobleme (P0-0-1, P0- 0-2 i P0-0-3), koji se sastoje u

pronalaženju permutacija sljedećih podnizova

A0-0-1[2..3]={3, 4},

A0-0-2[2..3]={2, 4},

A0-0-3[2..3]={2, 3}.

Naravno, na isti način se pristupa i rješavanju preostala tri podproblema P0-1, P0-2 i P0-3 na nultom nivou.

Problemi P0-0-1, P0-0-2 i P0-0-3 nisu dovoljno reducirani da se može primijeniti direktno rješenje, pa se proces nastavlja daljnim reduciranjem veličine problema.

Razmotrimo, na primjer, proces rješavanja problema P0-0-1. Pronalaženje permutacija podniza A0- 0-1[2..3]={3, 4} rješavamo dijeljenjem na drugom nivou rekurzije na još manje podprobleme P0-0-1- 0 i P0-0-1-1 koji se sastoje u pronalaženju permutacija sljedećih podnizova:

A0-0-1-1[3..3]={4}, A0-0-1-2[3..3]={3}.

Na isti način se pristupa i rješavanju preostala dva podproblema na prvom nivou P0-0-2 i P0-0-3. Razmotrimo sada pristup rješavanju problema P0-0- 2. Budući da podnizovi A 0-0-1-1 i A 0- 0-1-2 sadrže samo po jedan element, problemi ovi problemi su reducirani na osnovne slučajeve, te se rješavaju direktno bez rekurzivnog koraka.

Drugim riječima, mogu se ispisati sljedeće permutacije: (1, 2, 3, 4) i (1, 2, 4, 3).

Ove dvije permutacije zapravo predstavljaju rješenje problema P1-1.

Na sličan način se dobiva i rješenje problema P1-2 koje uključuje permutacije (1, 3, 2, 4) i (1, 3, 4, 2), te rješenje problema P1,3 koje uključuje permutacije (1, 4, 3, 2) i (1, 4, 2, 3).

**ISPISI-PERMUTACIJE (A, n, k)**

**1 if (k = n-1) then**

**2 ISPISI (A, n)**

**3 else**

**4 for it k to n-1 do**

**5 A[k] <--> A[i]**

**6 ISPISI-PERMUTACIJE (A, n, k+1)**

**A[k] <--> A[i]**

**end\_for**

**end\_if**

**ISPISI (A, n)**

**1 for it 0 to n-1 do**

**2 PRINT (A[1])**

**3 end\_for**

Prethodno opisani pristup za ispis pemutacija elemenata niza A možemo lako poopćiti i na nizove sa n elemenata na sljedeći način:

Osnovni slučaj: Ako podniz

A[k, n-1], 0 ≤ k ≤ n-1, za koji tražimo permutacije ima samo jedan element (k=n-1), tada ispisom elemenata niza A[0..n-1] direktno rješavamo problem.

Rekurzivni dio:

Na k-tom nivou rekurzije problem Pk dijelimo na n-k manjih podproblema Pk-k, Pk,k+1,...Pk,n-1

pomoću sljedećih koraka:

a) razmijenimo pozicije prvog elementa A[k] u podnizu A[k..n-1] i svih ostalih elemenata u

nizu A[i], k ≤ i ≤ n-1, što rezultira kreiranjem sljedećih n-k nizova: Ak,i[k..n-1]={ai, ak+1, ak+2,

...ai-1,ak, ai+1,...an-1), i=k, k+1,...,n-1;

b) Fiksiramo prvi element Ak,i[k], te kreiramo n-k novih manjih podproblema koji se sastoje

u pronalaženju permutacija za podnizove

Ak+1,i[k+1...n-1], i=k+1,1,...n-1.

Rješavamo podprobleme Pk,i na isti način kao i početni problem P0

Na kraju rješavanja svakog problema Pk,i ponovno izmjenjujemo pozicije elemenata A[k] i A[i].

Vrlo sličan pristup bismo mogli koristiti i pri rješavanju problema pronalaženja permutacija u situaciji kada je dozvoljeno mijenjati razmještaj samo elementima na pozicijama koje se nalaze u rasponu k do m.

Na primjer, neka je zadat niz A[0..4]={1, 2, 3,4, 5} i neka je dozvoljeno mijenjati razmještaj samo

elemenata na pozicijama k=1 do m=3.

Tada imamo sljedećih 6 permutacija:

(1, 2, 3, 4, 5)

(1, 2, 4, 3, 5)

(1, 3, 2, 4, 5)

(1, 3, 4, 2, 5)

(1, 4, 3, 2, 5)

(1, 4, 2, 3, 5)

**ISPISI-PERMUTACIJE-K-DO-m (A, n, k, m)**

**1 if (k = m) then**

**2 ISPISI(A, n)**

**3 else**

**4 for it k to m do**

**5 A[k] <--> A[i]**

**6 ISPISI-PERMUTACIJE-k-DO-M (A, n, k+1, m)**

**7 A[k] <-> A[1]**

**8 end for**

**end if**

## Uzajamna rekurzija

Ako neka funkcija F1 ne poziva samu sebe direktno, već poziva neku drugu funkciju F2, a funkcija F2 poziva neku treću funkciju F3, pa sve tako do neke n-te funkcije Fn, koja poziva prvu funkciju F1, onda takvu rekurziju nazivamo uzajamna ili indirektna rekurzija.

Funkcije koje rade na principu uzajamne rekurzije mogu funkcionirati u parovima ili u većim

grupama

**JE-LI-PARAN-UR (n)**

**1 if (n = 0) then**

**2 return true**

**3 else**

**4 return JE-LI-NEPARAN-UR (n-1)**

**5 end\_if**

**JE-LI-NEPARAN-UR (n)**

**1 return (!JE-LI-PARAN-UR (n))**

Ako neka funkcija F1 ne poziva samu sebe direktno, već poziva neku drugu funkciju F2, a funkcija F2 poziva neku treću funkciju F3, pa sve tako do neke n-te funkcije Fn, koja poziva prvu funkciju F1, onda takvu rekurziju nazivamo uzajamna ili indirektna rekurzija.

Funkcije koje rade na principu uzajamne rekurzije mogu funkcionirati u parovima ili u većim

grupama

**JE-LI-PARAN (n)**

**1 if (n mod 2 = 0) then**

**2 return true**

**3 else**

**4 return false**

**5 end if**

**JE-LI-NEPARAN (n)**

**1 if (n mod !2 = 0) then**

**2 return true**

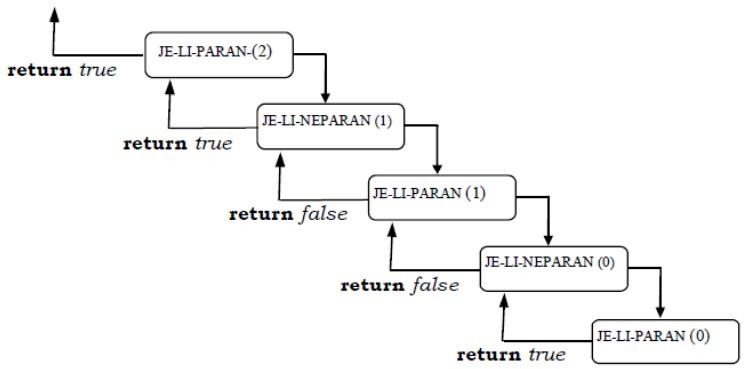
**3 else**

**4 return false**

**5 end\_if**

Ako neka funkcija F1 ne poziva samu sebe direktno, već poziva neku drugu funkciju F2, a funkcija F2 poziva neku treću funkciju F3, pa sve tako do neke n-te funkcije Fn, koja poziva prvu funkciju F1, onda takvu rekurziju nazivamo uzajamna ili indirektna rekurzija.

Funkcije koje rade na principu uzajamne rekurzije mogu funkcionirati u parovima ili u većim

grupama

## Efikasnost rekurzivnih algoritama

Generički algoritam PODIJELI-PA-VLADAJ kojim smo na jedan poopćen način opisali strategiju

„podijeli-pa-vladaj“ ima četiri koraka: korak direktnog rješavanja problema, korak dijeljenja

početnog problema na manje podprobleme, korak rekurzivnih poziva algoritma i korak kombiniranja rješenja.

Pri analizi vremenske složenosti nekog konkretnog rekurzivnog algoritama potrebno je da dovedemo u vezu pojedine korake tog algoritma i generičkog algoritma PODIJELI-PA-VLADAJ, prikazanog na donjoj slici

**PODIJELI-PA-VLADAJ ()**

**1 if (n <= no) then**

**2 Direktno rješiti početni problem P**

**3 else**

**4 Podijeliti početni problema P na m podproblema Pi, i=1..m**

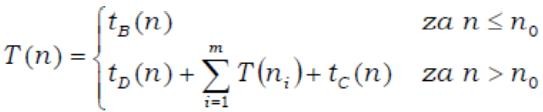
**5 for i=1 to m do**

**6 Kreirati rješenja Si za probleme Pi rekurz. pozivom algoritma PODIJELI-PA-VLADAJ**

**7 end\_for**

**8 Kreirati rješenje S kombiniranjem m rješenja Si**

**end\_if**

Nakon što za svaki korak konkretnog algoritma koji analiziramo odredimo njegovu vremensku složenost, možemo koristiti sljedeći generički izraz:

gdje je: T(n) broj operacija algoritma PODIJELI-PA-VLADAJ; tB(n) broj operacija za osnovni (bazni) slučaj koji se rješava direktno; tD(n) broj operacija faze kojom se početni problem dijeli na manje podprobleme; tC(n) broj operacija faze kojom se kombiniraju rješenja na nekom od nivoa rekurzije; n0 limit veličine problema koji se može riješiti direktno na temelju osnovnog slučaja; n veličina

početnog ulaznog problema; ni veličina reduciranih podproblema koji su istog tipa kao i početni

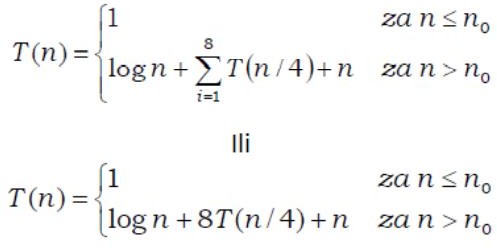
problem, i=1,2,...m (m- broj podproblema).

Pretpostavimo, na primjer, da smo neki problem rješavali strategijom „podijeli-pavladaj“ tako da je početni problem podijeljen na m=8 manjih podproblema, pri čemu svaki od tih podproblema ima veličinu ulaza npr. n/4. Nadalje, pretpostavimo da se direktno rješenje može koristiti za veličine

problema n ≤ 2, da je potreban broj operacija za direktno rješenje tB=1, da je potreban broj

operacija za dijeljenje početnog problema tD=log n, te da je potreban broj operacija za kombiniranje

rješenja tC(n)=n

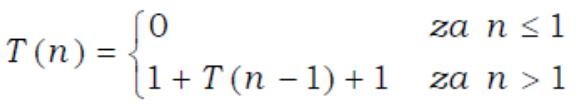


Pošto je vrijednost funkcije T(n) izražena pomoću nje same, ovakav oblik funkcije se naziva rekurzivna ili rekurentna relacija (eng. „recurrence relations“).

Vratimo se na primjer funkcije FAKTORIJEL. Prethodno smo identificirali sve korake ovog algoritma u odnosu na generički algoritam PODIJELI-PA-VLADAJ, pa ćemo tu identifikaciju koristiti da bismo odredili rekurzivnu relaciju za algoritam FAKTORIJEL.

Vidjeli smo da direktno rješenje ne zahtijeva nikakve operacije, pa je tB(n)=0. Korak reduciranja

početnog problema zahtijeva jednu operaciju, pa je tD(n)=1. Isto tako, korak kombiniranja zahtijeva jednu operaciju, pa je tC(m)=1. Rekurzivnim korakom se problem reducira na jedan problem (m=1) istog tipa s veličinom ulaza n-1. Dakle, rekurzivna relacija za broj operacija funkcije FAKTORIJEL je



## Stabla rekurzije

Kroz nekoliko primjera ćemo uvesti koncept stabla rekurzije. Privremeno zanemarimo osnovni slučaj, te se fokusirajmo samo na rekurzivni dio koji je npr. definiran sljedećim izrazom:



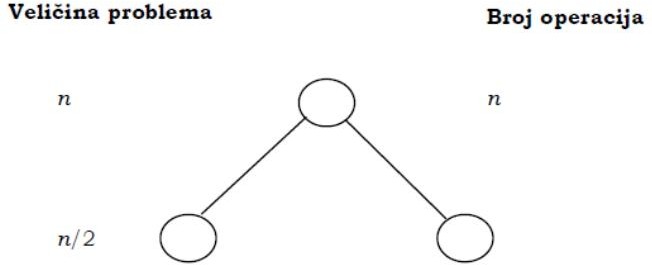
Izraz u smislu potrebnih operacija pri izvršavanju algoritamskog procesa interpretiramo na sljedeći način: „da bismo riješili problem veličine n moramo riješiti 2 problema veličine n/2 i dodatno još

izvršiti n operacija“. Na sličan način možemo interpretirati i sljedeći izraz.



„da bismo riješili problem veličine n, moramo riješiti 1 problem veličine n/4 i dodatno još izvršiti 𝑛 2

operacija“



Na slici je prikazan početni dio dijagrama stabla rekurzije za izraz (\*). S lijeve strane dijagrama je prikazana veličina problema, dok je s desne strane prikazan broj operacija.

Na nivou 0 je prikazano da je veličina problema n. Razbijanje početnog problema na dva

podproblema na dijagramu je prikazano pomoću dvije grane iz korijenskog čvora.

S desne strane je prikazano da je na nultom nivou potrebno još n dodatnih operacija, pored onih operacija koje će biti potrebno uraditi obzirom da je problem razbijen na dva dijela. Na taj način ćemo na dijagramu prikazati sve relevantne informacije na svakom nivou rekurzije.

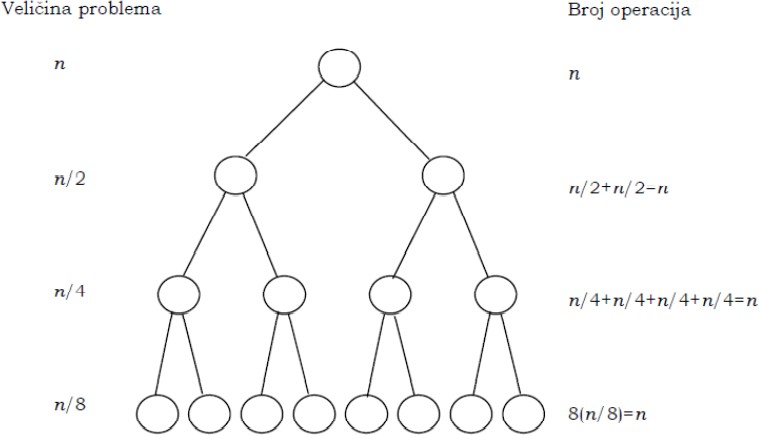
Dva nova podproblema na dijagramu su prikazana pomoću dva čvora na prvom nivou rekurzije, a s

lijeve strane je prikazano da svaki od problema ima veličinu n/2.

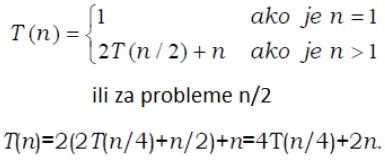
Dijagram stabla rekurzije nastavljamo crtati na isti način. Na sljedećoj slici su prikazana još dva nivoa.

Na dijagramu vidimo, na primjer, da je na nivou 1 potrebno ukupno n dodatnih operacija, jer imamo dva podproblema veličine n/2, od kojih svaki zahtijeva po n/2 dodatnih operacija. Slično, na nivou 2 se nalaze 4 podproblema veličine n/4, dok se na nivou 3 nalazi 8 podproblema veličine n/8.

Dakle, na svakom od nivoa rekurzije je potrebno n dodatnih operacija.



Da bismo ilustrirali na koji način se izraz (\*) reflektira na dijagramu stabla rekurzije, napišimo još jednom taj izraz uključujući i dio koji se odnosi na osnovni slučaj



Na slici na nivou 2 se nalaze 4 problema, od kojih svaki ima veličinu n/4, što zapravo predstavlja

rekurzivni član 4T(n/4) u izrazu sa prethodnog slida.

Međutim, nakon što se riješe podproblemi na nivou 2, vraćamo se na nivo 1 na kojem imamo 2 puta po n/2 dodatnih operacija, a zatim na nivo 1 gdje također imamo još n dodatnih operacija.

To znači, da nakon rješavanja podproblema na nivou 2, imamo još 2n dodatnih operacija na nivoima 1 i 0, što je u izrazu (donji \*) prikazano nerekurzivnim članom 2n. Prema tome, svaka primjena izraza (gornji \*) se na dijagramu stabla rekurzije reflektira kao dodavanje novog nivoa u to stablo.

Možemo reći da stablom rekurzije zapravo modeliramo proces ponavljanja rekurzivnih poziva,

odnosno proces razbijanja problema na manje podprobleme. To modeliranje radimo s ciljem

izračunavanja ukupnog broja potrebnih operacija.

Zadnji nivo stabla rekurzije predstavlja zadnju fazu ponavljanja rekurzivnog izraza i na tom nivou

imamo n problema, pri čemu je za rješavanje svakog od njih potrebno T(1)=1 operacija.

Ukupan broj operacija je jednak zbroju operacija na svim nivoima. Nakon što smo kreirali stablo,

tačno znamo koliko ima nivoa i koliko je potrebno operacija na svakom od njih. U našem slučaju ima

ukupno log2(n+1) nivoa, a na svakom nivou ima n operacija.

Prema tome, možemo zaključiti da je za rješavanje problema čija je vremenska složenost opisana

rekurzivnom relacijom

potrebno je n(log2n+1) operacija

Prema tome, pri analizi vremenske efikasnosti rekurzivnih algoritama potrebno je u obzir uzeti dva faktora.

Prvi faktor je dubina, odnosno broj nivoa na kojima se ostvaruju rekurzivni pozivi, prije nego se

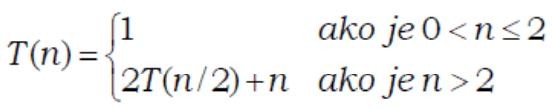
ispuni uvjet završetka rekurzije. Drugi faktor pri analizi efikasnosti koji treba uzeti u obzir su potrebni

vremenski resursi na svakom od niovoa rekurzije.

Naravno, što je veći nivo, veći je i broj stek-okvira (eng. stack frames) koji se moraju alocirati, a time je manja memorijska (prostorna) efikasnost algoritma. Isto tako, jasno je da rekurzivni pozivi do

većih nivoa zahtijevaju više vremena, što čini algoritam manje efikasnim i u vremenskom smislu.

## Rješavanje rekurzivnih relacija

Prethodno smo vidjeli da potrebno vrijeme za izvođenje nekog rekurzivnog algoritma modeliramo rekurzivnim relacijama. Na primjer, neka imamo sljedeću rekurzivnu relaciju:

Drugim riječima, za direktno rješavanje osnovnog slučaja (kada je n=1) je potrebna jedna operacija. Nadalje, broj operacija koje treba izvršiti algoritam za veličinu ulaza n (n > 1) je dva puta veći u odnosu na veličinu ulaza n/2, uvećan za dodatnih n operacija. Međutim, da bismo dobili vremensku složenost dotičnog algoritma, potrebno je riješiti rekurzivni gornji izraz, na način da se ukloni rekurzivna priroda tog izraza.

## Supstitucijska metoda

Ovaj pristup koristi tehniku procjene, kojom se pretpostavlja rješenje rekurzijske relacije (gornja i

donja granica), te se ispravnost te pretpostavke utvrđuje indukcijom.

Ako je dokaz pretpostavke uspješan, onda se pokušavaju suziti granice. Ako dokazivanje indukcijom ne uspije, onda se pokušava sa novim pretpostavkama. Ova tehnika je korisna kada tražimo samo složenost.

Međutim, ako želimo precizno rješenje u zatvorenoj formi, tada ova metoda nije prikladna

Ovu tehniku ćemo ilustrirati na primjeru pronalaženja asimptotskih granica za izraz \*. Pretpostavimo

da ova rekurzivna relacija ima gornju granicu O(𝑛 2 ). Odnosno, pretpostavimo sljedeće:

𝑇 𝑛 ≤ 𝑛 2

Ovu pretpostavku pokušavamo dokazati indukcijom. Radi pojednostavljenja ćemo pretpostaviti da je veličina ulaza n potencija broja 2.

Za osnovni slučaj imamo T (2) = 1 ≤ 2 2

U indukcijskom koraku, trebamo pokazati da je T (2n) ≤ (2n) 2 uz pretpostavku da je T (n) ≤ n 2 za n= 2

k , k≥1.

Indukcijska hipoteza je: T (i) ≤ i 2 , za sve i ≤ n

Slijedi T (2n ) = 2T (n ) + 2n ≤ (2n) 2 + 2n ≤ 4(n) 2

Prema tome, dokazali smo da je T(n) reda O(n 2 )

Međutim, postavlja se pitanje da li se mogla pretpostaviti neka funkcija sa manjim rastom u odnosu na funkciju n 2 . Na sličan način lako možemo utvrditi da npr. pretpostavka O(n) ne bi bila dobra.

Nadalje, zaključujemo da bismo trebali napraviti procjenu još sa nekom funkcijom koja ima rast

između cn i n 2 . Treba pokušati sa sljedećom pretpostavkom:

𝑇 𝑛 ≤ 𝑛 log 𝑛

Za osnovni slučaj

T (2) = 2 log2 2 = 2, metodom indukcije vrijedi za n

𝑇 (2𝑛) = 2𝑇 (𝑛) + 2𝑛 ≤ 2𝑛 log 𝑛 + 2𝑛 ≤ 2𝑛(log 𝑛 + 1) ≤ 2𝑛 log 2𝑛

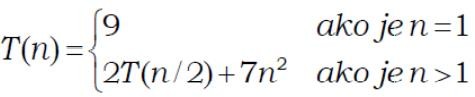
Prema tome, dokazano je da je T(n) reda O(n log n). Na sličan način se može dokazati da je T(n) reda Ω(n log n). Prema tome, T(n) je također reda Θ(n log n).

## Iteracijska metoda

Iteracijska metoda se sastoji u tome da razvijamo rekurzijsku relaciju, sve dok ne dobijemo osnovni

slučaj, za koji možemo primijeniti dati izraz.

Razvoj se sastoji u tome da se rekurzijski dio izraza s desne strane relacije zamjenjuje njegovom definicijom:

Potrebno je riješiti slijedeću relaciju:

Radi pojednostavljenja, opet ćemo pretpostaviti da je n potencija broja 2, n=2 𝑘 , k≥1. Prethodni izraz se može razviti na sljedeći način:



## Binarna stabla

**(Predavanje 10) Stabla – nastavak**

### Umetanje čvorova u binarno stablo pretraživanja

Pretpostavimo da imamo sljedeću listu brojeva: 15, 18, 8, 21, 3, 19, 13, 24 i 10. Ovu listu brojeva

ćemo ubaciti u binarno stablo pretraživanja, pri tome koristeći neki privremeni tekući pokazivač koji inicijalno pokazuje na korijen tog stabla, na sljedeći način:

Ako je tekući pokazivač 0, treba kreirati novi čvor u koji se upisuje informacioni sadržaj, te treba vratiti adresu novog čvora.

Ako nije, treba izvršiti usporedbu nove vrijednosti i čvora pohranjenog u tekućem čvoru. Ako je nova vrijednost manja od vrijednosti u tekućem čvoru, onda se ona treba umetnuti u lijevo podstablo

trenutnog čvora rekurzivnom primjenom istog algoritma. U suprotnom, nova vrijednost se umeće u desno podstablo trenutnog čvora.



### Umetanje čvora

U prvom koraku pokazivač na korijen je NIL, pa je prva vrijednost 15 umetnuta u prazno stablo (slika

a).

Sljedeća vrijednost 18 se uspoređuje sa vrijednošću u korijenu. Budući da je 18 veće od 15, vrijednost 18 je umetnuta kao desno dijete korijena (slika b). Na sličan način je umetnuta vrijednost 8 kao lijevo dijete korijena, imajući u vidu da je 8 manje od 15 (slika c).

Četvrta vrijednost 21 se uspoređuje sa vrijednošću 15 u korijenu, te pošto je 21 veće od 15, algoritam definira da se 21 treba umetnuti u desno dijete. Međutim, mjesto koje zauzima desno dijete je zauzeto, pa se algoritam rekurzivno primjenjuje od desnog djeteta korijena, tj. od korijena s vrijednošću 18.

Dakle, vrijednost 21 se uspoređuje sa 18, te se novi čvor umeće kao desno dijete jer je 21 veće od 18

(slika d).

Na sličan način se umeće vrijednost 3, koja se prvo uspoređuje sa vrijednošću 15 u korijenu, te pošto je 3 manje od 15, algoritam definira da se 3 treba umetnuti u lijevo dijete. Mjesto koje zauzima lijevo dijete je zauzeto, pa se algoritam sada rekurzivno primjenjuje od lijevog djeteta, tj. od korijena s

vrijednošću 8.

Prema tome, vrijednost 3 se uspoređuje sa 8, te se novi čvor umeće kao lijevo dijete, jer je 3 manje

od 8 (slika e).

## Procedura umetanja novog čvora rekurzivnom metodom

Opisani postupak umetanja novog čvora je opisan rekurzivnom procedurom UMETNI-BST-R.

Ova procedura umeće novi čvor sa adresom z u stablo sa korijenom na kojeg pokazuje pokazivač

korijen

**UMETNI-BST-R (korijen, z)**

**1 p <-- korijen**

**2 if (p = NIL) then**

**3 korijen <--z**

**4 else**

**5 if (kljuc(2) < kljucp)) then**

**6 UMETNI-BST-R (lijevi(p), z)**

**7 else**

**8 UMETNI-BST-R (desnilp), z)**

**9 end\_if**

**end\_if**

**UMETNI-BST-I (korijen, z)**

**1 y<-- NIL**

**2 x<-- korijen**

**3 while (x != NIL) do**

**4 y<-- X**

**5 if (kljuc(2) < kljuc(x) then**

**6 X<-- lijevi(x)**

**7 else**

**8 x <-- desni(x)**

**9 end\_if**

**end\_while**

**if (y = NIL) then**

**korijen - 2**

**else**

**if (kljuc(z) < kljuc(y)) then**

**lijevi(y)<-- z**

**else**

**desni(y) <--z**

**end\_if**

**19end\_if**

## Procedura umetanja novog čvora iterativno

Argumenti procedure su pokazivač na korijen stabla (korijen) u koje se umeće novi čvor, te pokazivač na novi čvor (z).

Pokazivač x prati putanju, dok pokazivač y pokazuje na roditelja od x. Put pretraživanja započinje od korijena (linija 2), a zatim se u while petlji,

ažuriranjem pokazivača x, usmjerava put

pretraživanja sve dok pokazivač x ne bude imao vrijednost NIL, odnosno dok se ne pronađe odgovarajuće mjesto za novi čvor (linije 3-10).

Pronalaženje tog mjesta se usmjerava na temelju

poređenja ključeva kljuc(x) i kljuc(z).

Nakon izlaska iz while petlje, pokazivač y pokazuje na čvor koji treba da bude roditelj novog čvora z.

Ako roditelj y ima vrijednost y=NIL, tada se zapravo u prazno stablo umeće prvi čvor, pa je potrebno

ažurirati pokazivač korijen (linije 11-12).

Inače, ako je ključ novog čvora z manji od njegovog roditelja, tada se novi čvor umeće kao lijevo

dijete (linije 14-15). U suprotnom, novi čvor se umeće kao desno dijete (linija 17).

## Brisanje čvorova iz binarnog stabla pretraživanja

Operaciju brisanja bit će opisana kroz tri različite situacije:

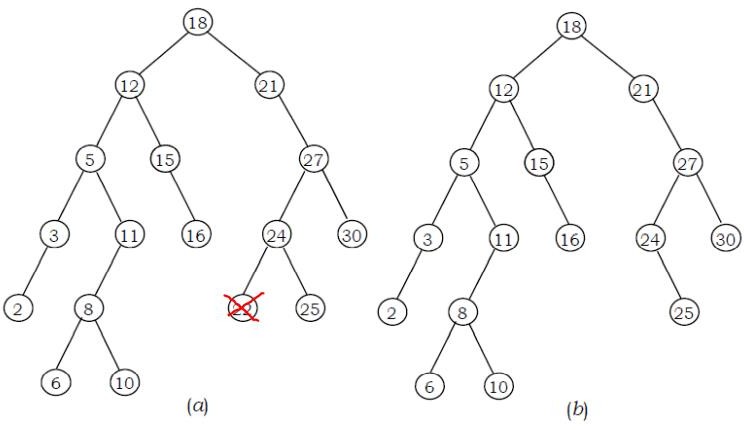
Brisanje jednog lista

Brisanje roditelja koji ima jedno dijete

Brisanje roditelja koji ima i lijevo i desno podstablo

### Brisanje čvora

Tada je potrebno roditelju čvora koji se briše postaviti odgovarajući pokazivač na vrijednost NIL.

Ova situacija je ilustrirana tako što je na slici a prikazano početno stanje stabla, a na slici b je prikazano stanje stabla nakon brisanja čvora 22

### Brisanje čvora sa jednim djetetom

Druga situacija je kada roditelj čvora koji se briše ima samo jedno dijete. U tom slučaju dijete zamjenjuje čvor koji se

briše, tako što roditelj dotičnog čvora pokazuje na dijete tog čvora.

Ova situacija je ilustrirana primjerom brisanja čvora 21 iz stabla prikazanog na slici b. Rezultirajuće stablo je prikazano na slici c.

Čvor 27 (dijete čvora koji se briše) je jedino dijete čvora 21, pa čvor 27 postaje dijete čvora 18 (roditelj čvora koji se

briše).

### Brisanje čvora sa dva djeteta

Najsloženija situacija je slučaj brisanja čvora koji ima i lijevo i desno podstablo. U ovom slučaju je potrebno u određenoj mjeri preurediti stablo na način da se čvor koji se briše zamijeni najdesnijim čvorom u njegovom lijevom podstablu.

Najdesniji čvor može imati lijevo podstablo, a sigurno nema desno podstablo. Roditelj najdesnijeg čvora kao svoje desno podstablo preuzima lijevo podstablo najdesnijeg čvora. Pronađeni najdesniji čvor preuzima oba podstabla obrisanog čvora.

Ovakva situacija je ilustrirana primjerom brisanja čvora 12 iz stabla prikazanog na slici c.

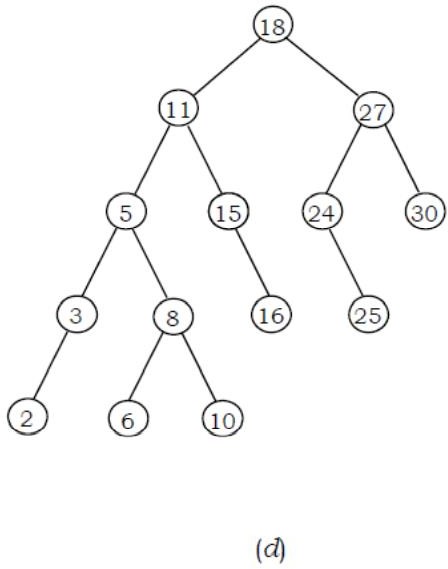
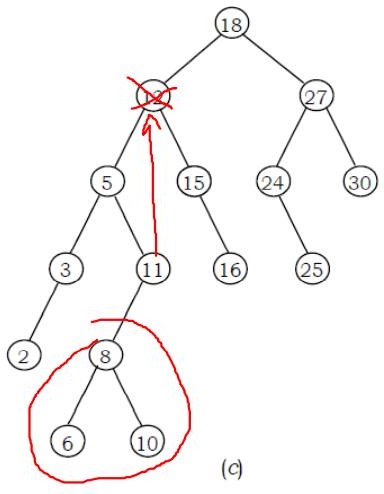
Rezultirajuće stablo je prikazano na slici d.

Najdesniji čvor u lijevom podstablu čvora 12 je čvor 11. To znači da čvor 11 treba zamijeniti čvor 12 koji se briše, tako što roditelj obrisanog čvora (čvor 18) pokazuje na pronađeni najdesniji čvor.

Roditelj čvora 11 (čvor 5) kao svoje desno podstablo preuzima lijevo podstablo čvora 11, čiji je korijen čvor 8.

Čvor 11 preuzima oba podstabla obrisanog čvora 12. Napomenimo još da smo umjesto pronalaženja najdesnijeg čvora u lijevom podstablu mogli isto tako pronaći najljevlji čvor u desnom podstablu

čvora koji se briše i primijeniti sličan postupak preuređenja stabla, kao što je to prethodno opisano.



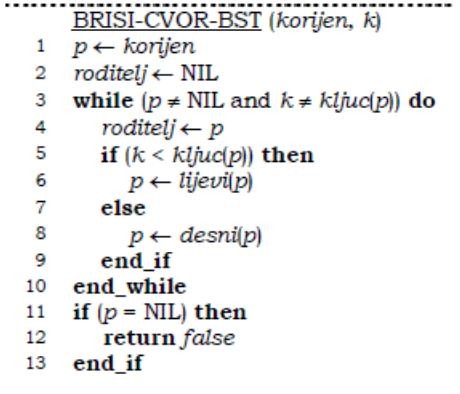
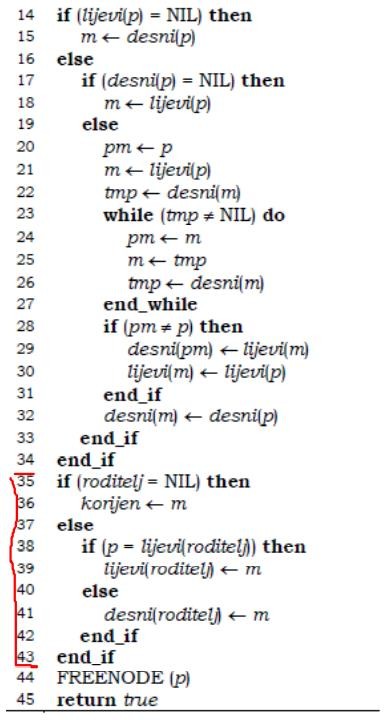
### Brisanje čvora sa jednim djetetom

Algoritam brisanja čvora je opisan procedurom BRISI-CVOR-BST

Ulazni argumenti procedure su pokazivač na korijen stabla korijen i vrijednost ključa k u čvoru koji se briše.

Na početku algoritma se u prvoj while petlji, počevši od korijena, pronalaze pokazivač p na čvor koji se briše i pokazivač roditelj na roditelja tog čvora, ako se ne briše korijen stabla (linije 1-10).

Ako se u stablu ne pronađe čvor sa ključem k, pokazivač p ima vrijednost NIL, pa funkcija vraća logičku vrijednost false (linije 11-13).



U nastavku algoritma se obrađuje situacija kada čvor koji se briše ima samo jedno ili nijedno dijete (linije 14- 18). Ako čvor koji se briše ima samo jedno dijete, tada pokazivač m pokazuje na to dijete.

Inače, ako čvor koji se briše ima oba djeteta, tada se u lijevom podstablu pronalazi najdesniji čvor na kojeg pokazuje pokazivač m, a pokazivač pm pokazuje na njegovog roditelja (linije 19-27).

Ako čvor koji se briše nije jednak roditelju

najdesnijeg čvora u njegovom lijevom podstablu, tada roditelj najdesnijeg čvora, na kojeg pokazuje pokazivač pm, preuzima njegovo lijevo podstablo, dok najdesniji čvor preuzima i lijevo i desno podstablo čvora koji se briše (linije 28-32).

Ako je pak čvor koji se briše istovremeno roditelj najdesnijeg čvora u njegovom lijevom podstablu, tada je samo potrebno da najdesniji čvor na kojeg pokazuje m preuzme desno podstablo čvora koji se briše (linija 32). • Ako se briše čvor u korijenu, tada je potrebno ažurirati pokazivač korijen, tako da pokazuje na najdesniji čvor (linije 35-36).

U suprotnom, ako se ne briše čvor u korijenu, potrebno je da roditelj čvora koji se briše

pokazuje na čvor koji se premješta, na kojeg

pokazuje m (linije 38-42).

Na kraju se oslobađa memorijski prostor koji je zauzimao čvor koji se briše i funkcija vraća logičku vrijednost true kao znak uspješno obavljene operacije (linije 44-45).

Operacija brisanja u stablu visine h ima

vremensku složenost O(h).

**BRISI-CVOR-BST(korijen, k)**

**1 p <-- korijen**

**2 roditelj <-- NIL**

**3 while (p != NIL and k != kljuc(p)) do**

**4 roditelj <-- p**

**5 if(k < kljuc(p)) then**

**6 p <-- lijevi(p)**

**7 else**

**8 p <-- desni(p)**

**9 end\_if**

**10 end\_while**

**if(p = NIL) then**

**return false**

**end\_if**

**if(lijevi(p) = NIL) then**

**m <-- desni(p)**

**else**

**if(desni(p) = NIL) then**

**m <-- lijevi(p)**

**else**

**pm <-- p**

**m <-- lijevi(p)**

**tmp <-- desni(m)**

**while(tmp != NIL) do**

**pm <-- m**

**m <-- tmp**

**tmp <-- desni(m)**

**end\_while**

**if(pm != p) then**

**desni(pm)<--lijevi(m)**

**lijevi(m)<--lijevi(p)**

**end\_if**

**desni(m)<--desni(p)**

**end\_if**

**end\_if**

**if(roditelj = NIL) then**

**korijen<--m**

**else**

**if(p = lijevi(roditelj))then**

**lijevi(roditelj)<--m**

**else**

**desni(roditelj)<--m**

**end\_if**

**end\_if**

**FREENODE(p)**

**45 return true**

## Operacija obilaska čvorova binarnog stabla

Kada je potrebno pristupiti svim čvorovima nekog binarnog stabla s ciljem da se izvrši željena obrada informacija pohranjenih u svakom pojedinom čvoru, tada je potrebno na određeni sistematičan način posjetiti (obići) sve čvorove

Posjećivanje svih čvorova nekog stabla nije tako jednostavna operacija, kao što je to slučaj kod

linearnih struktura podataka (povezane liste, nizovi, itd.). Najčešće se primjenjuju tri uobičajene

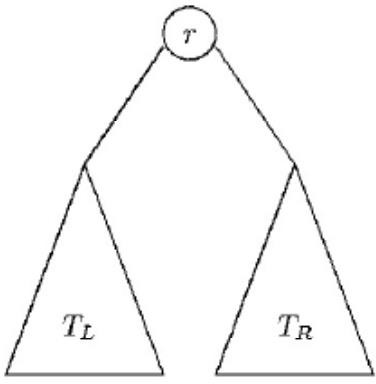
metode obilaska:

preorder, inorder i postorder.

### Obilazak

Neka je binarno stablo T sastavljeno od korijena r, te neka ostali čvorovi grade uređeni par (TL, TR)

manjih binarnih stabala, gdje stabla TL i TR predstavljaju lijevo i desno podstablo, respektivno



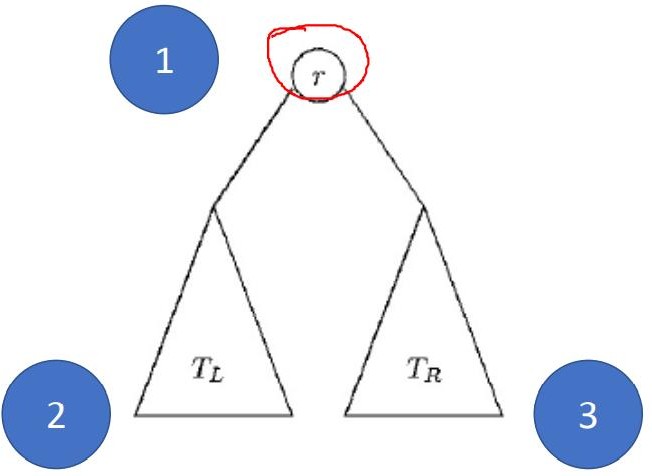
### Preorder

Preorder obilazak je definiran na sljedeći način:

Prvo se posjećuje čvor r (korijen).

Obilazi se lijevo podstablo TL na preorder način.

Obilazi se desno podstablo TR na preorder način



### Procedura preorder

Implementacija načina obilaska preorder je prikazana rekurzivnom procedurom PREORDER.

Procedura POSJETI predstavlja proceduru koja posjećuje čvor r koja eventualno može sadržavati i dodatnu obradu nad podacima pohranjenim u tom čvoru

Svakim obilaskom stabla se kreira odgovarajući linearni poredak čvorova

Primjer slika

**PREORDER(r)**

**1 If(r != NIL) then**

**2 POSJETI(r)**

**3 PREORDER(lijevi(r))**

**4 PREORDER(desni(r))**

**5 end\_if**



### Procedura inorder

Inorder obilazak je definiran na sljedeći način:

Obilazi se lijevo podstablo TL na inorder način.

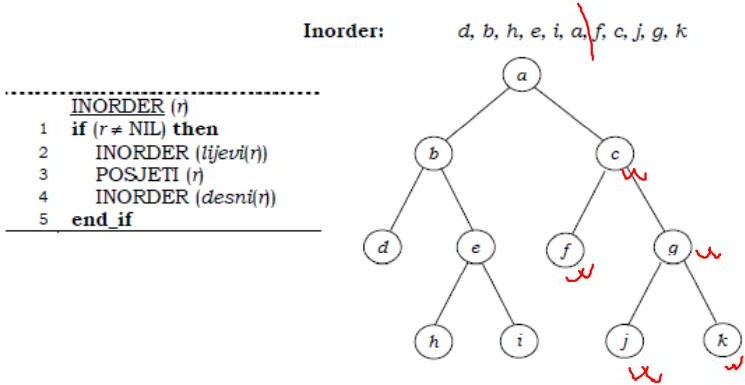
Posjećuje se čvor r (korijen).

Obilazi se desno podstablo TR na inorder način

Implementacija načina obilaska inorder je prikazana rekurzivnom procedurom INORDER

Obilaskom inorder se također dobiva određeni linearni poredak elemenata. Na primjer, za stablo

prikazano na slici d, b, h, e, i, a, f, c, j, g, k.



**INORDER(r)**

**1 If(r != NIL)then**

**2 INORDER(lijevi(R))**

**3 POSJETI(r)**

**4 INORDER(desni(r))**

**5 end\_if**

### Procedura postorder

Postorder obilazak je definiran na sljedeći način:

Obilazi se lijevo podstablo TL na postorder način.

Obilazi se desno podstablo TR na postorder način.

Posjećuje se čvor r (korijen).

Implementacija načina obilaska postorder je prikazana rekurzivnom procedurom POSTORDER

Na primjer, za stablo prikazano na slici, obilaskom na postorder način se dobiva sljedeći poredak čvorova: d, h, i, e, b, f, j, k, g, c, a.

**POSTORDER(r)**

**1 If(r!=NIL)then**

**2 POSTORDER(lijevi(r))**

**3 POSTORDER(desni(r))**

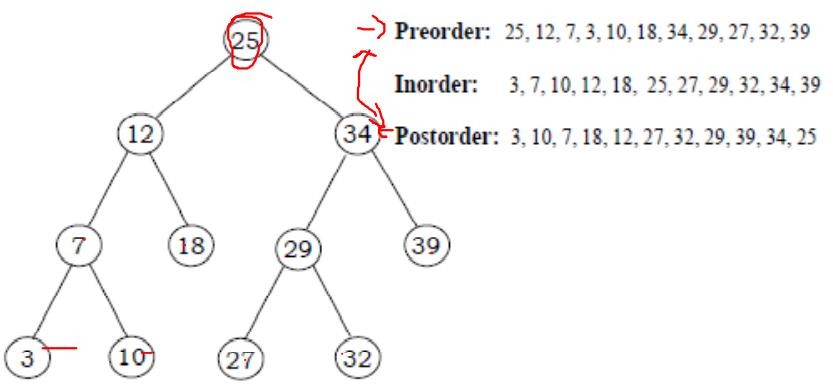
**4 POSJETI(r)**

**5 end\_if**

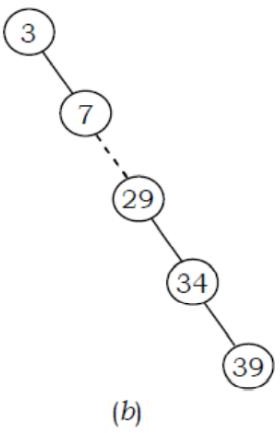
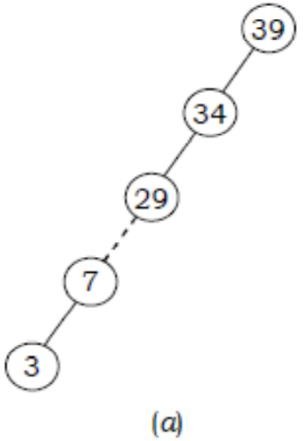


### Procedura inorder

Važno je istaknuti da inorder obilazak generira sortirani rastući poredak elemenata, ako se obilazi binarno stablo pretraživanja.

Na primjer, za binarno stablo pretraživanja prikazano na slici, obilazak na inorder način generira sljedeći poredak elemenata: 3, 7, 10, 12, 18, 25, 27, 29, 32, 34, 39.

## Balansiranje stabla



Sa aspekta efikasnosti izvođenja pojedinih operacija različite topologije stabala imaju i različitu efikasnost.

Općenito govoreći, poželjno je da za dati broj čvorova stablo ima što manju visinu sa što više grananja. Međutim, u situacijama kada stablo ima veliku visinu i malo grananje, raspored čvorova u binarnom stablu je nepovoljan.

Na primjer, ako bi se u stablo dodavali čvorovi sa ključevima u opadajućem poretku, tada bi svi čvorovi imali samo lijevu granu, kao što je prikazano stablom na slici a, koje nastaje kao rezultat umetanja ključeva sljedećim redoslijedom: 39, 34, 29, ..., 7, 3.

Slična topologija stabla bi se pojavila ako bismo u stablo dodavali čvorove sa ključevima u rastućem poretku.

Tada bi stablo imalo samo desnu granu, što je ilustrirano stablom prikazanim na slici b, koje nastaje kao rezultat umetanja ključeva sljedećim redoslijedom 3, 7, ... 29, 34, 39.

Složenost algoritma je O(n).

Potrebna balansiranost stabla

Jedna od najvažnijih vrsta balansiranih stabala su stabla balansirana po visini.

### Balansiranje stabla po visini (AVL- Adelson-Velsky i Landis)

Kao kriterij balansiranosti stabala uzeta je razlika u visini podstabala. Za neki čvor X definirajmo

balans bx na sljedeći način:

bx=hl-hr

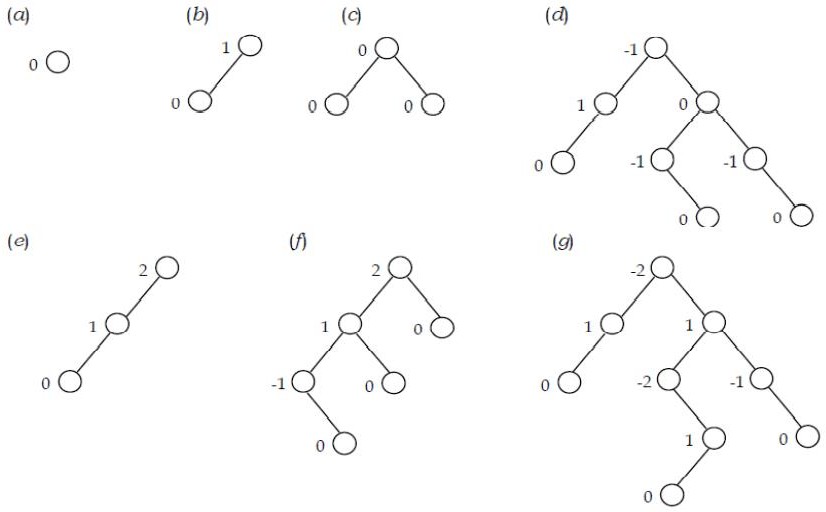
hl – visina lijevog podstabla, hr – visina desnog postabla

AVL stablo je stablo kod kojeg za svaki čvor X vrijedi da njegov balans bx ima jednu od sljedećih

vrijednosti: -1, 0 ili 1.

Dakle, kod AVL stabla je razlika visine lijevog podstabla i desnog podstabla najviše 1

a-d AVL

e-g obična stabla

## AVL stabla

### Operacije umetanja u AVL stablo

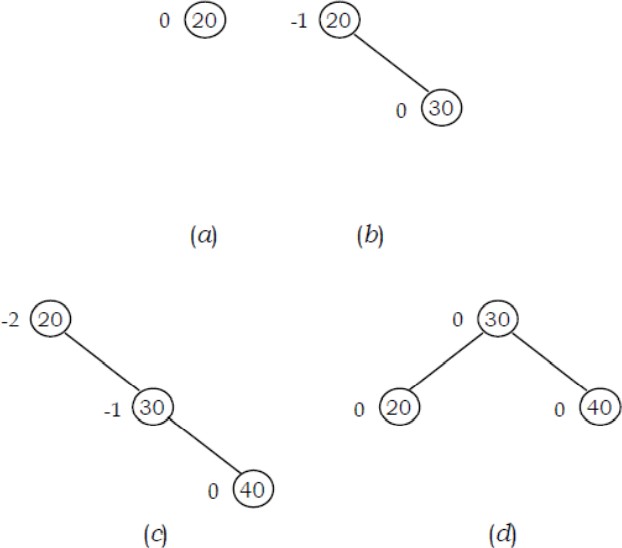
Počinjemo sa umetanjem čvora 20 u korijen (slika a). Budući da ovaj čvor nema djecu, njegov balans je 0, te stablo koje trenutno ima samo jedan čvor zadovoljava uvjete AVL stabla.

Sada dodajemo sljedeći čvor 30, što je prikazano na slici b. Čvor 30 ima balans 0, a čvor 20 sada ima

balans -1.

Sljedeći čvor kojeg dodajemo je čvor 40 (slika c).

Počevši od tog čvora, izračunavamo balans čvorova na putu prema korijenu. Pošto je čvor 40 terminalni čvor, njegov balans je 0.

Čvor 30 ima balans -1, što ne narušava uvjete AVL stabla. Međutim, čvor 20, koji ima balans -2, je čvor koji narušava uvjete AVL stabla, pa je potrebno preurediti stablo na način da se obnovi balansiranost stabla.

## Balansiranost stabla

Operacija kojom se obnavlja balansiranost stabla se naziva rotacija. Postoje dva tipa rotacije kod AVL stabala: jednostruka i dvostruka rotacija. Da bismo odlučili koju rotaciju treba primijeniti koriste se sljedeća pravila:

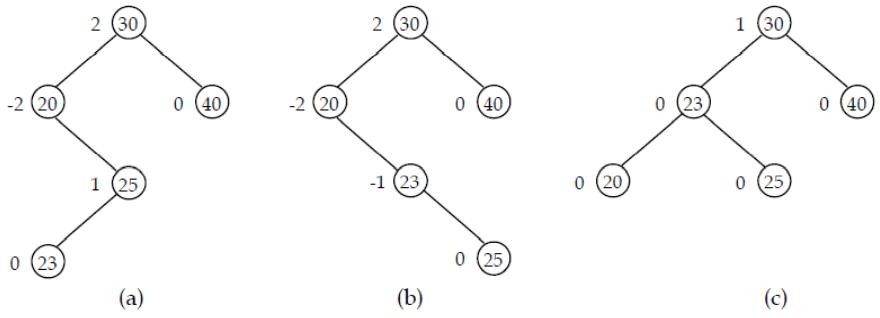
Krećemo od novog dodanog čvora u stablo, slijedeći put umetanja prema korijenu I provjeravamo balans čvorova na tom putu. Kada pronađemo kritični čvor kod kojeg je narušen balans, potrebno je utvrditi da li kritični čvor i njegovo dijete na putu umetanja naginju na istu stranu.

Ako kritični čvor i njegovo dijete na putu umetanja naginju na istu stranu, za obnavljanje balansiranosti stabla je potrebno primijeniti jednostruku rotaciju. Lijeva rotacija se primjenjuje ako čvorovi naginju na desnu stranu, dok se desna rotacija primjenjuje ako čvorovi naginju na lijevu stranu.

Ako kritični čvor i njegovo dijete na putu umetanja naginju na različite strane, za obnavljanje

balansiranosti je potrebno primijeniti dvostruku rotaciju

### Operacije umetanja u AVL stablo

Proces dodavanja novih čvorova nastavljamo sa dodavanjem čvorova 25 i 23. Nakon dodavanja čvora 25, nije došlo do narušavanja uvjeta AVL stabla. Međutim, dodavanjem čvora 23 dolazi do narušavanja balansiranosti. Izgled stabla nakon dodavanja čvorova 25 i 23 je prikazan na slici a

.

Utvrđujemo na koju stranu naginje kritični čvor (čvor 20) i njegovo desno dijete. Možemo vidjeti da čvor 20 naginje na desnu stranu, dok čvor 25 naginje na lijevu stranu. Prema tome, potrebno je primijeniti dvostruku rotaciju da bi se obnovila balansiranost stabla. Dvostruka rotacija se sastoji od dvije jednostruke rotacije u suprotnim smjerovima. Prva rotacija se obavlja na dva nivoa ispod

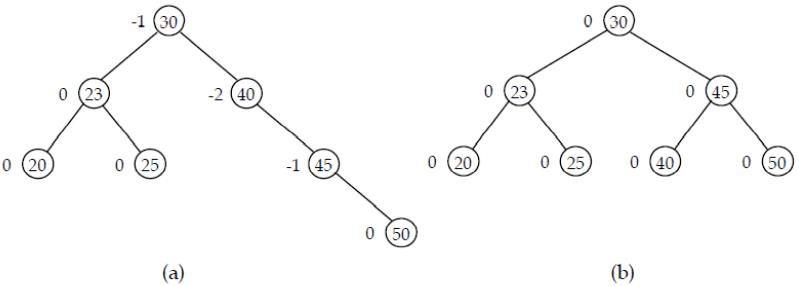
kritičnog čvora. U našem primjeru ta rotacija uključuje čvorove 25 i 23. Čvor 23 se rotira tako da zamjenjuje čvor 25, a čvor 25 postaje desno dijete čvora 23. Nakon ove prve rotacije stablo ima izgled kao na slici b.

Na toj slici možemo vidjeti da prva rotacija nije obnovila balansiranost stabla, pa je potrebno

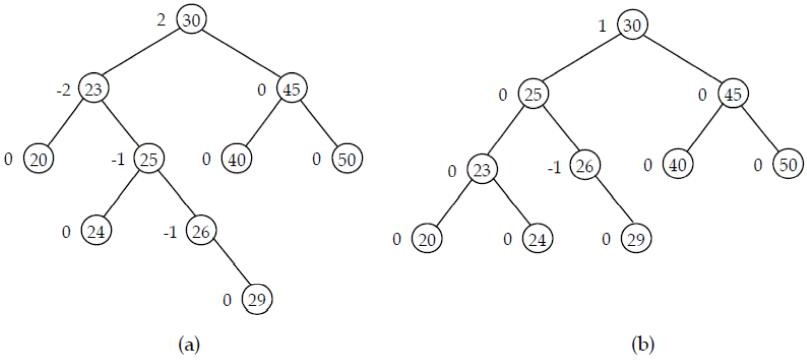
primijeniti još jednu rotaciju, koja uključuje čvorove: 25, 23 i 20. Čvor 23 se rotira tako da zamjenuje čvor 20, dok se čvor 20 rotira na način da postaje lijevo dijete čvora 23. Rezultirajuće stablo, nakon ove druge rotacije, je prikazano na slici c. Možemo vidjeti da je kod rezultirajućeg stabla obnovljena balansiranost, jer nijedan od čvorova u stablu nema apsolutnu vrijednost balansa veću od 1

Proces dodavanja novih čvorova u stablo nastavljamo sa dodavanjem čvorova 45 i 50. Dodavanje čvora 45 neće narušiti balansiranost stabla, za razliku od dodavanja čvora 50. Rezultirajuće stablo nakon dodavanja ova dva čvora je prikazano na slici a.

Prvi kritični čvor koji narušava balansiranost na putu umetanja novog čvora je čvor 40. Ponovno

utvrđujemo na koju stranu naginju kritični čvor i njegovo desno dijete, te zaključujemo da oba čvora naginju na desnu stranu. Prema tome, za obnavljanje balansiranosti stabla je potrebno napraviti jednostruku rotaciju. Rezultirajuće balansirano stablo nakon rotacije je prikazano na slici b

I na kraju, dodajmo čvorove 26, 24 i 29. Dodavanje čvorova 26 i 24 neće narušiti balansiranost stabla, za razliku od dodavanja čvora 29. Rezultirajuće stablo nakon dodavanja ovih čvorova je prikazano na slici a.

Počevši od novog čvora 29, vraćamo se nazad putem umetanja tog čvora i nailazimo na čvor 23, kao prvi kritični čvor koji narušava balansiranost stabla. Opet razmatramo na koju stranu naginju kritični čvor i njegovo desno dijete

Počevši od novog čvora 29, vraćamo se nazad putem umetanja tog čvora i nailazimo na čvor 23, kao prvi kritični čvor koji narušava balansiranost stabla. Opet razmatramo na koju stranu naginju kritični čvor i njegovo desno dijete. Vidimo da oba čvora naginju na desnu stranu, te zaključujemo da je za obnavljanje balansiranosti stabla potrebno napraviti jednostruku lijevu rotaciju.

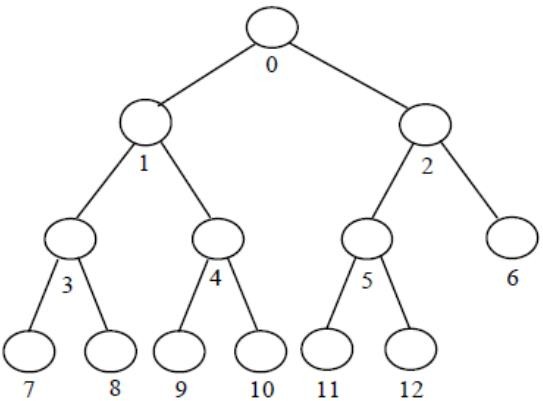
Čvor 25 se rotira tako da zamjenjuje čvor 23, dok se čvor 23 rotira tako da postaje lijevo dijete čvora

Nadalje, bitno je uočiti kako se pozicionira i čvor 24, koji je prethodno bio lijevo dijete čvora 25. Sada čvor 24 postaje novo desno dijete čvora 23. Primjenom prethodno opisanih izmjena, dobija se balansirano stablo kao na slici b

# (Predavanje 11) Heap

## Implementacija gomile pomoću niza i njena svojstva

Na početku razmotrimo kako se binarno stablo može prikazati pomoću niza.

Na slici čvorovi stabla su numerirani, tako da je korijenu pridružena vrijednost 0, a ostale vrijednosti (1, 2, 3, itd.) su pridružene čvorovima, tako da se poštuje poredak čvorova po nivoima, kao i poredak koji se dobiva kretanjem od lijeve prema desnoj strani na svakom nivou

Ako koristimo broj čvora kao indeks niza, onda nam ovaj pristup može dati redoslijed po kojem pohranjujemo čvorove stabla u niz. Stablo se može lako rekonstruirati iz niza na sljedeći način:

Lijevo dijete čvora numeriranog kao k ima indeks 2k+1. Na primjer, lijevo dijete čvora numeriranog

kao 5 ima indeks 11.

Desno dijete čvora numeriranog kao k ima indeks 2k+2. Na primjer, desno dijete čvora numeriranog

kao 3 ima indeks 8

Prema tome, korijen je smješten na poziciju 0, dvoje djece korijena je smješteno na pozicijama 1 i 2. Nadalje, djeca čvora 1 su smještena na pozicijama 3 i 4, djeca čvora 2 su smještena na pozicijama 5 i 6, itd

Bilo koje binarno stablo se može implementirati pomoću niza na opisani način, ako ostavimo praznine za one elemente za koje ne postoje odgovarajući čvorovi u stablu. Ovakav način

implementacije binarnog stabla je neefikasan, jer će u nizu ostati određeni broj praznih mjesta neiskorišten. Međutim, postoji vrsta binarnog stabla koja ima upravo takva svojstva, koja ga čine

izuzetno pogodnim za implementaciju pomoću nizova. Takva vrsta binarnog stabla se naziva gomila

(eng. heap).

Gomila je binarno stablo koje ima sljedeća svojstva:

Svi listovi su na dva susjedna nivoa

Svi listovi na najnižem nivou su u lijevom dijelu stabla.

Svi listovi iznad najnižeg nivoa su popunjeni

Oboje djece bilo kojeg čvora su ponovno gomile.

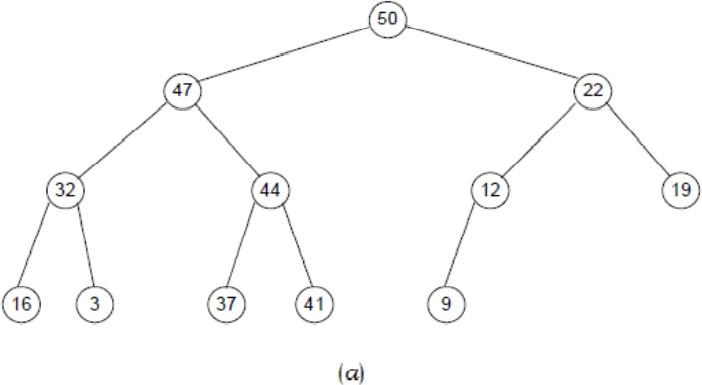
Vrijednost pohranjena u bilo kojem čvoru je barem toliko velika kao i vrijednosti pohranjene u čvorovima koji predstavljaju djecu dotičnog čvora.

Primijetimo da stablo na slici a ima najveći element (50) smješten u korijenu, a da su elementi smješteni u svakom od čvorova veći od elemenata smještenih u djeci dotičnog čvora.

Listovi gomile se nalaze na susjednim nivoima (treći i četvrti).

Čvorovi na najnižem nivou popunjavaju taj nivo s lijeve strane. Ako bismo uklonili npr. čvor 9, stablo bi opet zadržalo svojstva gomile. S druge strane, ako bismo uklonili bilo koji drugi čvor sa zadnjeg

nivoa, bilo bi narušeno svojstvo gomile označeno kao 2



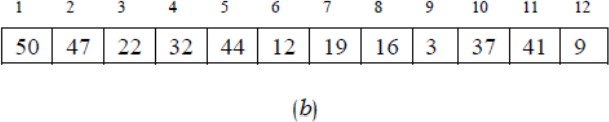
Gomila prikazana na slici a se može implementirati pomoću niza A, koji je prikazan na slici b.

Primijetite da redoslijed elemenata u nizu ima određeni stepen uređenosti po opadajućem poretku.

Niz A koji predstavlja gomilu je objekat koji ima dva atributa:

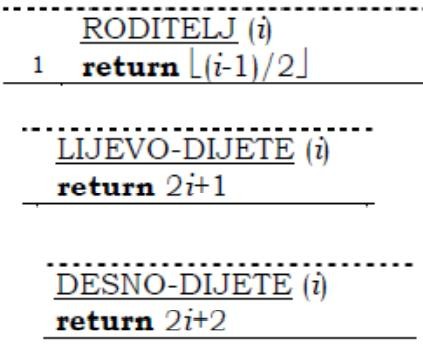
duzina[A] predstavlja broj elemenata u nizu;

velicina[A] predstavlja broj elemenata u nizu koji čine gomilu



Drugim riječima, mada niz A[0..duzina[A]-1] fizički može sadržavati određene vrijednosti, elementi koji se nalaze iza A[velicina[A]] u logičkom smislu ne pripadaju gomili.

Na primjer ako atributi gomile A imaju sljedeće vrijednosti: duzina[A]=12 i velicina[A]=5 – to znači da gomila A sadrži sljedeće elemente:

A[0..velicina[A]-1] = {50, 47, 22, 32, 44}, bez obzira što su u nizu prisutni i drugi elementi na pozicijama većim ili jednakim od velicina[A]

Budući da će opis tipičnih operacija u radu sa gomilom

zahtijevati često pristup roditelju nekog čvora, kao i njegovoj djeci uvest ćemo tri sljedeće funkcije:

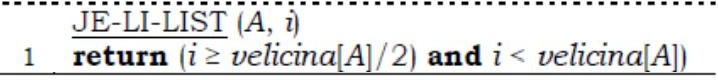
RODITELJ (i) vraća indeks roditelja čvora i;

LIJEVO-DIJETE (i) vraća indeks lijevog djeteta čvora i;

DESNO-DIJETE (i) vraća indeks desnog djeteta čvora i

Nadalje, za utvrđivanje da li je neki čvor list, koristit ćemo također vrlo jednostavnu funkciju JE-LI- LIST.

Funkcija vraća true ako je čvor s indeksom i list, odnosno vrijednost false ako dotični čvor nije list.



Dakako, na sličan način kako je implementirana gomila sa maksimalnim elementom u korijenu, može se na analogan način realizirati i gomila sa minimalnim elementom u korijenu.

Prema tome, postoje dvije vrste gomile:

gomile s maksimalnim elementom u korijenu;

gomile s minimalnim elementom u korijenu.

Kod obje vrste vrijednosti u čvorovima zadovoljavaju tzv. svojstvo poretka, koje je zapravo već prethodno navedeno kao peto, pri nabrajanju i ostalih svojstava gomile. To svojstvo kod gomile s maksimalnim elementom u korijenu je:

A[RODITELJ(i)] => A[i].

Dakle, svojstvo poretka definira da su vrijednosti u roditeljskim čvorovima veće ili jednake od vrijednosti u čvorovima koji su njihova djeca. Prema tome, najveća vrijednost je pohranjena u korijenu.

Gomila s minimalnim elementom u korijenu je organizirana na suprotan način. Svojstvo poretka

takve gomile je:

A[RODITELJ(i)] <= A[i]

U daljem izlaganju gomilu ćemo podrazumijevati kao niz max. Elemenata bez gubitka općenitosti

## Popravljanje gomile

Za proces izgradnje gomile jedna od ključnih procedura predstavlja postupak spuštanja nekog izabranog čvora u gomili, s ciljem pronalaženja njegove prihvatljive pozicije da bi se održalo potrebno svojstvo poretka.

Taj postupak je opisan procedurom POPRAVI-DOLJE

**POPRAVI-DOLJE(A, i)**

**1 While(JE-LI-LISTA(A, i) != true)do**

**2 veci** 🡨 **LIJEVO-DIJETE(i)**

**3 dd** 🡨 **DESNO-DIJETE(i)**

**4 if(dd<velicina[A] and A[dd] > A[veci])then**

**5 veci**🡨**dd**

**6 end\_if**

**7 if(A[i] > A[veci])then**

**8 return**

**9 end\_if**

**10 A[i]<->A[veci]**

**11 i**🡨**veci**

**12end\_while**

Ulaz u proceduru su niz A i indeks i. Prije poziva procedure pretpostavljamo da su lijevo i desno podstablo čvora i gomile, a nakon izvođenja procedure podstablo koje ima korijen smješten na poziciji s indeksom i također postaje gomila.

Postupak spuštanja čvora se izvršava u petlji while, tako da u pojedinim iteracijama varijabla veci poprima vrijednost indeksa većeg djeteta čvora i.

Ako je element A[i] veći od svoje djece, što se utvrđuje njegovim poređenjem sa većim djetetom

A[veci], tada se procedura prekida, jer ostatak stabla sigurno ima svojstvo poretka (linije 7-9).

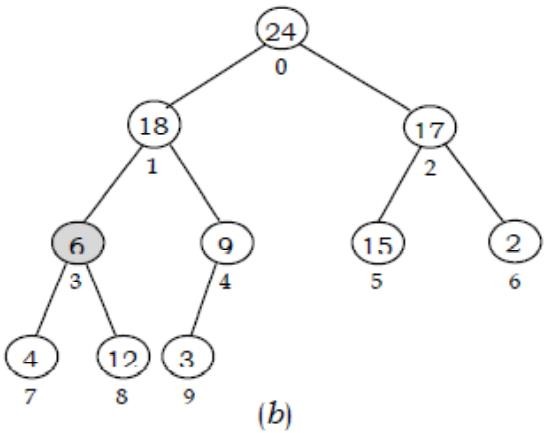
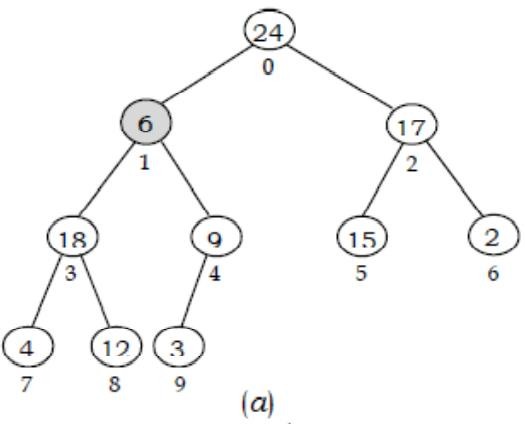
U suprotnom, elementi niza A sa indeksima i i veci izmjenjuju pozicije, te indeks i poprima vrijednost

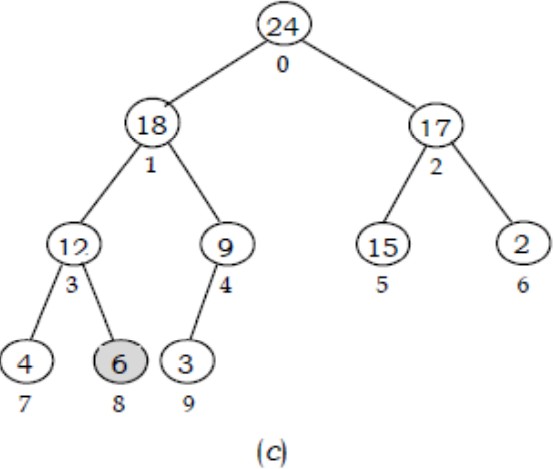
većeg djeteta veci (linije 10-11).

Opisani postupak se nastavlja dalje prema listovima stabla

Na slici je ilustriran način rada procedure POPRAVI-DOLJE, za primjer gomile prikazane pomoću niza A veličine 10.

Na slici a je prikazana početna konfiguracija gomile A, pri čemu element A[1]=6 narušava svojstvo poretka, jer taj element nije veći od svoje djece

Pozivom procedure POPRAVIDOLJE (A, 1) se popravlja gomila, tako da u prvom koraku dolazi do izmjene pozicija elemenata A[1] i A[3] , jer je veće dijete čvora s indeksom 1 pronađeno na poziciji 3 (slika b)

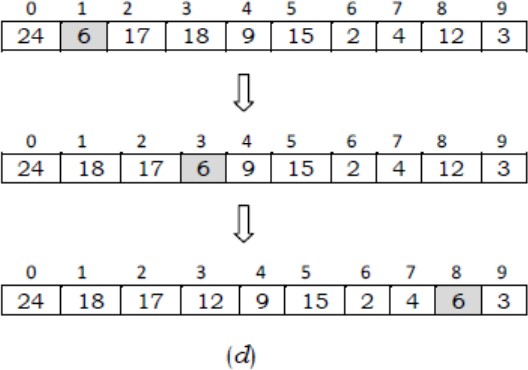
Varijabla i poprima vrijednost i=3, pa se postupak nastavlja sa podstablom koje ima korijen u čvoru A[3].

Prethodnom izmjenom je narušeno svojstvo poretka za podstablo sa korijenom A[3], pa u sljedećoj iteraciji dolazi do izmjene pozicija elemenata A[3] i A[8], jer je veće dijete čvora A[3] pronađeno na poziciji 8 (slika c).

Varijabla i poprima vrijednost i=8, pa u sljedećem koraku poziv procedure JE-LI-LIST(A, 8) daje

vrijednost true, te se izvođenje petlje while prekida.

Na slici d je ilustriran sadržaj niza A za konfiguracije gomila prikazanih na slikama a-c



## Stvaranje gomile

Proces stvaranja gomile je proces kojim neki niz A[0..duzina[A]-1] pretvaramo u gomilu A[0..velicina[A]-1], gdje je velicina[A]=duzina[A].

U tom procesu možemo koristiti proceduru POPRAVI-DOLJE, na način da proces započinje od prvog

neterminalnog čvora s indeksom pozicije p=└velicina[A]/2┘ prema korijenu.

Naime, u podnizu A[└velicina[A]/2┘ +1..duzina[A]-1] su svi elementi zapravo listovi stabla, pa ih možemo tretirati kao gomile koje sadrže samo po jedan element. Prema tome, proces stvaranja gomile možemo realizirati tako da proceduru POPRAVI-DOLJE pozivamo za podstabla koja imaju korijene na svim ostalim pozicijama: └velicina[A]/2┘, └velicina[A]/2┘ -1, ..., 0

Proces stvaranja gomile opisan je procedurom STVORI-GOMILU

**STVORI-GOMILU(A)**

**1 Velicina[A]**🡨**duzina[A]**

**2 for i**🡨**[velicina[A]/2]downto 0 do**

**3 POPRAVI-DOLJE(A, i)**

**4 end\_for**

Proces izgradnje gomile je ilustriran primjerom na slici. Proces

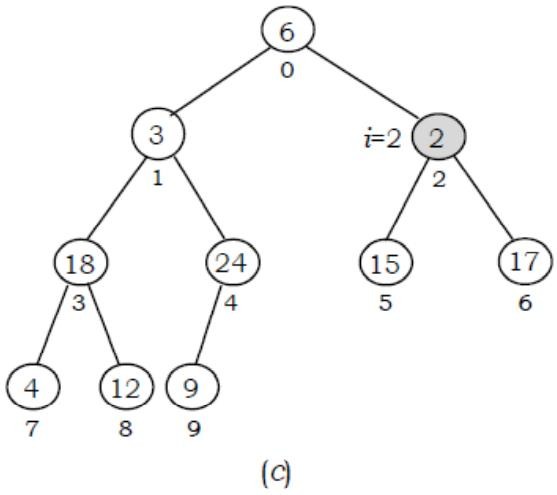
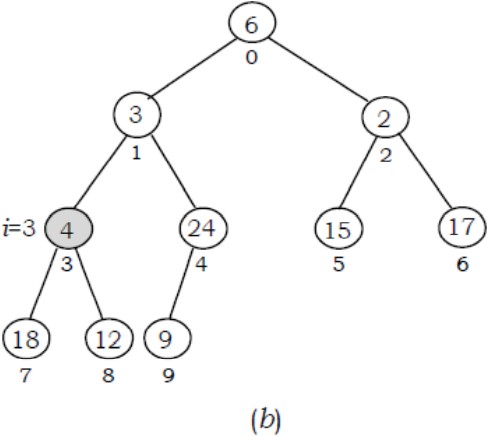
počinje razmatranjem zadnjeg čvora koji nije list.

Zadnji čvor koji nije list je čvor 24, koji se nalazi na poziciji s

indeksom i=4 (slika a).

Uspoređujemo taj čvor sa svojom djecom. U ovom slučaju, čvor 24 ima samo jedno dijete. Ako je čvor kojeg razmatramo veći od svoje djece, kao što je to slučaj sa čvorom 24, onda podstablo sa korijenom u tom čvoru ima svojstvo poretka, pa nema potrebe za promjenom. Vraćamo se nazad kroz stablo, razmatrajući čvor po čvor.

Sljedeći čvor koji razmatramo se nalazi na poziciji i=3, a to je čvor 4 (slika b).

Čvor 4 je manji od oba svoja djeteta, pa njegovu poziciju zamjenjujemo sa većim djetetom, a to je čvor 18.

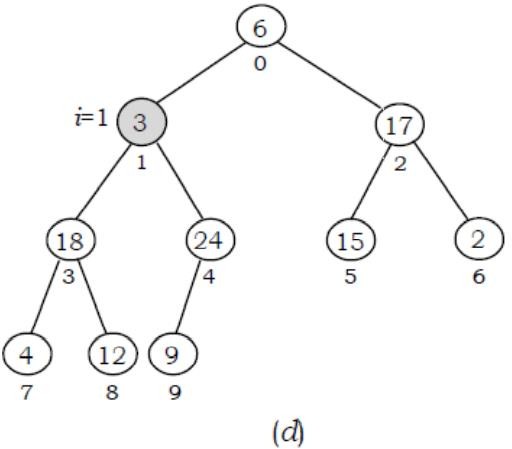
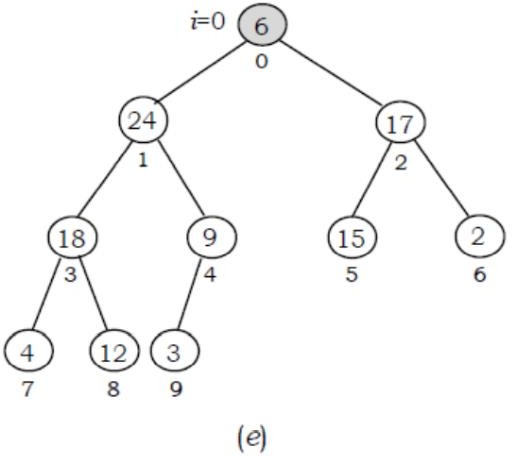
Rezultirajuće stablo je prikazano na slici c.

Sljedeći čvor koji razmatramo je čvor 2, koji se nalazi na poziciji i=2.

Taj čvor je manji od oba svoja djeteta, pa njegovu poziciju zamjenjujemo sa pozicijom čvora 17, jer je čvor 17 veći od čvora 15.

Rezultirajuće stablo je prikazano na slici d.

Zatim, razmatramo čvor 3 na poziciji i=1.

Taj čvor je manji od oba svoja djeteta, pa će pozivom procedure POPRAVI-DOLJE (A, 1) biti prvo zamjenjena pozicija čvorova 3 i 24, a zatim će također biti međusobno zamjenjene pozicije čvorova 3 i 9.

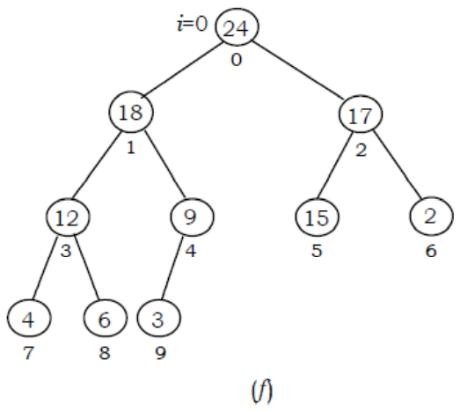
Rezultirajuće stablo je prikazano na slici e. Na kraju, pozivom procedure POPRAVI-DOLJE (A, 0)

razmatramo čvor 6 koji se nalazi na poziciji s indeksom 0.

U prvom koraku će se zamijeniti pozicije čvorova 6 i 24. Nakon toga će biti zamjenjene pozicije čvorova 6 i 18.

Pošto podstablo s korijenom na poziciji i=3 neće imati zahtijevano svojstvo poretka, u sljedećem koraku će biti zamjenjene pozicije čvorova 6 i 12.

Prema tome, spuštanje čvora 6 u stablu je došlo do lista.

Rezultirajuće stablo je prikazano na slici f.

Primijetimo da kada god dođe do poziva procedure POPRAVI- DOLJE (A, i), oba podstabla čvora na poziciji i imaju zahtijevano svojstvo poretka.

Odnosno, prije početka svake od iteracija for petlje, čvorovi i+1, i+2, ..., n, predstavljaju korijene podstabala koja imaju potrebno svojstvo poretka.

Procedura STVORI-GOMILU ima vremensku složenost O(n).

## Promjena vrijednosti i umetanje novog čvora

Pod promjenom vrijednosti zapravo podrazumijevamo povećanje vrijednosti nekog ključa u tipu gomile s maksimumom u korijenu, odnosno smanjivanje vrijednosti ključa u tipu gomile s minimumom u korijenu.

Pri takvim promjenama vrijednosti biti će nam potrebna procedura koja omogućuje popravak gomile nakon promjene vrijednosti nekog od ključeva A[i].

U nastavku ćemo podrazumijevati da se promjena vrijednosti obavlja u tipu gomile s maksimumom u korijenu. Nakon povećanja vrijednosti nekog ključa A[i], postupak popravljanja gomile se izvodi podizanjem čvora prema korijenu i opisan je procedurom POPRAVI-GORE

**POPRAVI-GORE(A, i)**

**1 while(i != 0 and A[i] > A[RODITELJ(i)])do**

**2 A[i]<->A[RODITELJ(i)]**

**3 i🡨RODITELJ(i)**

**4 end\_while**

Dakle, prije poziva procedure pretpostavljamo da niz A[0..velicina[A]-1] zadovoljava svojstvo

poretka, te da je došlo do povećanja ključa A[i] koje može narušiti to svojstvo.

Popravak se izvodi počevši od čvora u kojem je došlo do povećanja vrijednosti ključa, do korijena gomile da bi se pronašlo ispravno mjesto za novu povećanu vrijednost.

Kretanjem kroz gomilu prema korijenu uspoređuju se elementi sa svojim roditeljima, pri čemu dolazi do izmjene njihovih pozicija, ako je vrijednost ključa nekog elementa veća od vrijednosti ključa njegovog roditelja (linija 2).

POPRAVI-GORE O(logn)

Na slici je ilustriran način rada procedure POPRAVI-GORE, za primjer gomile prikazane nizom A

veličine 10.

Na slici a je prikazana početna konfiguracija gomile A, nakon povećanja vrijednosti ključa A[8] sa

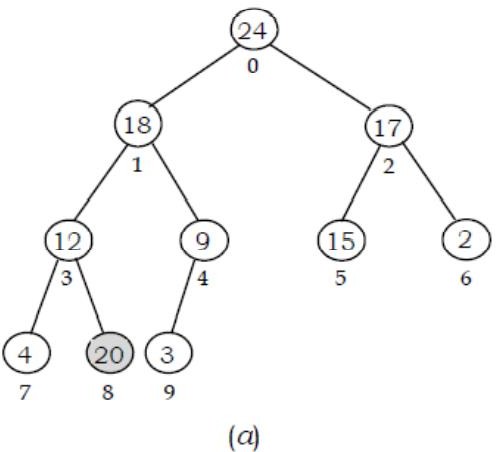
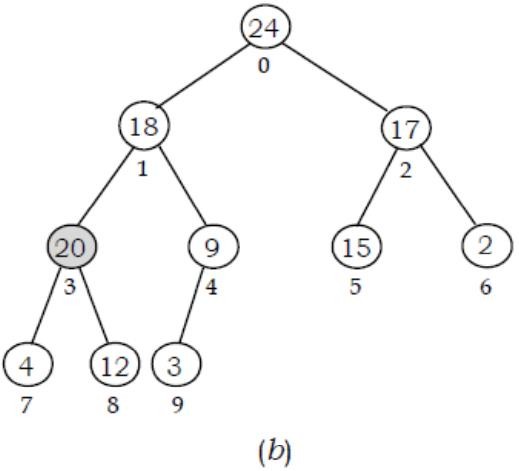
stare vrijednosti 6 na novu vrijednost A[8]=20.

To povećanje je prouzrokovalo narušavanje svojstva poretka, pa se gomila popravlja pozivom

procedure POPRAVI-GORE (A, 8).

U prvoj iteraciji (i=8) dolazi do izmjene pozicija elemenata A[3] i A[8], jer ključ na poziciji 8 ima veću

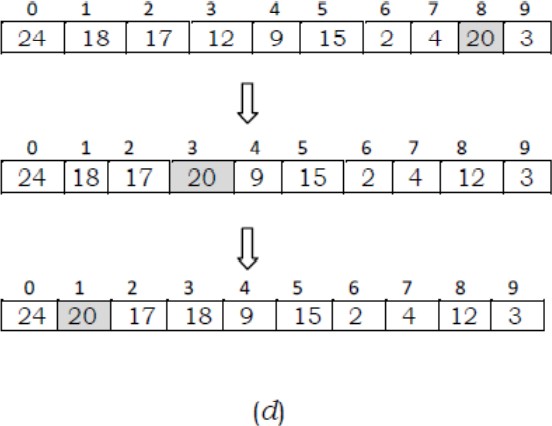
vrijednost od njegovog roditelja na poziciji 3 (slika a).

U sljedećoj iteraciji (i=3) dolazi do izmjene pozicija elemenata A[1] i A[3], jer ključ na poziciji 3 ima veću vrijednost od njegovog roditelja na poziciji 1 (slika b).

Budući da se u sljedećoj iteraciji (i=1) utvrđuje da roditelj na poziciji 0 ima veću vrijednost od čvora na poziciji 1, proces popravljanja se završava (slika c).

Na slici d je ilustriran sadržaj niza A, za konfiguracije gomila prikazanih na slikama 9.27a-c.



Operacija umetanja novog elementa u gomilu je implementirana procedurom UMETNI-U-GOMILU koja je prikazana u tabeli.

Procedura kao argumente uzima gomilu A i element x koji se umeće.

Ako je veličina gomile veća od kapaciteta, signalizira se njegovo prekoračenje (linije 1- 2).

Inače, u prvom koraku se na poziciju velicina[A] unosi novi ključ, a zatim se povećava veličina gomile

za 1 (linije 4-5).

Na kraju se poziva procedura POPRAVI-GORE, kojom se pozicionira čvor s novim ključem na odgovarajuću poziciju, da bi se održalo svojstvo poretka (linija 6).

**UMETNI-U-GOMILU(A, x)**

**1 if(velicina[A] >= kapacitet[a])then**

**2 ERROR(„Gomila je puna!“)**

**3 end\_if**

**4 A[velicina[A]]**🡨**x**

**5 velicina[A]**🡨**velicina[A]+1**

**6 POPRAVI-GORE(A, velicina[A]-1)**

Operacija promjene (povećanja) vrijednosti ključa je opisana procedurom PROMJENIKLJUC u tabeli.

Procedura kao argumente, pored gomile A, uzima poziciju ključa i, kojem se mijenja vrijednost, kao i novu vrijednost ključa x.

Nakon pridruživanja nove vrijednosti elementu A[i] (linija 4), jednostavno je potrebno samo još

pozvati proceduru POPRAVI-GORE, koja će obnoviti eventualno narušeno svojstvo poretka

Procedure UMETNI-U-GOMILU i PROMJENI-KLJUC imaju u najgorem slučaju vremensku složenost reda O(log n). Prema tome, ako je u gomilu potrebno umetnuti n elemenata, potrebno vrijeme za izvođenje te operacije je O(n log n).

**PROMJENI-KLJUC(A, i, x)**

**1 If(A[i]>=x)then**

**2 ERROR(„Nova vrijednost kljuca je manja od stare“)**

**3 end\_if**

**4 A[i]**🡨**x**

**5 POPRAVI-GORE(A, i)**

## Izbacivanje elemenata iz gomile

Operacija izbacivanja prvog (maksimalnog) elementa u gomili je opisana funkcijom IZBACI-PRVI u tabeli. Ako je veličina gomile jednaka nuli, funkcija prijavljuje grešku kojom se signalizira da je gomila prazna (linije 1-3).

Inače, prvo se smanjuje veličina gomile (linija 4), a zatim se izmjenjuju pozicije prvog elementa i

zadnjeg elementa na poziciji velicina[A] (linija 5).

Na taj način se prvi element pozicionira na mjesto koje je izvan raspona koji definira gomilu A[0..velicina[A]-1]. Navedena izmjena pozicija najčešće narušava svojstvo poretka, tako da se popravljanje gomile obavlja od korijena jednostavno pozivom procedure POPRAVI-DOLJE (A, 0).

Dakako, ako je gomila prethodno imala samo jedan element, tada se popravak ne treba niti obavljati, jer je gomila prazna (linije 6-8). Funkcija na kraju vraća izbačeni element (linija 9).

**IZBACI-PRVI(A)**

**1 if(velicina[A] = 0)then**

**2 ERROR(„Gomila je prazna“)**

**3 end\_if**

**4 velicina[A]**🡨**velicina[A]-1**

**5 A[0]<->A[velicina[A]]**

**6 if(velicina[a] != 0)then**

**7 POPRAVI-DOLJE(A, 0)**

**8 end\_if**

**9 return A[velicina[A]]**

Operacija izbacivanja elementa iz gomile na proizvoljnom mjestu s indeksom i je također relativno

jednostavna, uz implementirane procedure POPRAVI-GORE i POPRAVI-DOLJE.

Ta operacija je opisana funkcijom IZBACI u tabeli, koja kao argumente uzima pored gomile A i indeks i.

Slično kao i kod izbacivanja prvog elementa, najprije se smanjuje veličina gomile, a zatim se na

poziciju velicina[A], koja je izvan raspona gomile, upisuje element A[i], koji se izbacuje (linije 4-5).

Za obnavljanje svojstva poretka, potrebno je napraviti kako poziv procedure POPRAVI-GORE (A,i), tako i poziv procedure POPRAVIDOLJE (A, i) (linije 6-7). Funkcija IZBACI također vraća element koji se izbacuje iz gomile (linija 8).

**IZBACI(A, i)**

**1 if(i < 0 and i >= velicina[A])then**

**2 return „Indeks izvan dozvoljenog raspona“**

**3 end\_if**

**4 velicina[A]**🡨**velicina[A]-1**

**5 A[i]<->A[velicina[A]]**

**6 POPRAVI-GORE(A, i)**

**7 POPRAVI-DOLJE(A, i)**

**8 return A[velicina[A]]**

## Definicije i terminologija

**(Predavanje 12)**

**Heširanje**

Metoda koja omogućuje implementaciju algoritama kod kojih je vrijeme potrebno za pretraživanje

manje-više neovisno o dužini ulaznog niza

Možemo reći da smo se već upoznali s jednim primjerom metode heširanja koji omogućuje efikasan način pohranjivanja i pretraživanja ključeva, jer nizovi u svojoj suštini predstavljaju jedan jednostavan primjer takve organizacije podataka

Naime, pozicija s indeksom i, na koju se smješta neki element u nizu A[0..n-1], se izračunava počevši

od memorijske adrese na kojoj je pohranjen prvi element niza A[0].

Novi element se smješta na memorijsku adresu koja se dobije tako da se baznoj adresi niza doda umnožak veličine svakog elementa niza i indeksa i.

To znači da je potrebno vrijeme da se locira neki element u nizu na poziciji i konstantno, bez obzira

na njegovu poziciju, kao i na veličinu niza.

Dakle, pri smještanju i pronalaženju elementa A[i], potrebno je na temelju ulaznog indeksa pozicije i izračunati memorijsku lokaciju, na kojoj će biti smješten ili na kojoj će se tražiti taj element niza.

Pristup koji se koristi kod niza može se poopćiti na način da se koristi niz (tabela) koji ima onoliko

lokacija za pohranjivanje koliko ima različitih mogućih vrijednosti ključeva, te da indekse pozicija u

nizu koristimo kao vrijednosti ključeva

Na taj način je osigurano da za svaki ključ možemo koristiti jedinstvenu poziciju, kojoj se može

direktno pristupiti.

Međutim, ovakav pristup ima određene nedostatke. Pretpostavimo, na primjer, da je broj mogućih ključeva mnogo veći od broja ključeva koji će se u stvarnosti koristiti.

Tada se koristi niz (tabela) sa mnogo većim dimenzijama nego što je to potrebno, pa je iskorištenost

tabele vrlo mala.

Na primjer, pretpostavimo da je potrebno imati evidenciju za nekoliko stotina telefonskih

pretplatnika, te da je za ključ izabran telefonski broj. Nadalje, neka svaki telefonski broj ima format 0D7D6-D5D4D3-D2D1D0, gdje Di ∈ (i=0, 1, 2, ..., 7) označava znamenke pomoću kojih je dotični broj prikazan, pri čemu vrijedi Di ∈ {0, 1, 2, ...., 9}, i=0, 1, 2 ,..., 7.

Zatim, pretpostavimo da je svakom mogućem telefonskom broju pridružen indeks D7 × 107 + D6 × 106 + ⋯ + D0 × 100

Formalno, proces pridruživanja cjelobrojnog indeksa i nekom telefonskom broju možemo opisati

funkcijom (preslikavanjem) f:

𝑓: 𝑇𝐵 → {0, 1, 2, … , 99999999} TB je skup svih mogućih brojeva

To znači, na primjer, da će sukladno pravilu preslikavanja, koje je definirano funkcijom f, telefonskom broju 033-256-347 biti pridružen indeks 33256347, dok će telefonskom broju 061-780- 699 biti pridružen indeks 61780699.

Vidimo da indeks ima 8 decimalnih mjesta, što znači da je potreban niz veličine 108 .

Imamo li na umu da će evidencija sadržavati samo nekoliko stotina pretplatnika, možemo zaključiti

da je prethodno opisani pristup neprikladan sa stanovišta efikasnog korištenja memorijskih resursa

Međutim, iz prethodnog primjera možemo izvesti zaključak da je potrebno veličinu niza ograničiti, tako da ta veličina bude dovoljna za maksimalni broj ključeva koji će se u stvarnoj primjeni koristiti,

ali da se ipak na određeni način omogući što je moguće direktniji pristup pohranjenim ključevima u

tabeli.

Drugim riječima, biti će potrebno preoblikovati funkciju f, koja će trebati preslikavati ključeve u

reducirani raspon indeksa u nizu.

Takva funkcija se naziva heš funkcija, dok se tabela kojoj se pristupa na prethodno opisani način naziva heš tabela.

Idealno bi bilo kada bi heš funkcija za svaki ključ davala jedinstvenu poziciju u heš tabeli. Opisani

pristup podacima se može koristiti i za pristup podacima u unutrašnjoj memoriji (unutrašnje heširanje), kao i za pristup podacima na vanjskim memorijama (vanjsko heširanje).

Neka T[i], 0 ≤ i ≤ n-1, označava tabelu sa n lokacija.

Zatim, neka U označava univerzalni skup svih mogućih ključeva, te neka ki označava ključeve koji

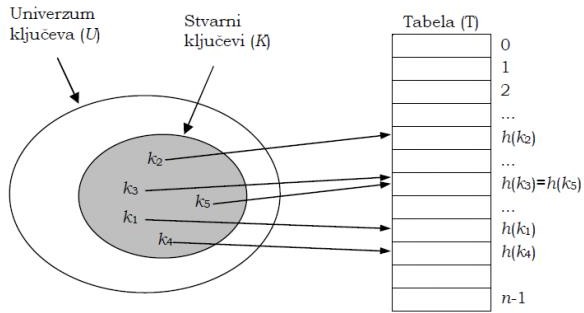
pripadaju skupu U, ki ∈U.

Isto tako, označimo skup stvarno korištenih ključeva sa K, pri čemu možemo pretpostaviti da je |K|

<< |U|.

Heš funkcija h je preslikavanje iz skupa ključeva U u skup mogućih pozicija u tabeli: ℎ:𝑈 → {0, 1, 2, …

, 𝑛 − 1}



Važno je istaknuti da smo uvođenjem heš funkcije bitno smanjili raspon indeksa u nizu koji će se

koristiti.

Taj raspon je smanjen sa |U| na ukupno n indeksa.

Međutim, ovo zapravo znači da je moguća situacija u kojoj će heš funkcija preslikati dva različita ključa ki i kj u istu vrijednost

h(ki) = h(kj), ki ≠ kj

Dakako, dva elementa se ne mogu pohraniti na istu poziciju u tabeli, pa ovakav slučaj predstavlja situaciju koja se naziva kolizija. Na slici kolizija je ilustrirana preslikavanjem ključeva k3 i k5, h(k3)=h(k5).

Međutim, postoje tehnike koje omogućuju efikasno rješavanje problema kolizije. Pretpostavimo da je u ovom primjeru u tabelu na matičnu poziciju prvo umetnut element s ključem k3, pa se kolizija pojavljuje pri umetanju elementa s ključem k5.

U ovoj situaciji se elementu s ključem k5 mora pronaći neka druga lokacija. Ta druga lokacija se

treba odrediti na takav način da, pri kasnijem pretraživanju elementa s ključem k5, postoji postupak kojim se može pronaći lokacija tog elementa, pošto se ne nalazi na matičnoj poziciji.

Naravno, idealna situacija bi bila da se izbjegne pojavljivanje slučajeva kolizije izborom prikladne funkcije heširanja h. To se može pokušati postići izborom funkcije h, tako da njenom primjenom za slučajni ključ ki pripada U postoji jednaka vjerojatnost da se on preslika u bilo koju matičnu adresu.

Tako izabrane heš funkcije minimiziraju vjerojatnost pojavljivanja kolizija.

Međutim, pošto je |U|>n, sigurno postoje najmanje dva ključa koji imaju istu heš vrijednost. Zbog

toga je potrebno predvidjeti način razrješavanja problema kolizije.

Iz prethodnog uvodnog opisa možemo zaključiti da je za primjenu principa heširanja od ključne važnosti sljedeće:

dizajniranje prikladne heš funkcije;

izbor prikladne tehnike za rješavanje problema kolizije

## Dizajniranje heš funkcije

Pri dizajniranju heš funkcija treba zadovoljiti sljedeće zahtjeve:

heš funkcija treba biti što jednostavnija, radi bržeg izračunavanja;

heš funkcija treba minimizirati broj kolizija

Prvi zahtjev zapravo podrazumijeva korištenje što manje različitih operacija. Jednostavnost te funkcije i brzina njenog izračunavanja su važni, jer se izračunavanje obavlja pri svakom pristupu tabeli.

Drugi zahtjev se postiže prikladnim izborom funkcije h, koja raspoređuje elemente što ravnomjernije u heš tabeli.

Isto tako, bitno je istaknuti da su ova dva zahtjeva ponekada međusobno suprotstavljena, pa je u

takvim situacijama potrebno napraviti kompromis

U većini primjena heš funkcije pretpostavljaju da je univerzalni skup ključeva skup prirodnih brojeva N={0, 1, 2, ...}.

U slučajevima kada ključevi nisu prirodni brojevi, potrebno je pronaći način da te ključeve

interpretiramo kao prirodne brojeve.

Na primjer, pojedine znakove od kojih je sastavljen ključ možemo predstaviti ASCII kodovima, te primijeniti npr. takozvanu radix notaciju.

Imajući u vidu da je primjena tog postupka zapravo vrlo jednostavna, u nastavku će se bez smanjenja općenitosti izlaganja pretpostavljati da su ključevi prirodni brojevi

## Metoda izdvajanja

Kod ove metode se iz ključa izdvaja određeni broj znamenki.

Na primjer, ako pohranjujemo telefonske brojeve prikazane pomoću devet znamenki u heš tabelu sa 1000 lokacija, za preslikavanje nekog broja u heš vrijednost mogli bismo izdvojiti četvrtu, petu i osmu znamenku iz tog telefonskog broja.

U tom slučaju bi npr. telefonski broj 033256347 imao matičnu poziciju s indeksom 254. Ovakav metod je vrlo jednostavan i brz, jer postupak preslikavanja uključuje jednostavno izdvajanje odgovarajućih znamenki iz ulaznog ključa.

Međutim, broj kolizija koje ovakav metod može generirati ovisi o uniformnosti ulaznih podataka.

Ako se pohranjuju, na primjer, telefonski brojevi pretplatnika koji svi žive u jednom području (dijelu grada), te ako pretpostavimo da je prva znamenka (ne uključujući pozivni dio broja koji pripada pojedinim regionima, npr. 033) ista za sve pretplatnike iz tog područja, onda će svi telefonski brojevi, iz područja kojem pripada i tel. broj 033256347, imati heš vrijednost koja počinje brojem 2.

To znači da će se koristiti samo pozicije s indeksom 200 do 299, pa će 900 lokacija ostati zapravo neiskorišteno.

Ovaj problem bi se mogao izbjeći tako da se pri dizajniranju heš funkcije izdvoje tri zadnje znamenke. Prema tome, pri korištenju metoda izdvajanja, potrebno je voditi računa o pravilnostima koje se mogu uočiti kod ulaznih ključeva, da bi se oblikovalo preslikavanje koje će onemogućiti da te

pravilnosti proizvedu takav učinak da se većina ulaznih ključeva preslikava u relativno uzak raspon

indeksa.

## Metoda dijeljenja

Metoda dijeljenja je jedna od najčešće korištenih tehnika za dizajniranje heš funkcija. Kod ove metode se ključ k preslikava u jednu od n pozicija uzimanjem ostatka cjelobrojnog dijeljenja k i n

h(k) = k mod n.

Na primjer, ako heš tabela ima veličinu n=1000, a vrijednost ključa je k=3481, onda će element sa ključem k biti pohranjen na poziciju s indeksom 481

Da bi se smanjila vjerojatnost pojavljivanja kolizija, potrebno je izbjegavati određene vrijednosti za n.

Na primjer, ako je n stepen broja 2 (n=2i), onda se u binarnom prikazu vrijednost h(k) dobije

izdvajanjem donjih i bitova ključa.

Dakle, h(k) ne ovisi o svim bitovima ključa, što svakako nije preporučljivo. Bolje je dizajnirati heš

funkciju tako da ona ovisi o svim bitovima ključa.

Nadalje, ako je unaprijed poznato da su ključevi većinom parni brojevi ili većinom neparni brojevi, onda se ne preporučuje da je n paran broj.

Naime, ako je n paran, tada se parni ključevi preslikavaju isključivo u parne pozicije, dok se neparni

ključevi preslikavaju isključivo u neparne pozicije.

To znači da će biti opterećena ili polovina lokacija sa parnim indeksima ili polovina lokacija sa neparnim indeksima. Pokazuje se da je dobro izabrati da broj n bude prost broj, koji nije previše blizu nekom stepenu broja 2

## Metoda množenja

Metoda množenja uključuje dva koraka. U prvom koraku množimo vrijednost ključa k sa konstantom

c, koja je u rasponu 0 < c < 1.

Nakon toga se iz proizvoda uzima dio iza decimalne tačke, koji se množi sa veličinom tabele n, da bi

se dobila pozicija u tabeli. Dakle, heš funkcija h je definirana na sljedeći način



gdje „ck mod 1“ označava dio iza decimalne tačke od umnoška ck.

Odnosno

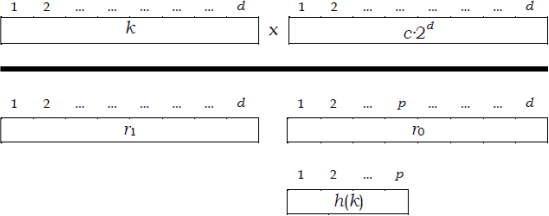
Jedna od prednosti metode množenja je u tome što izbor broja n nije kritičan.

Radi što brže i jednostavnije implementacije funkcije h može se npr. odabrati da n bude stepen broja

2 (𝑛 = 2 𝑝 ) za neku cjelobrojnu vrijednost p).

Označimo dužinu mašinske riječi kao d i pretpostavimo da se ključ može prikazati pomoću jedne mašinske riječi. Zatim, ograničimo da se konstanta c može prikazati kao s/2 𝑑 , gdje je s cjelobrojna vrijednost u intervalu 0 < s < 2d.

Implementacija izračunavanja funkcije h, uz prethodno navedena ograničenja, je ilustrirana na slici



Ključ k se množi sa └c2 𝑑┘, što će rezultirati vrijednošću koja se može prikazati sa 2d bitova:

𝑟112 𝑑 + 𝑟0

gdje r1 predstavlja gornju riječ, dok r0 predstavlja donju riječ rezultata množenja.

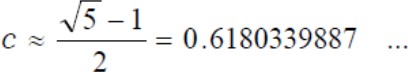
Rezultat heš funkcije h(k) se dobije jednostavnim izdvajanjem p najviših bitova donje riječi proizvoda

r0

Iako ova metoda radi sa bilo kojom vrijednošću konstante c, ipak određena teoretska razmatranja

pokazuju da neke vrijednosti konstante c mogu dati bolje rezultate u odnosu na druge vrijednosti.

Na primjer, Knuth sugerira korištenje konstante c koja ima sljedeću vrijednost



## Metoda sklapanja

Ova metoda se temelji na razbijanju ključa na nekoliko dijelova, koji se zatim na određeni način

sklapaju.

Na primjer, pretpostavimo da koristimo heš tabelu veličine n=1000 za pohranjivanje podataka o

telefonskim pretplatnicima, te da je telefonski broj, kao ključ elemenata u tabeli, prikazan pomoću 9

znamenki.

Tada se ključ može npr. razbiti u 3 grupe, pri čemu svaku grupu čine 3 znamenke

Nakon toga se grupe brojeva zbroje da bi se dobio indeks pozicije u rasponu 0 do 999.

Na primjer, telefonski broj 033256347 možemo podijeliti na sljedeće tri grupe: 033, 256 i 347.

Zbrajanjem se dobija heš vrijednost 636, koju možemo koristiti kao indeks pozicije za pristup dotičnom elementu tabele.

Ako uzmemo primjer telefonskog broja 061877345, tada bi heš vrijednost bila 61+877+345=1283.

Pošto heš vrijednost prelazi preko veličine tabele, možemo izdvojitisamo zadnje 3 znamenke, što bi u ovom primjeru značilo da je heš vrijednost 283

## Rješavanje kolizija otvorenim adresiranjem

Prethodno je istaknuto da postoje slučajevi u kojima heš funkcija preslikava dva različita ključa u istu vrijednost, te da takav slučaj nazivamo kolizija. Najčešće korišteni pristupi razrješavanju kolizija su:

otvoreno adresiranje;

ulančavanje.

Kod otvorenog adresiranja slučajevi kolizije se razrješavaju tako što se, nakon utvrđivanja da je matična adresa ključa zauzeta, pronalazi neka druga slobodna lokacija u tabeli, koristeći prethodno definirana pravila.

Uvedimo oznaku za početno ispitivanje i=0, te oznake i=1, 2, ..., n-1, za svako sljedeće i-to ponovno ispitivanje.

Da bi se, na primjer, obavilo umetanje novog ključa u tabelu, heš funkcija h kao argument ne koristi samo ključ k, nego i oznaku ispitivanja i, pri čemu i=0 odgovara početnom ispitivanju, i=1 odgovara prvom ponovnom ispitivanju, itd.

Formalno, to znači da je kod otvorenog adresiranja heš funkcija h preslikavanje:

ℎ:𝑈 × {0, 1, 2, … , 𝑛 − 1} → {0, 1, 2, … , 𝑛 − 1}

gdje je:

U – univerzum ključeva

N – veličina tabele

To znači da heš funkcija h(k, i), k ∈ U, i ∈ {0, 1, 2, ..., n-1}, daje heš vrijednost na kojoj će se ispitivati

slobodno mjesto za ključ k u i-tom ponavljajućem ispitivanju (nulti pokušaj je početno ispitivanje).

Na taj način, heš funkcijom h je za svaki ključ k definiran tzv. ispitni niz



koji zapravo definira pravilo koje pokazuje na kojoj lokaciji h(k, i) će se tražiti slobodno mjesto u i- tom ponavljajućem pokušaju, 0 ≤ i ≤ n-1.

Ispitni niz bi trebao biti takav da se u njemu ne pojavi niti jedna prethodna pozicija, prije nego što se ispitaju sve pozicije. Drugim riječima, ispitni niz bi trebao biti neka permutacija (0, 1, 2, ..., n-1), što osigurava da se kolizija u najgorem slučaju razriješi nakon n koraka ispitivanja

Na taj način, svaka lokacija u tabeli može

eventualno biti razmotrena kao mjesto gdje će se

pohraniti novi element

Na slici je prikazan algoritam umetanja novog elementa u heš tabelu pomoću otvorenog adresiranja, u obliku funkcije HASH-UMETNI.

Pretpostavljeno je da su elementi u tabeli T ključevi bez pratećih informacija. Svaka lokacija u tabeli sadrži ili ključ ili NIL (ako je lokacija prazna).

Funkcija HASH-UMETNI vraća adresu (indeks pozicije) u tabeli na kojoj je pohranjen ključ ili

prijavljuje grešku porukom „Tabela je puna“, ako su ispitane sve lokacije u tabeli, a nije pronađena slobodna lokacija za pohranjivanje novog elementa.

**HASH-UMETNI (T, k)**

**1 i<--0**

**2 repeat**

**3 j<-- h(k, i)**

**4 if (T[j] = NIL) then**

**5 T[j]<--k**

**6 return j**

**7 else**

**8 i<-- i+1**

**9 end\_if**

**10 until i = n**

**11 ERROR(„Tabela je puna!")**

Algoritam pretraživanja tabele T za zadati ključ k je prikazan

na slici, u obliku funkcije HASH-TRAZI.

Funkcija kao ulaz uzima heš tabelu T i ključ k, a vraća

poziciju j, ako je na toj poziciji pohranjen traženi ključ k, ili vraća NIL, ako ključ k nije prisutan u tabeli.

Funkcija HASH-TRAZI koristi isti ispitni niz, koji koristi i funkcija za umetanje novog elementa HASH-UMETNI.

Prema tome, algoritam će završiti rad neuspješno kada na nekoj od pozicija u ispitnom nizu pronađe praznu lokaciju (T[j] = NIL), ili kada je iscrpljen čitav ispitni niz (i = n).

**HASH-TRAZI (T, k)**

**1 i <--0**

**2 repeat**

**3 j<-- h(k, i)**

**4 if (T[j] = k) then**

**5 return j**

**6 else**

**7 i<--i+1**

**8 end\_if**

**9 until (T[j] = NIL or i = n)**

**10return NIL**

Heširanje sa otvorenim adresiranjem karakteriziraju problemi koji se pojavljuju pri brisanju

elemenata iz tabele.

Ako bi trebali obrisati ključ na nekoj poziciji, to ne možemo uraditi tako što ćemo u tu lokaciju upisati NIL i na taj način je označiti praznom.

Pretpostavimo, na primjer, da se na nekoj poziciji (h(k1, 0)) nalazi ključ k1. Zatim, pretpostavimo da se u tabelu umeće ključ k2, koji ima istu matičnu adresu, h(k2, 0)=h(k1, 0). To znači da će se koristiti ponovno ispitivanje (h(k2, 1), h(k2, 2), ...), te će se ključ k2 pohraniti na nekoj drugoj lokaciji (h(k2, i), i ≠ 0).

Zatim, pretpostavimo da se iz tabele izbriše ključ k1, tako što se na njegovoj poziciji upiše vrijednost NIL. Ako sada pretpostavimo da se u tabeli treba pronaći ključ k2, procedura HASH-TRAZI će, nakon što utvrdi da je na poziciji početnog ispitivanja h(k2, 0)=h(k1, 0) upisana vrijednost NIL, završiti rad

tako što će vratiti vrijednost NIL kao znak da ključ k2 nije pronađen, iako je on prisutan u tabeli.

Jedno od rješenja za prethodno opisani problem je u tome da se pri brisanju, umjesto upisivanja oznake NIL, koristi upisivanje neke druge oznake koja je različita od vrijednosti svih ključeva, kao i od vrijednosti NIL.

Uzmimo, na primjer, da je to oznaka DEL. Na ovaj način smo postigli da se ispitni niz neće prekidati, ako se u ispitivanjima naiđe na vrijednost DEL.

Pri pretraživanju i nailasku na vrijednost DEL, to će samo biti znak da je element na toj lokaciji obrisan, ali će se ispitni niz nastaviti.

S druge strane, prilikom umetanja lokacije u kojima je upisana oznaka DEL se smatraju slobodnima, pa se te lokacije mogu koristiti za pohranjivanje novih elemenata.

U ovom slučaju bi bilo potrebno malo modificirati funkciju HASH-UMETNI, da bi se omogućilo umetanje elemenata u lokacije u kojima je upisana specijalna vrijednost DEL. Funkciju HASH-TRAZI nije potrebno modificirati, jer će algoritam nastaviti pretraživanje i nakon što u ispitnom nizu naiđe na lokaciju u kojoj je upisana oznaka DEL.

Na kraju, možemo zaključiti da ova tehnika ima nedostatak koji se ogleda u lošijim performansama pretraživanja, pa je ova tehnika zapravo prikladna samo u situacijama kada se ne očekuju česta brisanja elemenata.

U prethodnom opisu heširanja sa otvorenim adresiranjem nisu opisani postupci dizajniranja ispitnog niza. Zato su u nastavku opisani najčešće korišteni pristupi: linearno ispitivanje, kvadratno ispitivanje i dvostruko heširanje

## Linerano ispitivanje

Linearno ispitivanje je najjednostavnija metoda dizajniranja ispitnog niza.

Kod ove metode, u slučaju kada se na nekoj poziciji pojavi kolizija, ispituje se sljedeća lokacija.

Kada se u procesu ispitivanja dođe do kraja tabele, tada se ispitivanje nastavlja od njenog početka.

Uvedimo pomoćnu funkciju

ℎ:𝑈 → {0, 1, 2, … , 𝑛 − 1}

koja opisuje početno ispitivanje.

Tada, metodu linearnog ispitivanja možemo formalno prikazati funkcijom h:

ℎ (𝑘, 𝑖) = (ℎ0 (𝑘) + 𝑖) 𝑚𝑜𝑑 𝑛, 𝑖 = 0, 1, 2, … , 𝑛 − 1

Dakle, za dati ključ k, prvo se ispituje matična adresa ℎ0 (𝑘) .

Nakon toga se ispituju pozicije sljedećim redoslijedom: ℎ0 (𝑘) +1, ℎ0 (𝑘) +2, ..., n-1, do kraja tabele.

Zatim se prelazi na početak tabele, pa se ispituju pozicije 0, 1, 2, ..., ℎ0 (𝑘) -1.

### Primjer

Pretpostavimo da heš tabela ima 10 lokacija (n=10).

Pretpostavimo da se koristi heš funkcija početnog ispitivanja h0(k)=(k mod 10). Prema tome, heš vrijednost je zapravo definirana zadnjom znamenkom, jer je ta znamenka jednaka ostatku dijeljenja ključa sa 10.

Prema tome, linearno ispitivanje je za ovaj primjer definirano heš funkcijom h(k, i):

ℎ (𝑘, 𝑖) = (𝑘 𝑚𝑜𝑑 10 + 𝑖) 𝑚𝑜𝑑 10 , 𝑖 = 0, 1, 2, … , 9

Za ilustraciju nekih tipičnih situacija koje se pojavljuju kod heširanja pomoću otvorenog adresiranja, za primjer ćemo uzeti da se izvršavaju operacije sljedećim redoslijedom:

prvih osam operacija predstavljaju umetanje ključeva: 31, 54, 25, 7, 14, 19, 70, 89;

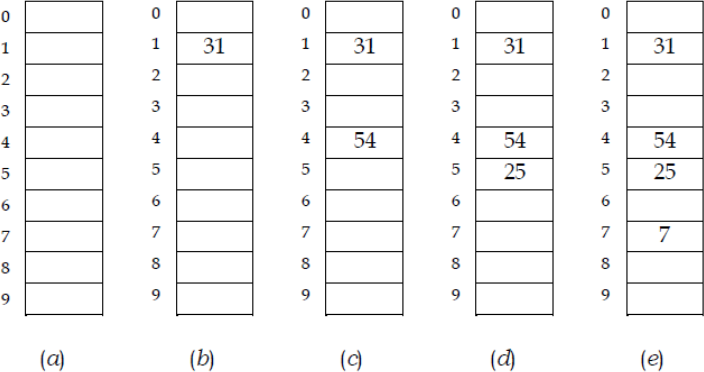
sljedeća operacija je brisanje ključa 54;

sljedeće četiri operacije su pronalaženje ključeva: 33, 31, 14 i 61;

zadnje dvije operacije predstavljaju umetanje ključeva 64 i 29

Počevši od prazne tabele (slika a), prva 4 ključa (31, 54, 25, 7) se umeću bez kolizija na svoje matične

adrese (slike b-e).



Međutim, pri umetanju petog ključa 14, utvrđuje se da je njegova matična adresa 4 zauzeta, pa se zato prelazi na pronalaženje prve slobodne lokacije.

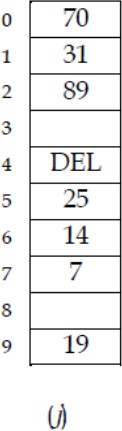
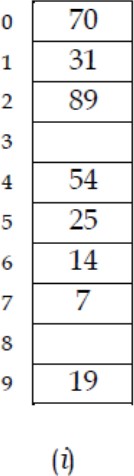
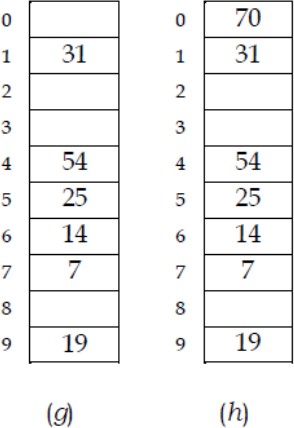
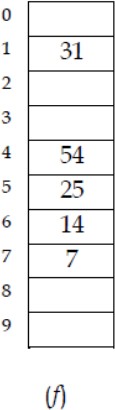
Kao prva slobodna lokacija, pronalazi se lokacija sa adresom 6, pa se ključ 14 umeće na tu lokaciju

(slika f).

Dakle, pri umetanju ključa 14 je generiran ispitni niz: 4, 5, i 6

Nakon toga slijede još dvije operacije umetanja ključeva 19 i 70, koje se također obavljaju bez

kolizija (slike g i h).



U sljedećoj operaciji se umeće ključ 89. Njegova matična adresa je 9. Međutim, lokacija sa tom

adresom je zauzeta, a istovremeno je to i zadnja lokacija u tabeli, pa se zbog toga pronalaženje prve prazne lokacije nastavlja prvim ponovnim ispitivanjem, što je definirano funkcijom h(89,1)=((89mod 10)+1)=0.

Dakle, pronalaženje slobodne lokacije se nastavlja od početka tabele, što je i očekivano, jer je

prethodno ispitana zadnja lokacija u tabeli.

Prva slobodna lokacija ima adresu 2, na koju se pohranjuje ključ 89, u četvrtom pokušaju (slika i). Prema tome, pri umetanju ključa 89, generiran je ispitni niz 9, 0, 1 i 2.

Sljedeća operacija je brisanje ključa 54.

Ovaj ključ se pronalazi već pri početnom ispitivanju matične adrese, koju dobijemo iz izraza h(54,

0)=((54 mod 10) +0) mod 10=4.

Na mjestu obrisanog ključa 54 se upisuje predefinirana oznaka DEL (slika j).

Sljedeća operacija je pronalaženje ključa 33.

Matična adresa ovog ključa je 3, te se nakon ispitivanja na toj adresi utvrđuje da je riječ o praznoj

lokaciji, pa zaključujemo da ključ 33 nije prisutan u tabeli (slika j)

Naredna operacija je pronalaženje ključa 31.

Matična adresa ovog ključa je 1, pa se već u početnom pokušaju pronalazi traženi ključ (slika j).

Nakon toga slijedi operacija pronalaženja ključa 14.

Matična adresa je 4, pa se nakon ispitivanja te lokacije, sukladno pravilu linearnog ispitivanja,

nastavlja sa lokacijama 5 i 6.

Na lokaciji 6 je pronađen traženi ključ (slika j).

Primijetite da nismo pri brisanju čvora 54 upisali specijalnu oznaku DEL, tada bi pronalaženje čvora 14 rezultiralo neuspjehom, zato što bi prema proceduri pretraživanja HASH-TRAZI proces

pronalaženja bio prekinut, jer bismo u nizu ispitivanja naišli na praznu lokaciju

Sljedeća operacija je operacija pronalaženja ključa 61.

Pošto je matična adresa 1, proces pronalaženja počinje od te lokacije.

Iz slike j zaključujemo da će biti generiran niz ispitivanja: 1, 2 i 3 - do prve slobodne lokacije.

Nailaskom na prvu slobodnu lokaciju, pronalaženje traženog ključa je neuspješno, te se ispravno zaključuje da ključ 61 nije prisutan u tabeli.

I na kraju, imamo još dvije operacije umetanja.

Prvo se umeće ključ 64. Na matičnoj adresi 4 je pohranjena oznaka DEL, što znači da se na tu lokaciju može pohraniti ključ (slika k).

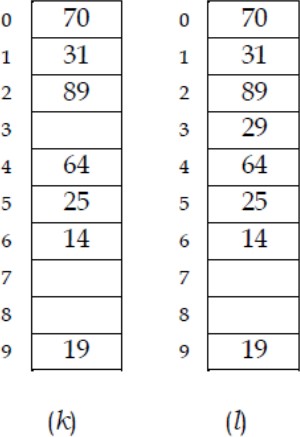
U zadnjoj ilustriranoj operaciji se umeće ključ 29.

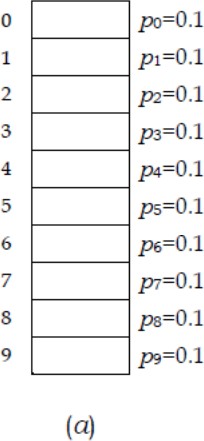
Proces pronalaženja prve slobodne lokacije počinje od lokacije s matičnom adresom 9, te se

nastavlja ispitivanjem lokacija: 0, 1, 2 i 3 - do prve slobodne lokacije.

Dakle, pri umetanju ključa 29 je generiran niz ispitivanja: 9, 0, 1, 2 i 3 - pa je tek u petom pristupu

tabeli pohranjen taj ključ



Glavni nedostatak linearnog ispitivanja je pojava da se sa rastom popunjenosti tabele pojavljuju tendencije grupiranja zauzetih lokacija, što za posljedicu ima povećanje ispitnog niza, odnosno smanjenje ukupnih

performansi heširanja.

Ova pojava se naziva primarno grupiranje.

Da bi razumjeli mehanizam nastajanja primarnog grupiranja, pretpostavimo

da heš funkcija uniformno distribuira elemente u heš tabeli, te pretpostavimo da heš tabela ima 10 lokacija (n=10).

Kada je tabela prazna, sve lokacije imaju istu vjerojatnost popunjavanja (pi =

0.1, 0 ≤ i ≤ 9), zbog pretpostavke o uniformnosti heš funkcije (slika a).

Kada u praznu heš tabelu umećemo prvi element s ključem k1, taj element se pohranjuje tačno na onu adresu koja je dobivena heš funkcijom.

Nadalje, pretpostavimo da je prvi element pohranjen na lokaciju s adresom 3 (slika b).

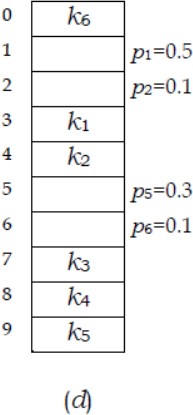
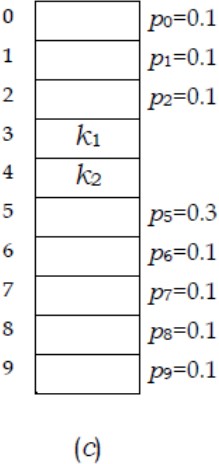
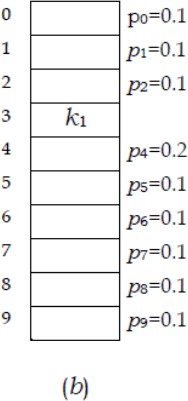
Razmotrimo sada situaciju pri umetanju drugog elementa s ključem k2, tako što ćemo se fokusirati

na lokacije s adresama 3 i 4 .

Zbog pretpostavke o uniformnosti heš funkcije, možemo zaključiti da adrese 3 i 4 imaju istu vjerojatnost da element s ključem k2 bude heš funkcijom preslikan u neku od te dvije adrese.

Međutim, lokacija s adresom 3 je već popunjena prvim elementom, što znači da ako heš funkcija preslikava ključ k2 u bilo koju od spomenute dvije adrese (3 i 4), tada će drugi element biti pohranjen na adresi 4.

Prema tome, lokacija s adresom 4, pri umetanju drugog elementa, ima dvostruko veću vjerojatnost (p4=0.2) da će biti popunjena u odnosu na vjerojatnost (pi=0.1, i ∋ {3, 4}) za sve ostale prazne lokacije (slika b).



Pretpostavimo sada da je umetanjem drugog elementa s ključem k2 popunjena upravo lokacija s

adresom 4 (slika c).

U ovoj situaciji će pri umetanju trećeg elementa s ključem k3, lokacija s adresom 5 biti popunjena u slučaju da heš funkcija preslikava ključ k3 u jednu od sljedeće tri adrese: 3, 4 i 5.

Drugim riječima, to znači da pri umetanju trećeg elementa, lokacija s adresom 5 ima tri puta veću vjerojatnost (p5=0.3) da će biti popunjena u odnosu na vjerojatnost (pi = 0.1, i ne pripada {3, 4, 5}), odnosno za sve ostale prazne lokacije (slika c).

Ako sa gi označimo broj zauzetih lokacija koje čine jednu grupu lokacija u neprekinutom nizu, koja se nalazi neposredno ispred neke lokacije s adresom i, onda možemo izračunati vjerojatnost popunjavanja pi lokacije i kao:

pi = (gi+1)/n,

gdje n označava ukupan broj mjesta u tabeli.

Na slici d su prikazane vjerojatnosti, nakon što su u tabelu umetnuta još dodatna 3 elementa s ključevima: k4, k5 i k6.

Za prazne lokacije s adresama 1, 2, 5 i 6, vjerojatnoće popunjavanja su 0.5, 0.1, 0.3 i 0.1, respektivno

Primarno grupiranje se u općem slučaju pojavljuje kada za dva različita ključa ki i kj vrijedi da od

nekog p-tog ponovnog ispitivanja za ključ ki dolazi do podudaranja sa nizom ispitivanja za ključ kj.

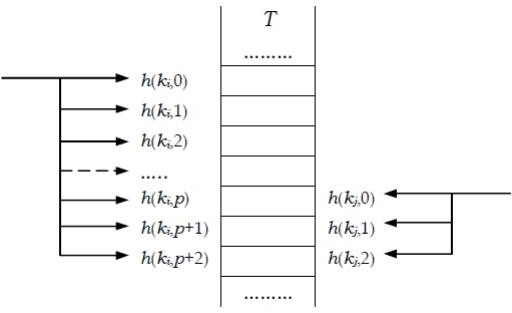
U toj situaciji vrijedi h(kj, i) = h(ki, p+i) za neko p i za i=0, 1, 2, itd. • Drugim riječima, primarno

grupiranje nastaje kada se ispitni nizovi za neke ključeve ki i kj u određenoj mjeri podudaraju (slika).

Pojava primarnog grupiranja se javlja uvijek u situacijama kada se sljedeća adresa u ispitnom nizu izračunava samo na temelju prethodne adrese.

Postoje različiti načini kako se mogu izbjeći pojave primarnog grupiranja. Jedan od jednostavnih a

efikasnih načina je kvadratno ispitivanje



## Kvadratno ispitivanje

Kvadratno ispitivanje koristi heš funkciju sljedećeg oblika:



gdje je

h0 - funkcija početnog ispitivanja

c1 i c2 su pomoćne konstante, s tim da c2≠ 0

i – oznaka ispitivanja (i=0, 1, 2, ... n-1)

Prvo se ispituje lokacija s adresom h0(k), a zatim se ispituju lokacije s određenim skokom, koji ovisi o rednom broju ponovnog ispitivanja i, na kvadratni način.

Iako metoda kvadratnog ispitivanja općenito daje nešto bolje rezultate od linearnog ispitivanja, ipak jedan od najvažnijih nedostataka je što kod kvadratnog ispitivanja ispitni niz u općem slučaju ne sadrži sve adrese u tabeli.

To znači da su moguće situacije u kojima, pri umetanju novog elementa, heš funkcija ne osigurava

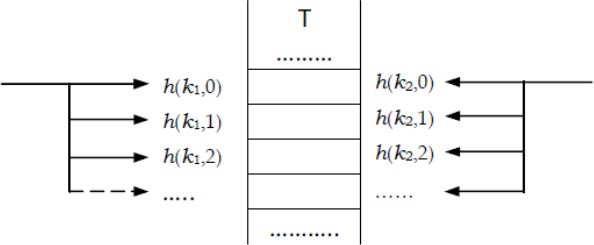
pristup nekim lokacijama, iako su one slobodne.

Međutim, prikladnim izborom koeficijenata c1 i c2, kao i vrijednosti n, može se omogućiti puno korištenje heš tabele

## Dvostruko heširanje

Ako dva ključa imaju istu matičnu adresu, kvadratno ispitivanje će generirati isti ispitni niz, što može

voditi stvaranju tzv. sekundarnog grupiranja (slika).



Problem sekundarnog grupiranja se može riješiti primjenom tzv. dvostrukog heširanja, jer ova

metoda generira ispitne nizove koji imaju mnoge karakteristike slučajno izabranih nizova ispitivanja, što ovu metodu čini jednom od najboljih metoda u okviru otvorenog adresiranja.

Dvostruko heširanje koristi dvije neovisne funkcije h1 i h2, što možemo prikazati u sljedećem obliku:

ℎ (𝑘, 𝑖) = (ℎ1 (𝑘) + 𝑖(ℎ2(𝑘)) 𝑚𝑜𝑑 𝑛 , 𝑖 = 0, 1, 2, … , 𝑛 – 1

Početno ispitivanje se odnosi na poziciju s adresom h1(k), a sljedeće pozicije koje se ispituju se

dobivaju pomakom u iznosu h2(k) mod n, u odnosu na prethodne pozicije.

Ovakav način heširanja možemo promatrati kao linearno ispitivanje kod kojeg pomak nije

predefinirana konstanta, već taj pomak ovisi od ključa i izračunava se pomoću druge funkcije h2.

Na taj način kod dvostrukog heširanja ispitni niz zapravo ovisi dvostruko od ključa k.

Dakako, i kod dvostrukog heširanja postoji određena vjerojatnost pojave sekundarnog grupiranja, koja se javlja pri umetanju elemenata sa ključevima k1 i k2, u situaciji kada je:

h1(k1)=h1(k2)

i h2(k1)=h2(k2).

Međutim, ta vjerojatnost je ipak mnogo manja kod dvostrukog heširanja, nego kod ostalih

prethodno opisanih metoda.

Da bi se osiguralo da ispitni niz sadrži sve adrese, potrebno je obratiti pažnju na izbor funkcije h2.

Funkcija h2 mora biti uzajamno prosta sa n.

Jedan od načina da se to osigura je da n bude stepen broja 2, a da se h2 oblikuje tako da uvijek daje neparan broj u rasponu 1 do n-1.

Druga mogućnost je da n bude prost broj, a da se funkcija h2 oblikuje tako da daje bilo koju vrijednost između 1 do n-1.

Primjer izbora za h1 i h2 bi mogao biti:

h1(k) = k mod n,

h2(k) = 1+ k mod n‘

gdje je n' broj izabran tako da je malo manji od broja n (npr. n'=n-1 ili n'=n-2).

Na slici je ilustriran primjer umetanja elemenata u heš tabelu upotrebom dvostrukog heširanja.

Korištena je tabela veličine n=13, a funkcije h1 i h2 su u ovom primjeru definirane kao:

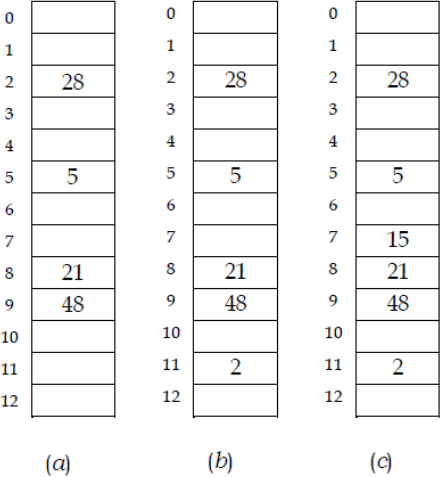
h1(k) = k mod 13,

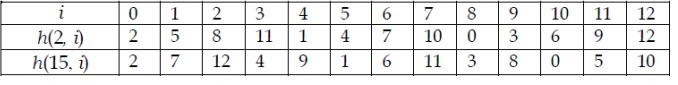
h2(k) = 1+(k mod 11).

Pretpostavimo da se u četiri operacije umeću sljedeći brojevi: 21, 28, 5 i 48.

Svi navedeni brojevi će biti umetnuti na svoje matične adrese, koje su definirane funkcijom h1.

Stanje u heš tabeli u tom trenutku je prikazano na slici a

•



Zatim, pretpostavimo da se u sljedeća dva koraka umeću brojevi 2 i 15.

Oba broja imaju istu matičnu adresu 2. U tabeli su, radi preglednosti, prikazani izračunati nizovi ispitivanja za oba broja koji se umeću.

Pri umetanju broja 2 prvo se ispituje matična adresa 2.

Nakon što se utvrdi da je ta adresa zauzeta, koristi se niz ispitivanja definiran heš funkcijom h.

Iz tabele vidimo da je prva lokacija ponovnog ispitivanja (i=1) lokacija s adresom 5

Međutim, i ta lokacija je zauzeta, pa se u drugom ponovnom ispitivanju (i=2) prelazi na sljedeću lokaciju s adresom 8, koja je također zauzeta.

Tek u trećem ponovnom ispitivanju (i=3), na lokaciji s adresom 11, se pronalazi slobodno mjesto na koje se pohranjuje broj 2 (slika b).

Pri umetanju sljedećeg broja 15, ponovno se kreće od zauzete lokacije s matičnom adresom 2.

Za razliku od umetanja broja 2, pri umetanju broja 15, već se u prvom ponovnom ispitivanju (i=1) pronalazi slobodna lokacija s adresom 7 (slika c).

Iako pri umetanju oba prethodna broja (2 i 15) nije bilo potrebno koristiti čitav ispitni niz, ipak su u tabeli prikazani ispitni nizovi u cjelini, da bi se ilustrirala činjenica da su generirana dva različita

ispitna niza, što omogućuje izbjegavanje pojave sekundarnog grupiranja.

## Rješavanje kolizija ulančavanjem

Prethodno smo vidjeli da razrješavanje kolizija otvorenim adresiranjem ima nedostatak koji se

ogleda u činjenici smanjivanja efikasnosti pretraživanja u situacijama kada se nad heš tabelom često

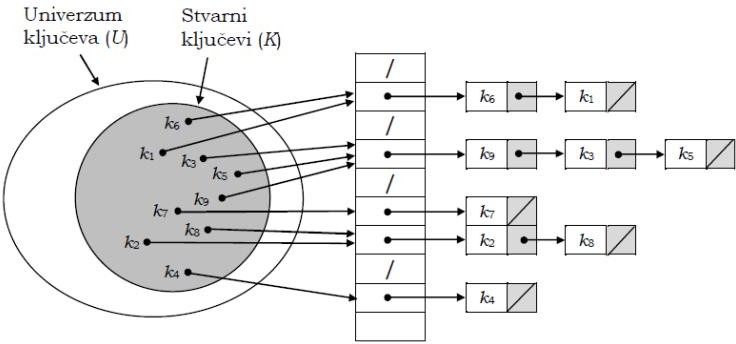
obavljaju operacije brisanja.

Osim toga, ako je heš tabela u većoj mjeri popunjena, tada se pri pretraživanju povećava i broj slučajeva u kojima se ispituju ključevi koji nisu sinonimi traženog ključa.

Pod sinonimima podrazumijevamo ključeve koji se preslikavaju u istu matičnu adresu heš tabele.

Ovi nedostaci nisu karakteristični za tehniku rješavanja kolizija koja se naziva ulančavanje

Kod ulančavanja se svi elementi, koji se heš funkcijom preslikavaju u istu adresu, pohranjuju u jednu povezanu listu (slika).



Svaka lokacija j u tabeli T sadrži pokazivač na čelo liste u kojoj su pohranjeni svi elementi koje heš

funkcija preslikava u adresu j.

Na primjer, ako je h(k9)=h(k3)=h(k5), onda su elementi s ključevima k9, k3 i k5 pohranjeni u istu listu

(slika).

Ako se nijedan element ne preslikava u adresu j, tada ta lokacija sadrži vrijednost NIL. Prema tome, svaka lokacija tabele T sadrži pokazivač na listu sinonima, koja je pridružena nekoj matičnoj adresi, pri čemu se elementi pohranjuju u listama kao dinamičkim strukturama.

Kada se umeće novi element, heš funkcija se koristi da se pronađe povezana lista u koju će se

pohraniti taj element, a zatim se koristi standardan postupak za umetanje elemenata u listu.

Ako se dogodi kolizija, na primjer na adresi j, tada jednostavno dodajemo novi čvor u listu, čije

zaglavlje je pohranjeno na adresi j

Ako pretpostavimo da se novi ključ dodaje u listu podrazumijevajući da već nije prisutan u tabeli, onda je umetanje tog ključa u odgovarajuću povezanu listu vrlo brzo i u najgorem slučaju ta

operacija ima vremensku složenost O(1).

Ako nije poznato da li je ključ već umetnut u povezanu listu, onda je pri njegovom umetanju prethodno potrebno provjeriti da li je on prisutan u listi, što zahtijeva dodatno vrijeme.

Novi ključ se može npr. dodati na početak liste, ako znamo da taj ključ već nije prisutan u listi, ili na kraj liste, ako je potrebno provjeriti da li je taj ključ već prethodno umetnut u listu

Kod pretraživanja, heš funkcijom prvo izračunavamo adresu na kojoj je pokazivač na čelo liste, koja će se pretraživati.

Ako se na izračunatoj adresi nalazi vrijednost NIL, to znači da traženi ključ nije prisutan niti u jednoj

listi.

S druge strane, ako se na izračunatoj adresi nalazi neki pokazivač različit od NIL, onda se

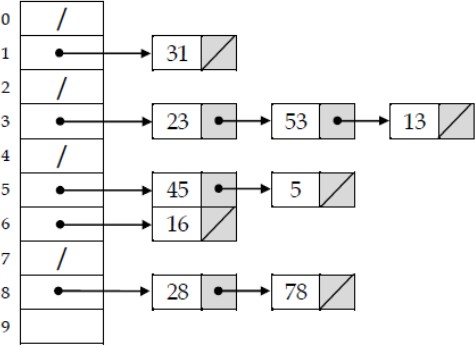
sekvencijalno pretražuje lista, da se utvrdi da li je traženi ključ prisutan u listi.

Ako se pretraživanjem liste traženi element ne pronađe, pretraživanje je neuspješno

Na sljedećoj slici je ilustrirano rješavanje kolizija ulančavanjem na primjeru tabele sa 10 lokacija, pri

čemu je korištena heš funkcija h(k) = k mod 10.

Na primjer, pri umetanju ključa k=13, heš funkcija daje vrijednost 3, pa se u listu, čije se zaglavlje nalazi na adresi 3, umeće ključ k=13.



U primjeru prikazanom na slici, ključ k=13 je umetnut na kraj liste. Međutim, ako želimo povećati efikasnost umetanja, povezane liste se mogu održavati sortiranim po vrijednostima ključa.

To znači da se pri umetanju, ključ treba postaviti na mjesto koje mu po poretku pripada.

Isto tako, upotreba sortiranih povezanih listi kod ulančavanja povećava efikasnost kod neuspješnog pretraživanja

Kod operacije brisanja se prvo heš funkcijom izračuna adresa lokacije na kojoj je pohranjeno čelo liste u kojoj će se tražiti element koji se briše, a zatim se u dotičnoj listi pronalazi ključ koji se želi obrisati.

Brisanje ključa iz liste se radi na uobičajeni način, a to znači da se ta operacija svodi na povezivanje

liste.

Prema tome, operacija brisanja kod heširanja ulančavanjem je jednostavnija i efikasnija u odnosu na

otvoreno adresiranje

Algoritmi osnovnih operacija kod razrješavanja kolizija ulančavanjem su u osnovi isti kao i kod povezanih listi, s jedinom razlikom da je kod ulančavanja još dodatno potrebno izračunavati heš vrijednost.

Ako su dimenzija niza s pokazivačima i heš funkcija dobro odabrani, onda se mogu očekivati male prosječne dužine povezanih listi.

U tom slučaju možemo reći da su i operacije kod razrješavanja kolizija ulančavanjem efikasnije.

Sljedeća prednost ulančavanja je u tome što broj ključeva nije ograničen, već se potrebni memorijski

prostor za pohranjivanje novih elemenata dinamički alocira.

Osim toga, kod otvorenog adresiranja se za razrješavanje kolizija koristi niz ispitivanja, pri čemu se za svako novo ispitivanje treba izračunavati heš vrijednost, dok se kod ulančavanja pri koliziji samo slijede pokazivači u povezanoj listi.

Jedini nedostatak ulančavanja je u tome što se za pokazivače mora koristiti dodatni memorijski

prostor.

Ovaj memorijski prostor može biti značajan, ako je odnos potrebnog memorijskog prostora za smještanje elemenata, u odnosu na potreban memorijski prostor za smještanje pokazivača, relativno mali.

Na kraju još spomenimo, da bi se umjesto povezanih listi mogla koristiti i binarna stabla, ako bi broj elemenata u povezanim listama bio velik, što se uz prikladno dimenzioniranu heš tabelu i dobro dizajniranu heš funkciju ne bi trebalo očekivati

## Definicije i terminologija

**(Predavanje 13) Sortiranje**

Metoda koje se koristi kada treba da se podaci organiziraju na neki unaprijed određen način • Svaki od algoritama sortiranja ima svoje prednosti i nedostatke – posljedica s obzirom na potrebe odabire se i algoritam

Npr. ako je stepen uređenosti polaznih podataka visok onda se odabire takvo sortiranje koje ovo podržava

## Sortiranje umetanjem

Algoritam sortiranja umetanjem se temelji na međusobnom poređenju vrijednosti ključeva, pri čemu se u svakoj iteraciji održavaju sortirani i nesortirani dio niza.

U svakom sljedećem koraku se iz nesortiranog dijela uzima jedan element i taj element se umeće na odgovarajuće mjesto u sortiranom dijelu.

Prema tome, svakim novim korakom sortiranja, nesortirani dio niza se smanjuje, dok se sortirani dio

niza postepeno povećava

Algoritam sortiranja umetanjem ćemo objasniti na primjeru sortiranja niza A[0..n-1]={a0, a1, a2, ..., an-1} u rastućem poretku.

Razmotrimo prvi korak i neki i-ti korak u procesu sortiranja.

Na početku izvršavanja algoritma, sortirani dio se sastoji samo od prvog elementa a0, dok svi ostali elementi čine nesortirani dio niza

U prvom koraku se porede elementi a1 i a0 (drugi i prvi element u nizu) .

Ako je element a0 veći od elementa a1, tada se element a1 pohranjuje u privremenu lokaciju priv, element a0 se pomjera na drugo mjesto (i=1), te se na prvo mjesto u nizu s indeksom i=0 pohranjuje element a1 iz privremene lokacije priv. U ovom trenutku sortirani dio niza sadrži dva elementa (a0 i a1), a nesortirani dio sadrži n-2 elemenata (a2, a3,...an-1).

Opišimo sada neki i-ti korak sortiranja.

Poslije prethodnog (i-1)-vog koraka u sortiranom dijelu se nalaze elementi a1, a2,...ai-1. Zatim se u i-

tom koraku iz nesortiranog dijela uzima element ai, nakon čega se prema kraju niza pomjeraju svi elementi iz sortiranog dijela niza koji su veći od elementa ai, da bi se oslobodio prostor za

pohranjivanje tog elementa na odgovarajuće mjesto u nizu.

Na taj način se veličina sortiranog dijela niza povećava sa i-1 na i. Sortiranje se završava kada u nesortiranom dijelu niza nema više elemenata, odnosno kada zadnji element an-1 zauzme svoju odgovarajuću poziciju

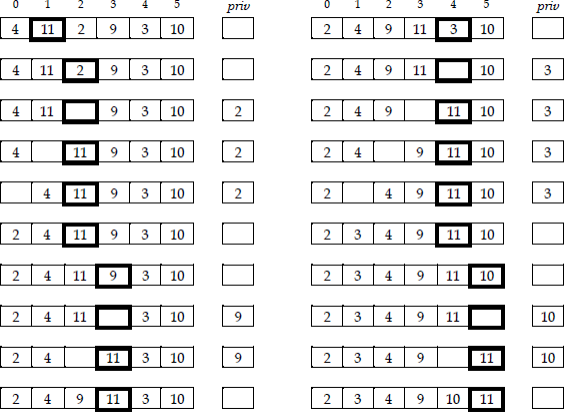
Algoritam za sortiranje umetanjem je ilustriran po koracima na slici, za jedan konkretan primjer

početnog nesortiranog niza, koji sadrži 6 elemenata.

Podebljanim linijama su istaknuti elementi za koje se u sortiranom dijelu niza u pojedinim fazama

pronalazi odgovarajuće mjesto.

Algoritam počinje usporedbom elemenata 11 i 4. Pošto je 11 > 4, nije potrebno nikakvo pomjeranje, te se prelazi na sljedeću lokaciju (i=2), što je na slici ilustrirano podebljanom ćelijom u kojoj se nalazi element 2.



U ovom koraku se porede elementi 2 i 11. Budući da je 2 < 11, element 2 se upisuje u privremenu

lokaciju priv, a element 11 se pomjera za jedno mjesto udesno.

Zatim se uspoređuju elementi 2 i 4. Pošto je 2 < 4, element 4 se pomjera sa prvog na drugo mjesto,

oslobađajući prostor za pohranjivanje elementa 2 iz privremene lokacije priv.

U ovom trenutku sortirani dio niza čine elementi 2, 4, i 11, dok nesortirani dio niza čine elementi 9, 3

i 10.

Opisani postupak se ponavlja za sve preostale elemente u nizu, sve dok se na odgovarajuće mjesto

ne umetne i zadnji element 10.

Ostali koraci algoritma do samog kraja procesa sortiranja su također ilustrirani na slici

Algoritam sortiranja umetanjem je opisan procedurom INSERTION-SORT na slici.

Ulaz u algoritam je niz ključeva A[0..n-1], a nakon završetka algoritma niz A je sortiran

**INSERTION-SORT (A)**

**1 for i** 🡨 **1 to duzina[A] -1 do**

**2 priv** 🡨 **A[i]**

**3 j** 🡨 **i-1**

**4 while (j >= 0 and A[j] > priv) do**

**5 A[j+1]<--A[j]**

**6 j**🡨**j-1**

**7 end\_while**

**8 A[j+1]** 🡨 **priv**

**9 end\_for**

Vanjska for petlja omogućuje da se, počevši od drugog elementa iz nesortiranog dijela niza, uzima

element A[i] (i=1,2,...duzina[A]-1) i umeće na odgovarajuću poziciju.

U while petlji se elementi iz sortiranog dijela niza pomjeraju udesno, sve dok se ne pronađe odgovarajuće mjesto za element A[i] (linije 4-7).

Nakon izlaska iz while petlje, pronađena pozicija ima indeks j+1, pa se element A[i], koji je pohranjen

u varijabli priv, smješta na tu poziciju (linija 8).

## Sortiranje umetanjem – efikasnost

Razmotrimo efikasnost algoritma sortiranja umetanjem. Najbolji slučaj se pojavljuje u situaciji kada je niz već sortiran, pa je za pozicioniranje svakog elementa potrebna samo jedna operacija

poređenja.

Tom operacijom se zapravo samo utvrđuje da je prethodnik svakog elementa manji od tog

elementa, te svi elementi ostaju na svojim mjestima.

To znači da je u najboljem slučaju ukupan broj operacija usporedbi Tc(n):

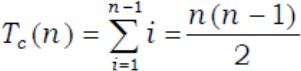


Prema tome, vremenska složenost za najbolji slučaj je O(n).

Najgori slučaj se pojavljuje u situaciji kada je za umetanje svakog elementa iz nesortiranog dijela niza

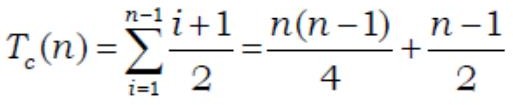
potrebno porediti taj element sa svim elementima u sortiranom dijelu niza.

Drugim riječima, najgori slučaj je kada se svaki element trenutno pozicionira na početak sortiranog dijela niza. To se događa kada je niz sortiran u obrnutom poretku. Tada je ukupni broj operacija usporedbi Tc(n):



vremenska složenost u najgorem slučaju je O(n2).

U prosječnom slučaju možemo pretpostaviti da će za umetanje nekog elementa na odgovarajuću poziciju biti potrebno da se taj element poredi sa pola već sortiranog niza, pa je prosječan broj usporedbi po iteraciji (i+1)/2. To znači da je ukupan broj operacija usporedbi u prosječnom slučaju Tc(n):



u prosječnom slučaju je vremenska složenost O(𝑛 2 ).

## Poboljšanja

Prethodno smo vidjeli da se kod algoritma INSERTION-SORT obavlja pretraživanje sortiranog dijela niza, da bi se odredila pozicija umetanja prvog elementa iz nesortiranog dijela niza.

Upravo na svojstvu da je prvi dio niza sortiran se temelji poboljšanje efikasnosti kod kojeg se pozicija umetanja pronalazi binarnim pretraživanjem, umjesto sekvencijalnim pretraživanjem, kao što je to slučaj u osnovnoj varijanti algoritma INSERTION-SORT.

U toj poboljšanoj varijanti prosječan broj operacija usporedbi u svakoj iteraciji se smanjuje sa i/2 na približno l+log2 i.

To znači da je u poboljšanoj varijanti vremenska složenost za operacije poređenja reda O(n log n), što je znatno poboljšanje, ali samo za te operacije.

Međutim, efikasnim pronalaženjem mjesta umetanja se ne smanjuje i broj potrebnih operacija premještanja, koji zapravo ostaje isti kao i u osnovnoj varijanti.

Taj broj operacija je dominantan u odnosu na broj operacija usporedbi za velike vrijednosti n, pa

ukupna vremenska složenost algoritma ostaje O(𝑛 2 ).

## Shell Sort

U osnovnoj varijanti algoritma sortiranja umetanjem elementi se pomjeraju za samo jednu poziciju,

što za posljedicu ima veliki broj premještanja.

Efikasnost algoritma se može poboljšati na način da se uspoređuju elementi koji su odvojeni nekim razmakom, te da se umetanje elemenata obavlja tako da se prave premještanja elemenata sa skokovima većim od jednog mjesta razmaka.

Uzmimo, na primjer, da imamo element koji ima neku malu vrijednost koja se prije sortiranja nalazi pri kraju niza, te da želimo sortirati niz u rastućem poretku. Koristeći osnovnu varijantu algoritma sortiranja umetanjem bio bi potreban veliki broj operacija poređenja i premještanja dok taj element ne bi bio umetnut na odgovarajuću poziciju.

Efikasnost možemo poboljšati tako da se elementi prvo pomjeraju prema konačnoj poziciji koristeći

velike skokove, tako da bi se, na primjer, element sa nekom malom vrijednošću pomjerio prema konačnoj poziciji uz samo nekoliko operacija poređenja i premještanja.

Poboljšanje koje se temelji na prethodno opisanoj strategiji je predložio Donald Shell, po kojem je algoritam nazvan Shell sort

Prethodno je istaknuto da je osnovna varijanta algoritma umetanjem efikasna ako ulazni niz već ima visok stupanj uređenosti.

Algoritam Shell sort razdvaja početni niz na grupe, tako što se u svakoj grupi nalaze elementi koji su

odvojeni jednakim razmakom u nizu, kojeg ćemo označiti sa h1.

Zatim se svaka grupa sortira posebno primjenom osnovne varijante sortiranja umetanjem, nakon

čega dobivamo niz sa većim stupnjem uređenosti u odnosu na početni niz.

U sljedećem koraku se uzima razmak h2, koji je manji od razmaka h1.

U ovom koraku je smanjen broj grupa, ali svaka grupa ima veći broj elemenata

Ponovno se grupe posebno sortiraju primjenom osnovne varijante algoritma sortiranja‚ umetanjem, te se dobija još veći stupanj uređenosti ulaznog niza. U sljedećim koracima se biraju sve manji i manji razmaci, a postupak završava primjenom završnog razmaka ht=1, što zapravo znači da se u zadnjoj fazi svi elementi tretiraju kao jedna grupa.

Drugim riječima, u zadnjem koraku, kada je ulazni niz „već gotovo sortiran“, primjenjujemo osnovnu

varijantu sortiranja umetanjem

Dakle, u prethodno opisanim fazama algoritma sortiranja Shell sort, koristili smo niz razmaka h= [h0,

h1, …, ht-1], pri čemu je hi > hi+1, za i=1, 2, …, t-2.

## Primjer

Pošto algoritam Shell sort uspoređuje brojeve iz niza uzimajući neki razmak, pretpostavimo da je u prvoj fazi taj razmak 7. To znači da će se uspoređivati i međusobno eventualno razmjenjivati mjesta, na primjer, sljedeći brojevi: 28, 33 i 59 (jer su ovi brojevi razmaknuti za 7 mjesta). Isto tako,

uspoređivat će se i eventualno međusobno razmjenjivati mjesta sljedeći brojevi: 17,12 i 49 (jer su

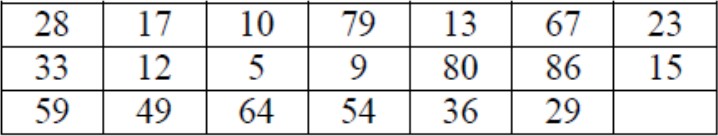
također razmaknuti za 7 mjesta).



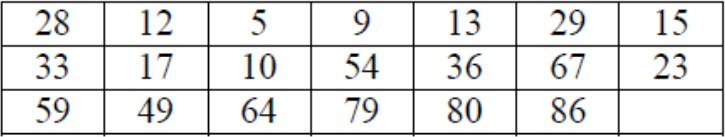
Da bi ovaj proces što jasnije prikazali, možemo se poslužiti jednim malo drugačijim načinom prikaza početnog niza. Naime, ako smo izabrali u ovoj fazi algoritma razmak 7, onda gornji početni niz

brojeva možemo prikazati sljedećom tabelom koja ima 7 kolona, pri čemu svaka kolona sadrži

elemente pojedine grupe, koji se posebno sortiraju:



Napomenimo da je ovakav prikaz napravljen samo da bi vizualizirali procese poređenja i sortiranja pojedinih grupa, koji se obavljaju u pojedinim fazama primjene algoritma Shell sort. Sada sortirajmo svaku kolonu (grupu) posebno. Rezultirajuća tabela je



Gornju tabelu ćemo sada prikazati u obliku jednodimenzionalnog niza, što zapravo i odgovara

stvarnom rasporedu elemenata u memoriji

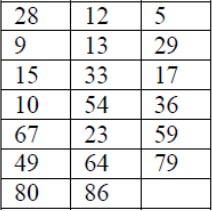


Možemo primijetiti da rezultirajući niz u trenutnoj fazi sortiranja, kada smo uzeli razmak 7, ima veći stupanj uređenosti od početnog niza.

Naime, kao rezultat prve faze imamo pomijeranja elemenata sa većim vrijednostima prema kraju niza, te pomijeranja elemenata sa manjim vrijednostima prema početku niza. Primijetimo, na

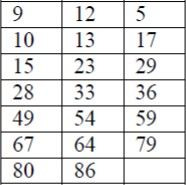
primjer, da se element 29 sa zadnje pozicije pomjerio na šesto mjesto, dok se, na primjer, element 36 sa predzadnjepozicije pomjerio na dvanaesto mjesto.

Postupak primjene algoritma Shell sort nastavljamo tako što ćemo sada uzeti, na primjer, razmak h2=3. To znači da ulazni niz u drugoj fazi, za potrebe vizualizacije elemenata koji se tretiraju kao posebne grupe, možemo prikazati u obliku sljedeće tabele sa 3 kolone:



Opet se grupe (kolone) sortiraju posebno primjenom osnovne varijante sortiranja umetanjem, te

dobijamo sljedeću tabelu kao i rezultirajući niz:



Primijetimo da rezultirajući niz, u trenutnoj fazi sortiranja, ima još veći stupanj uređenosti u odnosu na niz u prethodnoj fazi. Naime, nakon provedene prethodne faze, došlo je do daljnjeg pomijeranja većih brojeva prema kraju niza i manjih brojeva prema početku niza.

Međutim, trenutni rezultirajući niz ipak još nije sortiran, te je ostalo da napravimo još jednu fazu

sortiranja, nakon čega će niz biti konačno sortiran. Pošto je trenutni rezultirajući niz „skoro sortiran”, razumno je zapravo uzeti razmak h3=1, a to znači primijeniti osnovnu varijantu algoritma sortiranja umetanjem, jer će biti potreban mali broj poređenja i premještanja s obzirom da je niz već „skoro sortiran”.

Dakle, primjenom algoritma sortiranja umetanjem s razmakom h3=1, dobit ćemo rezultirajući

sortirani niz.



Prethodno opisani algoritam je prikazan u obliku procedure SHELL-SORT. Ulazni argumenti procedure su niz A koji se sortira i niz razmaka H. Dio algoritma koji je omeđen unutarnjom for petljom (linije 3-11) zapravo predstavlja modificirani Insertion sort, pri čemu se modifikacija sastoji u tome da se u nekoj i-toj iteraciji porede elementi s razmakom h=H[i] (linija 2), umjesto susjednih elemenata, kako je to učinjeno u osnovnoj varijanti.

Vanjska for petlja omogućuje ponavljanje izvođenja te modificirane varijante

algoritma Insertion sort, za svaki od razmaka H[i], i=0, 1,...duzina[H]-1 (linija 1).

Eksperimentalni rezultati pokazuju da se vremenska složenost ovog algoritma može aproksimirati

kao O(𝑛 1.3 ).

**SHELL-SORT (A, H)**

**for i <-- 0 to duzina [H]-1 do**

**h** 🡨 **H[i]**

**for j** 🡨 **h to duzina [A ] – 1 do**

**priv** 🡨 **A[j]**

**k** 🡨 **j-h**

**while ( k >= 0) and (A[k] >priv) do**

**A[k+h]** 🡨 **A[k]**

**k** 🡨 **k-H**

**end\_while**

**A[k+h]<--priv**

**end\_for**

**end\_for**

## Sortiranje selekcijom

Jedan od nedostataka algoritma Insertion sort je to što zahtijeva veliki broj operacija pomijeranja unutar sekvence. Doduše, ovaj nedostatak nije prisutan ako su podaci koji čine neki linearni niz implementirani kao povezane liste. Primjer jednostavnog algoritma sortiranja, koji je oblikovan tako da se bitno smanjuje broj premještanja, je upravo algoritam Selection sort (sortiranje selekcijom)

Kod ovog algoritma se pomjeranje podataka pri sortiranju obavlja direktno na njegovu konačnu

poziciju u sortiranom nizu.

Slično kao i kod algoritma Insertion sort, i kod algoritma Selection sort se u procesu sortiranja

razlikuju sortirani i nesortirani dio niza.

Na početku svi elementi niza A[0..n-1]={a0, a1, ..., ai, ai+1,...an-1} pripadaju nesortiranom dijelu, a zatim se sekvencijalnim pretraživanjem pronalazi najmanji element, koji se zatim premješta na prvu poziciju, dok se prethodni element s prve pozicije premješta na poziciju pronađenog minimalnog elementa.

Na taj način je u prvom koraku oblikovan sortirani dio niza koji čini element a1, dok ostali elementi niza (a2, a3, ..., an-1) čine nesortirani dio niza. Poslije nekog (i-1)-vog koraka u sortiranom dijelu niza su elementi: a0, a1, ..., ai-2; dok su u nesortiranom dijelu niza elementi: ai-1, ai,..., an-1.

U sljedećem i-tom koraku se u nesortiranom dijelu niza pronalazi najmanji element, te se taj

element pozicionira na prvo mjesto u neuređenom dijelu niza, a to je pozicija i-1.

U tom trenutku sortirani dio niza je: a0, a1, ..., ai-1; dok je nesortirani dio niza: ai, ai+1, ..., an 1. Postupak sortiranja se završava kada u nesortiranom dijelu na poziciji n-1 preostane samo jedan element, koji zapravo ostaje na toj poziciji.

Primijetite da se poređenje elemenata kod sortiranja selekcijom obavlja nad elementima nesortiranog dijela niza, za razliku od sortiranja umetanjem, kod kojeg se usporedba elemenata obavlja nad elementima sortiranog dijela niza.

Ova razlika ima za posljedicu da postupak sortiranja selekcijom ne može započeti, sve dok nisu prisutni svi elementi koji se trebaju sortirati, što nije slučaj kod sortiranja umetanjem, kod kojeg elementi koji se sortiraju mogu pristizati jedan za drugim

## Primjer

Algoritam za sortiranje selekcijom je ilustriran po koracima na slici za jedan primjer početnog nesortiranog niza, koji sadrži 6 elemenata.

Podebljanim linijama su istaknute pozicije niza za koje se u procesu sortiranja u pojedinim fazama selekcionira minimalni element iz nesortiranog dijela niza, dok osjenčena pozicija predstavlja poziciju na kojoj je pronađen minimalni element, također u nesortiranom dijelu niza.

U prvom koraku je označena prva pozicija sa indeksom 0, na kojoj se nalazi element 4, te je pronađen minimalni element 2, na poziciji s indeksom 2 (slika a).

U sljedećem koraku elementi 4 i 2 mijenjaju pozicije, te se u nizu razlikuju sortirani dio niza, koji se sastoji samo od elementa 2, i nesortirani dio niza, koji uključuje sve ostale elemente (slika b).

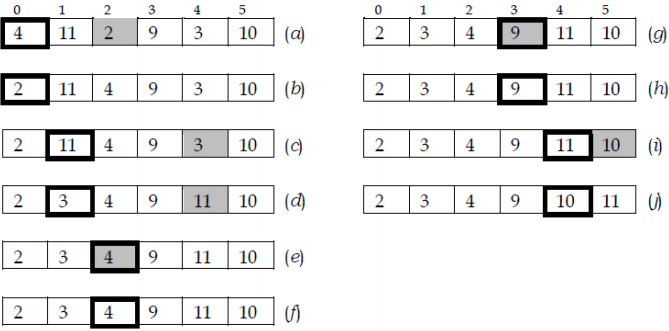
U sljedećem koraku je označena pozicija s indeksom 1, na kojoj se nalazi element 11, te je u nesortiranom dijelu niza pronađen minimalni element 3 (slika c), pa elementi 11 i 3 mijenjaju pozicije u nizu (slika d).

U tom trenutku sortirani dio niza čine elementi 2 i 3, dok elementi 4, 9, 11 i 10 čine nesortirani dio niza.

Opisani postupak se ponavlja za sve ostale pozicije i (i=2, 3, ..., n-2), sve dok se na pretposljednje mjesto u nizu ne umetne element 10.

Zadnji preostali element 11 zauzima jedinu moguću (zadnju) poziciju, pa se postupak završava selekcioniranjem elementa za poziciju i=n-2.

Ostali koraci algoritma, do samog kraja procesa sortiranja selekcijom, su također ilustrirani na slici



Algoritam za sortiranje selekcijom je opisan procedurom SELECTION-SORT. Ulazni argument u proceduru je niz A koji se sortira.

Vanjska for petlja omogućuje ponavljanje postupka pronalaženja minimalnog elementa u

nesortiranom dijelu niza, za svaku od lokacija i (i=0, 1, 2,..., duzina[A]-2).

Za zadnju lokaciju (duzina[A]-1) nije potrebno ponavljati postupak, jer se u tom slučaju nesortirani

dio niza sastoji samo od jednog elementa, pa je taj podniz sortiran sam po sebi.

**SELECTION-SORT (A)**

**for i🡨 0 to duzina[A] -2 do**

**min 🡨 A[i]**

**pmin 🡨 i**

**for j 🡨 i+1 to duzina[A] -1 do**

**if (A[j] < min) then**

**min 🡨 A[j]**

**pmin 🡨 j**

**end\_if**

**end\_for**

**A[pmin] 🡨 A[i]**

**A[i] 🡨 min**

**end\_for**

U svakoj i-toj iteraciji se prvo inicijaliziraju vrijednosti privremeno minimalnog elementa min i indeksa njegove pozicije pmin (linije 2-3), a zatim se u sklopu unutarnje for petlje izvode iteracije kojima se u nekom j tom prolazu u nesortiranom dijelu niza A, kojeg čine elementi na pozicijama i do duzina[A]-1, pronalazi minimalni element (linije 4-9).

Na kraju svake i-te iteracije vanjske for petlje, element na poziciji pmin i prvi element A[i] iz nesortiranog dijela niza izmjenjuju svoje pozicije (linije 10-11).

## Efikasnost pridruživanja

Algoritam sortiranja selekcijom je specifičan po tome što je i broj pridruživanja i broj usporedbi

uvijek isti, bez obzira na dužinu niza.

Algoritam se izvršava u n-1 prolaza kroz nesortirani dio niza, a svaki prolaz ima jednu zamjenu

pozicija koja uključuje 3 operacije pridruživanja. To znači da je ukupan broj operacija pridruživanja

Ta(n): Ta(n)=3(n-1)

Dakle, operacije pridruživanja kod algoritma sortiranja selekcijom imaju složenost O(n), što je znatno poboljšanje u odnosu na algoritam sortiranja umetanjem, barem kada su u pitanju operacije

pridruživanja.

## Efikasnost usporedbi

U prvom prolazu ima n-1 operacija usporedbi, u drugom prolazu ih ima n-2, a u nekom i-tom prolazu broj usporedbi je n-i. Dakle, ukupan broj usporedbi Tc(n) je



Dakle, možemo zaključiti da kod algoritma sortiranja selekcijom dominira broj operacija usporedbi, te da je vremenska složenost O(𝑛 2 ). Nema najgoreg i najboljeg slučaja

## Bubble sort – sortiranje poređenjem susjednih elemenata

Jedan od najjednostavnijih algoritama sortiranja je algoritam Bubble sort.

Ovaj algoritam radi tako da više puta sekvencijalno prolazi kroz niz, pri čemu se pri svakom prolazu uspoređuje svaki element sa elementom koji se nalazi neposredno iza njega.

Ako se nakon usporedbe utvrdi da susjedni elementi nisu u željenom poretku, onda elementi međusobno izmjenjuju svoje pozicije, što ima za posljedicu da se prema kraju niza postepeno

pomjeraju elementi sa većim vrijednostima, ako je željeni poredak sortiranja rastući, ili da se prema kraju niza postepeno pomjeraju elementi sa manjim vrijednostima, ako je željeni poredak sortiranja opadajući.

U prvom prolazu kroz niz, na zadnjem mjestu, će se sigurno pozicionirati najveći (najmanji) element.

Na taj način sortirani dio niza čini element A[n-1], a nesortirani dio niza čine elementi A[0], A[1], …., A[n-2]. U sljedećem, drugom prolazu, se uspoređuju elementi na pozicijama 0 do n-2, te će se nakon tog prolaza, na poziciju s indeksom n-2, sigurno pozicionirati najveći element iz trenutno nesortiranog dijela niza A[0..n-2].

Općenito, prije nekog i-tog prolaza, za i > 1, sortirani dio niza čine elementi: A[n-i+1], A[n-i+2], …,

A[n-1]; dok nesortirani dio niza čine elementi: A[0], A[1], …, A[n-i].

U nekom i-tom prolazu poređenja i izmjene pozicija susjednih elemenata će na (n-i)-tu poziciju doći najveći element iz nesortiranog dijela niza A[0..n-i].

Sekvencijalni prolazi kroz niz se nastavljaju sve (n-1)-og prolaza kroz niz, odnosno sve dok se na

drugu poziciju u nizu s indeksom 1 ne pozicionira odgovarajući element.

Naime, n-ti prolaz kroz niz nije niti potreban, jer će preostati samo jedan element u nesortiranom dijelu niza i samo jedna dostupna pozicija s indeksom 0, pa je praktično niz nakon (n-1)-og prolaza već sortiran

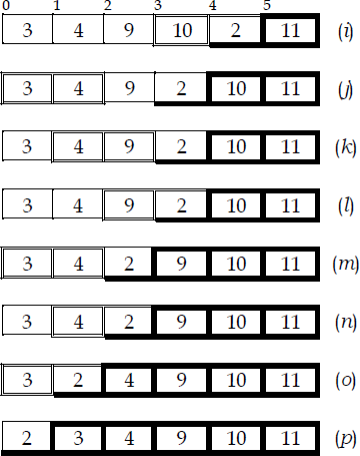
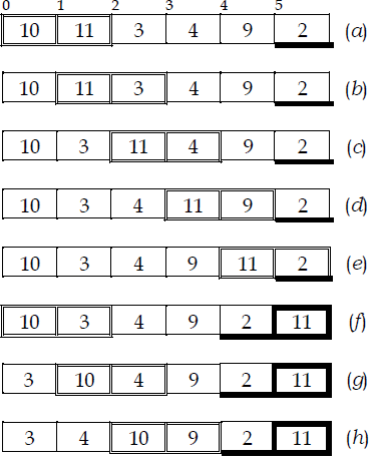
Algoritam Bubble sort je ilustriran po koracima na slici za jedan primjer početnog nesortiranog niza,

koji sadrži 6 elemenata.

Algoritam uključuje n-1 prolaza kroz nesortirani dio niza, pri čemu svaki prolaz osigurava smještanje najvećeg elementa iz nesortiranog dijela niza na zadnju poziciju tog dijela niza.

Ta pozicija je na slici za svaki prolaz istaknuta podebljanom linijom s donje strane. Podebljanim okvirom su označeni elementi koji se nalaze u sortiranom dijelu niza, odnosno elementi koji se nalaze na svojim konačnim pozicijama.

Susjedni elementi niza koji se trenutno uspoređuju su istaknuti dvostrukom linijom.



U prvom koraku su označene prva pozicija s indeksom 0, na kojoj se nalazi element 10, i druga

pozicija s indeksom 1, na kojoj se nalazi element 11 (slika a).

Pošto su elementi 10 i 11 u željenom poretku, ne dolazi do zamjene njihovih pozicija.

U sljedećem koraku se porede elementi 11 i 3 (slika b). Budući da se ovi elementi ne nalaze u željenom poretku, dolazi do zamjene njihovih pozicija. Prvi prolaz kroz niz se nastavlja, sve dok varijabla j ne dostigne vrijednost j=5. Nakon prvog prolaza će se najveći element (11) nalaziti na svojoj konačnoj poziciji s indeksom i=5 (slika f). U ovom trenutku sortirani dio niza čini jedino element 11, dok svi ostali elementi predstavljaju nesortirani dio niza.

U drugom prolazu kroz niz (i=4), na poziciju s indeksom 4, će se pozicionirati najveći element iz

nesortiranog dijela niza, a to je element 10. Nakon drugog prolaza sortirani dio niza čine elementi 10 i 11, a nesortirani dio niza čine elementi: 3, 4, 9 i 2 (slika j).

Opisani postupak se ponavlja za sve ostale vrijednosti varijable i (i=3, 2, 1), sve dok se na drugo mjesto u nizu ne umetne element 3. Jedini preostali element u nesortiranom dijelu niza je element 2, koji na kraju zauzima jedinu slobodnu poziciju s indeksom 0 (slika p).

Prethodno opisani algoritam je prikazan u obliku procedure BUBBLE-SORT.

Ulaz u proceduru je niz A koji se sortira. Vanjskom for petljom (linije 1-7) se implementira duzina[A]- 1 prolaza kroz nesortirani dio niza, čime se osigurava smještanje najvećeg elementa na zadnju poziciju tog dijela niza.

Unutarnjom for petljom se implementira jedan prolaz, u kojem se nakon poređenja susjednih elemenata i utvrđivanja da nisu u željenom poretku izmjenjuju njihove pozicije (linije 2-6).

**BUBBLE-SORT (A)**

**for i 🡨 duzina [A] -1 to 1 do**

**for j 🡨 1 to i do**

**if (A[j-1] > A[j]) then**

**A[j-1] < -- > A[j]**

**end\_if**

**end\_for**

**end\_for**

## Modificirani Bubble sort

Pretpostavimo da je ulazni niz A, kao argument procedure BUBBLE-SORT, već sortiran. U toj situaciji će broj potrebnih operacija ostati opet isti, bez obzira na početnu uređenost niza. Međutim, da bi se u određenoj mjeri povećala efikasnost, moguće je oblikovati takvu varijantu algoritma Bubble sort

koja će detektirati prolaz kroz niz u kojem nije došlo niti do jedne zamjene pozicija, što je zapravo znak da je niz već sortiran.

Nakon takve detekcije, proces sortiranja se može prekinuti da bi se izbjegle nepotrebne operacije usporedbi. Isto tako, postoji mogućnost da je samo jedan dio niza već sortiran, pa se efikasnost može povećati i tako što će se identificirati taj dio niza, te u sljedećoj iteraciji neće biti potrebno prolaziti kroz taj dio niza.

Detekciju prolaza kroz niz, u kojem nije došlo niti do jedne zamjene pozicija, a isto tako i

identificiranje dijela niza koji je već sortiran, možemo napraviti tako da se u svakom prolazu kroz niz zapamti pozicija sa najvećim indeksom j na kojoj je izvršena zamjena.

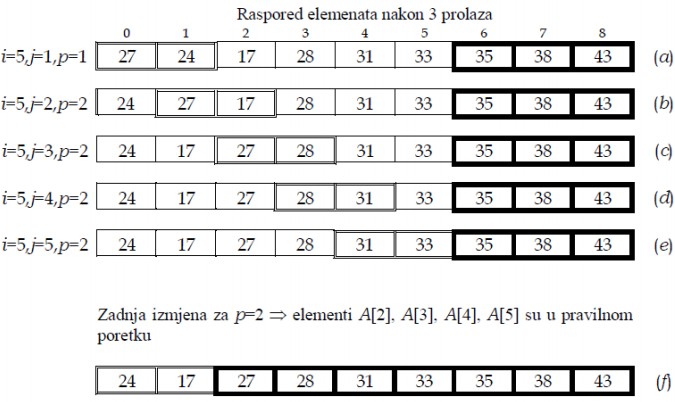
Pretpostavimo, na primjer, da se sortira niz koji ima n=9 elemenata, te da je nakon trećeg prolaza

raspored elemenata kao na slici a.

U četvrtom prolazu varijabla i ima vrijednost 5, dok se vrijednost varijable j mijenja od 1 do 5.

U ovom četvrtom prolazu, zadnja izmjena je izvršena kada je vrijednost varijable j bila 2, što zapravo znači da su elementi na pozicijama s indeksom 2, 3, 4 i 5 već u pravilnom poretku.

Pozicija zadnje zamjene j varijable je zapamćena u varijabli p.



Nadalje, to znači da u sljedećem prolazu neće biti potrebno uspoređivati elemente koji se nalaze na pozicijama s indeksom većim ili jednakim od p=2, pa se zapravo nakon četvrtog prolaza na svojim

konačnim pozicijama ne nalaze samo 4 elementa, već 7 elemenata kako je to prikazano na slici f.

Ako je nakon nekog od prolaza kroz niz vrijednost varijable p=0, to je znak da je čitav preostali dio niza zapravo već sortiran, pa se proces sortiranja može prekinuti

U skladu s prethodno opisanom mogućnošću poboljšanja, opisana je modificirana varijanta

algoritma u obliku procedure BUBBLE-SORT-M.

Varijabla p se koristi da se zapamti pozicija iznad koje u nekom prolazu kroz niz nije bilo izmjene pozicija (linija 8).

Prolazi se izvode u sklopu repeat-until petlje, te pri svakom novom prolazu se vrijednost varijable p postavlja na nulu (linija 4).

Ako se dogodi da u nekom prolazu već imamo sortiran poredak elemenata, tada će i na kraju prolaza biti p=0, pa će algoritam završiti svoj rad (linija 11).

Unutarnjom for petljom (linije 5-10) se ostvaruje jedan prolaz kroz nesortirani dio niza, ali samo do pozicije i=p-1 (linija 3), jer na pozicijama iznad p nije bilo izmjena, pa je taj dio niza već sortiran

**BUBBLE-SORT-M (A)**

**p 🡨 duzina[A]**

**repeat**

**i🡨 p-1**

**p🡨 0**

**for j 🡨 1 to i do**

**if (A[j-1] > A[j]) then**

**A [j-1] < -- > A[j]**

**p 🡨 j**

**end\_if**

**end\_for**

**until (p = 0)**

## Algoritmi sortiranja sa vremenskom složenošću O(n log n)

Merge sort,

Quick sort i

Heap sort.

## Merge sort

Algoritam Merge sort koristi strategiju „podijeli-pa-vladaj“, kod koje se do rješenja nekog problema

dolazi njegovim smanjivanjem na sve manje dijelove, sve dok se on ne reducira na tzv. osnovni

slučaj, koji se može riješiti direktno.

Naime, ovaj algoritam se temelji na ideji da se iz sortiranih podnizova na efikasan način mogu

spajanjem tih podnizova dobiti novi sortirani podnizovi.

Međutim, prije faze spajanja sortiranih podnizova, koja zapravo predstavlja drugu fazu algoritma Merge sort, u prvoj fazi se dijeli početni nesortirani niz na sve manje i manje dijelove, sve dok se ne dobiju nizovi sa samo jednim elementom.

Nakon te prve faze dijeljenja, slijedi druga faza spajanja sortiranih podnizova u kojoj se njihovim objedinjavanjem stvaraju sve veći i veći sortirani podnizovi, sve dok se na kraju ove faze ne kreira cijeli sortirani niz.

Algoritam Merge sort se izvodi rekurzijom:

dijeljenje početnog nesortiranog niza na dva podniza, od kojih svaki ima po n/2 elemenata;

rekurzivno sortiranje dobijena dva podniza koristeći Merge sort;

spajanje sortiranih podnizova.

Slika prikazuje proceduru MERGE-SORT koja predstavlja rekurzivnu implementaciju algoritma. Ulazni argument A predstavlja niz koji se treba sortirati.

Argument l označava indeks prve pozicije, dok argument u označava indeks zadnje pozicije podniza

A[l..u] koji se sortira.

**MERGE-SORT (A, l, u)**

**If (u > l) then**

**p🡨 [ (l+u-1) / 2 ]**

**q 🡨 p+1**

**MERGE-SORT (A,l.p)**

**MERGE-SORT (A,q,u)**

**MERGE (A,l,p,q,u)**

**end\_if**

Postupak sortiranja se ostvaruje tako što se ulazni podniz A[l..u] dijeli na dva podniza, a zatim se algoritamski proces nastavlja rekurzivnim pozivima procedure MERGE-SORT, koji kao ulazne nizove imaju podnizove nastale dijeljenjem.

Dijeljenje ulaznog niza na dva podniza se ostvaruje izračunavanjem vrijednosti varijabli p i q (linije 2- 3).

Varijabla p predstavlja indeks zadnje pozicije prvog podniza, a vrijednost varijable q označava prvu

poziciju drugog podniza.

Na taj način ulazni niz A[l..u] je podijeljen na podnizove A[l..p] i A[q..u]. U proceduri MERGE-SORT se nalaze dva rekurzivna poziva ove procedure.

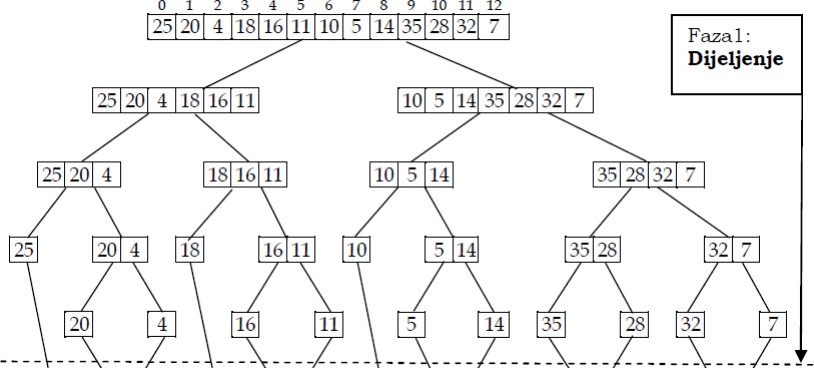
Prvim pozivom MERGE-SORT(A, l, p) se rekurzivni proces nastavlja nad prvom polovinom podijeljenog niza (linija 4), dok se pozivom MERGE-SORT (A, q, u) rekurzivni proces nastavlja nad drugom polovinom podijeljenog niza (linija 5).

Nesortirani niz se rekurzivno dijeli na podnizove, sve dok podnizovi ne budu sadržavali samo jedan

element, tj. dok vrijedi u > l (linija 1).

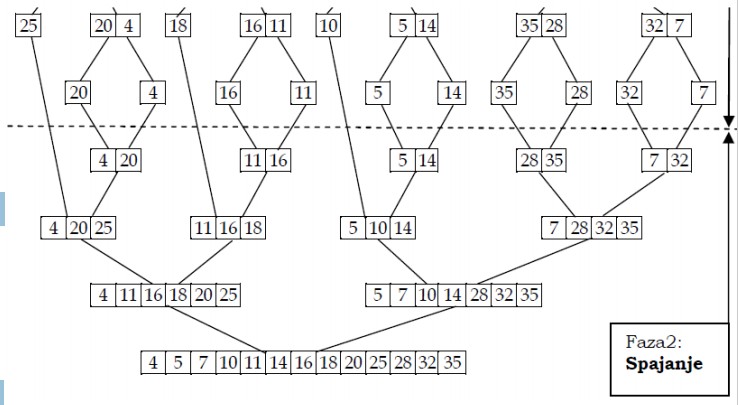
MERGE procedura spaja nizove

Ključni koraci u dijeljenju nakon kojih ide MERGE procedura



Ključni koraci u algoritmu Merge sort su koraci spajanja dva sortirana podniza.

Razmotrimo, na primjer, nizove B[0..5] i B[6..12] koji su sortirani u rastućem poretku.

Da bismo spojili ta dva sortirana podniza u jedan sortirani niz A, prvo je potrebno odrediti najmanji element iz oba podniza.

Sigurno znamo da je najmanji element ili B[0] ili B[6].

Isto tako, znamo da je najveći element ili B[5] ili B[12].

Operaciju spajanja počinjemo tako da u A[0] smještamo manji od elemenata: B[0] ili B[6]. Pretpostavimo da je B[0]

Trenutno su neraspoređeni elementi B[1..5] i B[6..12].

U A[1] trebamo smjestiti manji od elemenata na prvim pozicijama neraspoređenih dijelova

podnizova B[1..5] i B[6..12].

Dakle, u A[1] trebamo smjestiti manji od elemenata: B[1] ili B[6].

Pretpostavimo da je manji element B[1], pa je A[1]=B[1].

Trenutno su neraspoređeni elementi B[2..5] i elementi B[6..12].

Možemo zaključiti da će nam, za formiranje sortiranog niza A, trebati mehanizam koji će nam

omogućiti označavanje raspoređenih i neraspoređenih elemenata iz podnizova B[0..5] i B[6..12].

Ovaj mehanizam praćenja ćemo ostvariti uvođenjem dva cjelobrojna indeksa za spomenute podnizove, koji će označavati poziciju u oba podniza na kojoj se trenutno nalaze kandidati za minimalni element od preostalih neraspoređenih elemenata.

Uvedimo indeks i za praćenje podniza B[0..5], a indeks j za praćenje podniza B[6..12].

Svakim određivanjem minimalnog elementa se taj element smješta u niz A, a indeks pridružen podnizu koji je u tom koraku sadržavao minimalni element se povećava za 1.

Dakle, preostali neraspoređeni elementi u podnizu B[0..5] će biti elementi B[i..5], a u podnizu B[6..12] će biti neraspoređeni elementi B[j..12].

U jednom trenutku će se rasporediti svi elementi iz jednog od podnizova, pa će se preostali elementi iz podniza u kojem je ostalo još neraspoređenih elemenata jednostavno nadovezati na spojeni niz

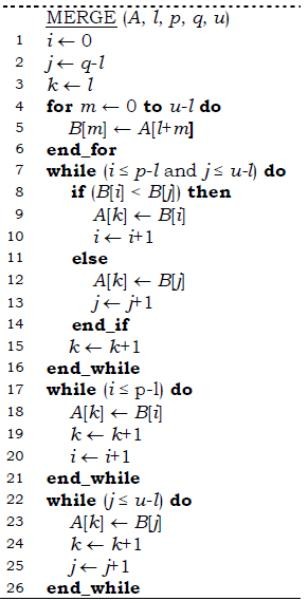
Ulazni argument A označava niz u kojem su smješteni podnizovi.

Argumenti l i q označavaju indekse početnih pozicija prvog i drugog podniza, respektivno. Argumenti p i u označavaju indekse zadnjih pozicija prvog i drugog podniza, respektivno.

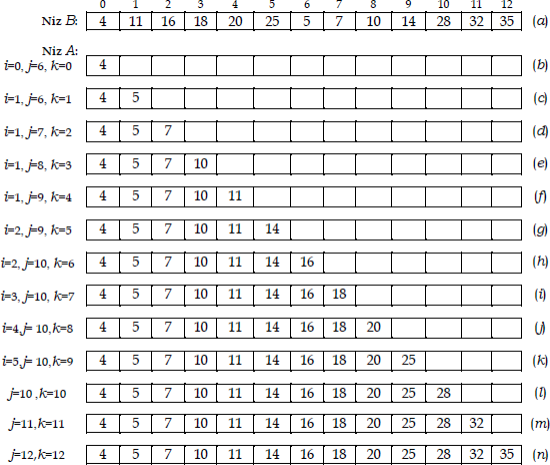
Varijabla i u proceduri MERGE je varijabla pridružena prvom podnizu i označava indeks pozicije iznad koje su još neraspoređeni elementi iz prvog podniza.

Varijabla j je pridružena drugom podnizu i označava indeks pozicije iznad koje su još neraspoređeni

elementi iz drugog podniza.

U proceduri MERGE se također koristi i privremeni niz B, u kojem se privremeno smještaju sortirani podnizovi, koji se trebaju spojiti. Varijabla k označava indeks pozicije u rezultirajućem nizu A, na koju će se smjestiti trenutno određeni minimalni element iz sortiranih podnizova

•



Varijabla i=0 na početku procesa predstavlja indeks pozicije od koje počinje prvi podniz, dok varijabla j=6 označava indeks pozicije od koje počinje drugi podniz.

Varijabla k označava poziciju u rezultirajućem nizu A, na koju se u nekom od pojedinih koraka spajanja treba smjestiti manji od elemenata: B[i] ili B[j].

Ako je manji element B[i], onda se u rezultirajući niz A na poziciju k smješta taj element, te se

inkrementira indeks i.

Tada podniz B[i..5] predstavlja trenutno još neraspoređene elemente iz prvog podniza. S druge strane, ako se utvrdi da je manji element B[j], onda se u rezultirajući niz A na poziciju k smješta taj element, pa se inkrementira indeks j.

Tada podniz B[j..12] predstavlja trenutno još neraspoređene elemente iz drugog podniza.

Nakon svakog smještanja elementa u rezultirajući niz A, inkrementira se varijabla k.

Na primjer, inicijalno varijable i i j imaju vrijednosti i=0, j=6, k=0 (slika b).

Poređenjem elemenata B[0] i B[6] se utvrđuje da je manji element B[0]=4, pa se taj element u nizu A

smješta na poziciju k=0, A[0]=4.

Pošto je u ovom koraku izabran element iz prve polovine, inkrementira se varijabla i, kao i varijabla k, koja se inkrementira u svakom pojedinom koraku smještanja elemenata u rezultirajući niz A.

U ovom trenutku vrijednosti varijabli su: i=1, j=6, k=1.

Dakle, u sljedećem koraku se izabire manji od elemenata: B[1] ili B[6], te se utvrđuje da je manji element B[6]=5, pa se taj element u nizu A smješta na poziciju k=1, A[1]=5 (slika c).

Na slici su ilustrirani i svi ostali koraci spajanja podnizova.

Primijetite da su u koraku koji je ilustriran slikom k, svi elementi iz prvog podniza (B[0..5]) već raspoređeni u rezultirajući niz A, pa je u nastavku algoritamskog procesa potrebno samo da se

preostali elementi (B[10..12]) iz drugog podniza (B[6..12]) jednostavno dodaju u rezultirajući niz A

(slike l-n).

## Quick sort

Algoritam Quick sort također koristi strategiju “podijeli, pa vladaj”.

Međutim, kod algoritma Quick sort se ulazni niz dijeli na dva dijela, tako što se taj niz reorganizira na način da su svi elementi u prvom donjem dijelu manji od svih elemenata u drugom gornjem dijelu niza.

Ta dva dijela nakon reorganizacije su nesortirana i razdvojena su elementom koji se naziva pivot i koji se nalazi na nekoj poziciji j.

Dakle, donji dio čine elementi koji imaju vrijednost manju ili jednaku vrijednosti pivota (A[i] ≤ pivot, i=0, 1,…, j-1), dok gornji dio čine elementi koji imaju vrijednost veću ili jednaku vrijednosti pivota (A[i] ≥ pivot, i=j+1, j+2,…, n-1). Mogući su različiti načini izbora pivota.

Na primjer, izbor pivota možemo napraviti tako da uzmemo prvi element niza koji se dijeli, mada postoje i efikasniji načini za njegov izbor.

Idealno, pivot element bi trebao dijeliti niz na dva podniza sa jednakim brojem elemenata.

Zbog načina na koji je ulazni niz reorganiziran, možemo zaključiti da je pivot element pozicioniran na svoju konačnu poziciju j, te se u daljnjem postupku više neće pomjerati.

Nadalje, možemo zaključiti da se sortiranje može nastaviti tako da isti algoritam rekurzivno

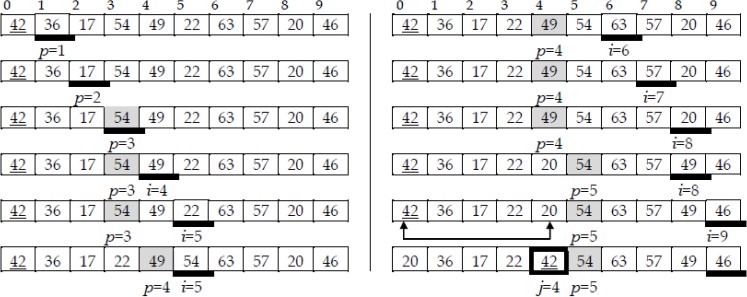
primijenimo na dobivena dva dijela ulaznog niza.

Rekurzivni postupak se nastavlja, sve dok rezultirajući dijelovi ne budu imali samo jedan element. Završetkom rekurzivnog postupka se dobiva sortirani niz

Funkcija PARTICIJA vraća indeks pozicije na kojoj je smješten pivot.

U primjeru na slici za pivot se uzima element 42 (element je označen tankom donjom crtom).

Funkcijom PARTICIJA je potrebno reorganizirati niz, tako da se pronađe odgovarajuća pozicija za pivota, te da svi elementi manji od pivota budu na pozicijama ispred njega, a svi veći elementi da budu na pozicijama iza pivota.



Na slici oznaka p označava poziciju elemenata koji se trenutno uspoređuju s pivotom, do trenutka

dok se ne pronađe prvi element koji je veći od pivota.

Kada se pronađe prvi element veći od pivota, za označavanje pozicija elemenata koji se trenutno uspoređuju sa pivotom koristi se index i.

Te pozicije su na slici istaknute donjom podebljanom crtom.

Počevši od drugog elementa u nizu, započinjemo pronalaženje prvog elementa koji je veći ili jednak

pivotu (42), te na poziciji s indeksom 3 pronalazimo prvi takav element, a to je element 54.

Pronađenu poziciju p=3 označavamo kao privremeni početak druge particije (particije koja će se nalaziti iza pivota), što je na slici označeno osjenčenom pozicijom.

Postupak nastavljamo usporedbom pivota i elemenata iz niza (i=p+1, p+2,…, n-1) koji se nalaze iza

označene pozicije

Kada se pronađe element koji je veći ili jednak pivotu, taj element se ostavlja na svojoj poziciji, te se

nastavlja proces usporedbi pivota i elemenata na preostalim pozicijama.

Na primjer, na slici na poziciji i=4 je element 49, pa taj element ostaje na svojoj poziciji.

Kada se pronađe element koji je manji od pivota, tada taj element i element na poziciji p razmjenjuju pozicije, te se oznaka p povećava za 1, tako da oznaka p pokazuje na novi početak druge particije.

Na primjer, na slici na poziciji s indeksom i=5 se nalazi element 22, koji je manji od pivota (42), pa element 22 razmjenjuje poziciju s elementom na poziciji p, a to je 54.

Istovremeno, privremena vrijednost indeksa p za početak druge particije se povećava za 1, pa sada

ima vrijednost p=4.

Ovaj proces se nastavlja, sve dok se ne dođe do kraja niza.

Pri kraju procesa, kada indeks i dostigne vrijednost 9 (i=n-1), pivot se još uvijek nalazi na prvoj

poziciji.

Međutim, niz smo prethodno reorganizirali tako da su svi elementi od pozicije 1 do pozicije 4 (p-1) manji od pivota, dok su elementi na pozicijama 5 do 9 (p do n-1) veći od pivota.

Pošto nam je cilj da pivot dijeli te dvije grupe elemenata, potrebno je još razmjeniti pozicije pivota i

elementa na poziciji 4 (p-1), čime se završava particioniranje niza.

Dakle, na kraju opisanog procesa particioniranja niza, pivot element 42 se nalazi na poziciji 4. Svi elementi na pozicijama s indeksima 0 do 3 su manji od pivot elementa 42, dok su svi elementi na pozicijama 5 do 9 veći od pivot elementa

Istaknimo još da zamjena pozicija pivot elementa i elementa na poziciji p-1 mijenja poredak elemenata u donjoj particiji. Na primjer, na slici je nakon izmjene poredak elemenata koji pripadaju donjoj particiji (36, 17, 22, 20) promijenjen u sljedeći poredak: (20, 36, 17, 22).

Procedura particija

**PARTICIJA (A, prvi, zadnji)**

**pivot 🡨 A[prvi]**

**p 🡨 prvi+1**

**while (p <= zadnji and A[p] < pivot) do**

**p 🡨 p+1;**

**end\_while**

**for i 🡨 p+1 to zadnji do**

**if (A[i] <pivot) then**

**A [i] <--> A[p]**

**p 🡨 p+1**

**end\_if**

**end\_for**

**A[prvi] <--> A[p-1]**

**return p-1**

Algoritam je opisan procedurom QUICK-SORT.

Pošto na početku čitav niz predstavlja jednu particiju, algoritam se poziva sa početnim argumentima

prvi=0 i zadnji=n-1

**QUICK-SORT (A, prvi, zadnji)**

**If (prvi < zadnji) then**

**j 🡨 PARTICIJA (A, prvi, zadnji)**

**QUICK-SORT (A, prvi, j-1(**

**QUCIK-SORT (A, j+1, zadnji)**

**end\_if**

Inicijalni raspored elemenata niza je prikazan prvim rasporedom u gornjem lijevom kutu na slajdu 65

, dok je na slici a prikazan raspored elemenata u nizu nakon prve particije.

U pojedinim fazama izabrani pivot je označen kao podvučeni element, a elementi koji su zauzeli

svoje konačne pozicije su uokvireni podebljanom linijom.

Isprekidanim linijama su uokvireni elementi koji u pojedinim fazama izvršavanja algoritma čine

podnizove koji u nastavku algoritma ulaze u proces particije.



U sljedećem koraku (slika b) kao pivot se uzima element 20, a proces particije se nastavlja sa

podnizom koji čine elementi koji se nalaze na pozicijama 0 do 3.

Rezultat particije je prikazan na slici c.

Pivot element 20 je zauzeo svoje konačno mjesto na poziciji s indeksom 1, donju particiju čini samo jedan element (17), dok gornju particiju čine dva elementa (36 i 22).

Na slikama d do h su ilustrirane i ostale faze u radu algoritma.

## Heap sort

Gomila nije korisna samo za efikasnu implementaciju prioritetnog reda, nego se koristeći gomilu kao strukturu podataka može realizirati i vrlo efikasan algoritam sortiranja koji se naziva sortiranje gomilom (eng. Heap sort).

Algoritam Heap sort je karakterističan upravo po tome što se za vrijeme izvođenja sortiranja za manipulaciju podacima koristi jedna specifična struktura podataka

Algoritam počinje korištenjem procedure STVORI-GOMILU pomoću koje se stvara gomila od ulaznog

niza A[0..n-1], gdje n označava veličinu niza n=duzina[A].

Na taj način je najveći element smješten u korijenu A[0].

Konačna pozicija najvećeg elementa je zadnja pozicija n-1, što možemo jednostavno postići

zamjenom pozicija elemenata A[0] i A[n 1].

Nakon te zamjene, u nizu A se na zadnjoj poziciji n nalazi najveći element, dok se u ostatku niza A[0..n-2] nalaze elementi od kojih se lako može stvoriti gomila koja ima jedan element manje u odnosu na prethodno stanje.

Naime, nakon zamjene pozicija, djeca korijena ostaju gomile, a novi element u korijenu može narušiti svojstvo poretka.

Međutim, da bi se obnovila za jedan element manja gomila, jednostavno možemo pozvati

proceduru POPRAVI-DOLJE (A, 0), koja će kao rezultat stvoriti gomilu A[0..n-2].

Prema tome, u tom trenutku u nizu A imamo dva dijela.

Prvi nesortirani dio je gomila A[0..n-2] sa n-1 elemenata. Drugi dio A[n-1] je sortirani dio, koji u ovom trenutku ima samo jedan element.

Lako zaključujemo da cjelokupan sortirani niz A možemo dobiti ponavljanjem prethodno opisanog procesa, na način da u svakoj sljedećoj iteraciji za jedan element povećavamo sortirani dio, te

također za jedan element smanjujemo veličinu gomile.

Iteracije ponavljamo sve dok veličina gomile ne dostigne vrijednost 1.

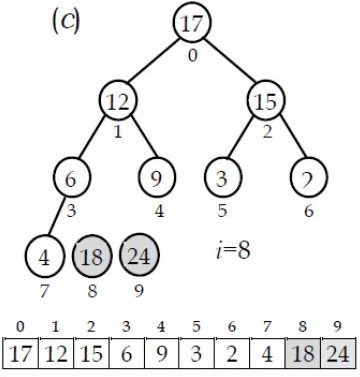
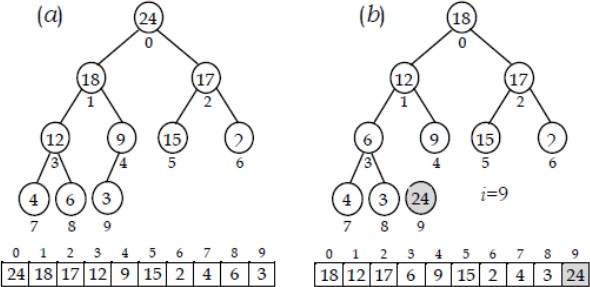
Na slici a je prikazano stanje gomile nakon poziva procedure STVORI-GOMILU

Nadalje, na slici b je prikazano stanje gomile nakon pozicioniranja najvećeg elementa na zadnju

poziciju s indeksom 9, te obnavljanja gomile A[0..8], pozivom procedure POPRAVI-DOLJE (A, 0).

U ovom trenutku element A[9] je izbačen iz gomile, a u korijen gomile, koja ima veličinu 9, je smješten element 18.

Trenutno nesortirani dio niza čine elementi A[0..8]={18, 12, 17, 6, 9, 15, 2, 4, 3}. Sortirani dio niza sadrži samo element A[9]=24.



Elementi koji su izbačeni iz gomile su elementi koji se nalaze na svojim konačnim pozicijama i na slici su istaknuti sjenčenjem.

U sljedećem koraku vrijednost varijable i je 8, pa se na tu poziciju treba postaviti maksimalni

element iz nesortiranog dijela, a to je element A[0]=18, koji se nalazi u korijenu gomile.

Pozivom funkcije POPRAVI-DOLJE (A, 0) se obnavlja gomila A[0..7], tako da se u korijen A[0] pozicionira najveći element 17 iz nesortiranog dijela niza. Stanje gomile A u ovom trenutku je prikazano na slici c.

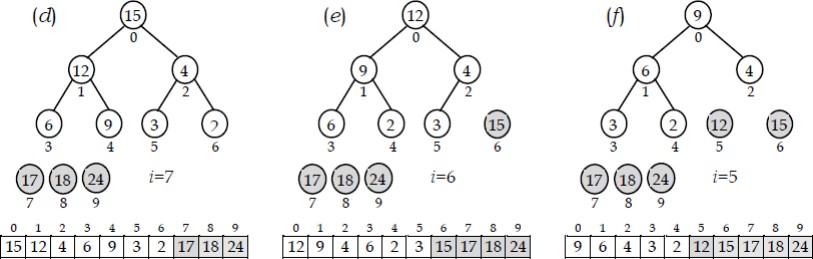
Gomilu veličine 8 koja predstavlja trenutno nesortirani dio niza čine elementi A[0..7]={17, 12, 15, 6,

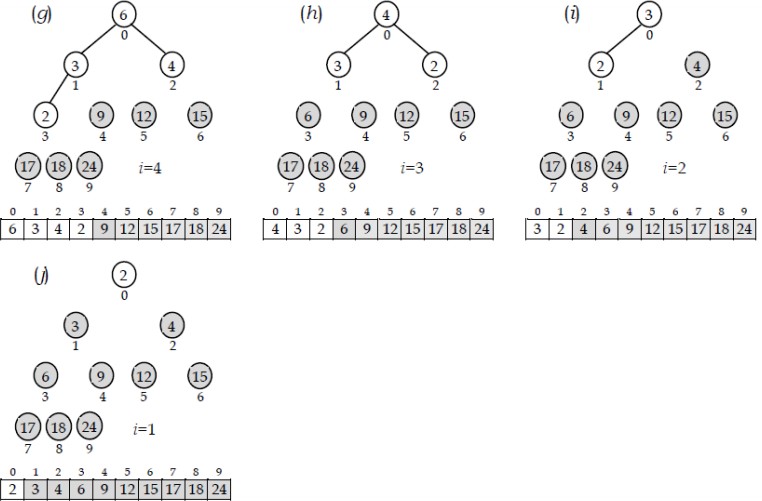
9, 3, 2, 4}.

Sortirani dio niza čine dva elementa A[8..9]={18, 24}.

Stanja gomile A u svim ostalim koracima do završetka algoritamskog procesa, nakon poziva funkcije

POPRAVI-DOLJE (A, 0), su prikazana na slikama d-f.





Algoritam sortiranja koji koristi gomilu kao strukturu podataka na prethodno opisani način je prikazan procedurom GOMILA-SORT.

Ulaz u proceduru je niz A, koji se sortira. Nakon početnog stvaranja gomile pozivom procedure STVORI-GOMILU (linija 1), u for petlji se izvode koraci kojima se izmjenjuju pozicije najvećeg elementa A[0] iz korijena i elementa A[i] na zadnjoj poziciji nesortiranog dijela niza (linija 3).

U svakoj iteraciji se, smanjivanjem gomile za 1, element A[i] izbacuje iz gomile, jer je zauzeo svoju

konačnu poziciju (linija 4).

Budući da se prethodno izvedenom izmjenom pozicija obično narušava svojstvo poretka, potrebno

je to svojstvo obnoviti pozivom procedure POPRAVIDOLJE (A, 0) (linija 5).

**GOMILA-SORT (A)**

**STVORI-GOMILU (A)**

**for i** 🡨 **duzina[A] -1 downto 1 do**

**A[0] <--> A[i]**

**velicina[A]** 🡨 **velicina [A] -1**

**POPRAVI-DOLJE (A, 0)**

**end\_for**

U tabeli je prikazana modificirana verzija sortiranja gomilom, u kojoj se koristi procedura IZBACI- PRVI.

Naime, nakon stvaranja gomile iz neuređenog niza (linija 1), za sortiranje niza A[0..duzina[A]-1] je dovoljno duzina[A]-1 puta izbaciti prvi element iz gomile, jer su izbačeni elementi samo u logičkom smislu uklonjeni iz gomile, dok u fizičkom smislu ti elementi formiraju sortiran niz A[0..duzina[A]-1] (linije 2-4).

**GOMILA-SORT-M (A)**

**1 STVORI-GOMILU (A)**

**2 for i🡨0 to duzina[A] – 2 do**

**3 IZBACI-PRVI (A)**

**4 end\_for**

## Sortiranje u linearnom vremenu

Sortiranje brojanjem

Radix sortiranje

Adresno sortiranje

## Sortiranje brojanjem

Algoritam sortiranja brojanjem (eng. Counting sort) koristi pretpostavku da su elementi A[i] iz ulaznog niza cjelobrojne vrijednosti iz raspona 0 do k, 0 <=A[i] <= k.

Osnovni mehanizam rada algoritma se sastoji u tome da se za svaki ulazni element x odredi broj elemenata iz niza koji su manji od x.

Ta informacija se zatim koristi da se element x direktno pozicionira na svoje odgovarajuće mjesto u

sortiranom nizu.

Na primjer, ako postoji 13 elemenata u nizu koji su manji od elementa x, onda se element x treba u sortiranom nizu postaviti na poziciju s indeksom 13.

U realizaciji algoritma Counting sort se pored ulaznog niza A[0..n 1], koriste i dva dodatna niza:

B[0..n-1], koji će sadržavati sortirani niz.

C[0..k], koji predstavlja privremeni radni prostor u kome se pohranjuju brojači vrijednosti.

Procedura

Prvo se u algoritmu brojači u nizu C inicijaliziraju na 0 (linije 1-3). Sadržaj niza A i inicijalni sadržaj niza

C su prikazani na slici a.

Zatim se prolazi kroz ulazni niz A, pri čemu se za svaki element A[j] inkrementira odgovarajući brojač, C[A[j]]=C[A[j]]+1 (linije 4-6). Na taj način, nakon druge for petlje, u nizu C se na poziciji s indeksom i nalazi broj koji je jednak broju elemenata u ulaznom nizu sa vrijednošću i.

Na slici b je prikazan sadržaj niza C, nakon završetka druge for petlje. Na primjer, element C[2] ima vrijednost 3, jer se u ulaznom nizu A nalaze 3 elementa sa vrijednošću 2

**COUNTING-SORT (A, B, k)**

**for i🡨 0 to k do**

**C[i] 🡨0**

**end\_for**

**for j🡨 0 to n-1 do**

**C[A[j]] 🡨 C[A[j]] +1**

**end\_for**

**for i 🡨1 to k do**

**C [i] 🡨 C[i] + C[i-1]**

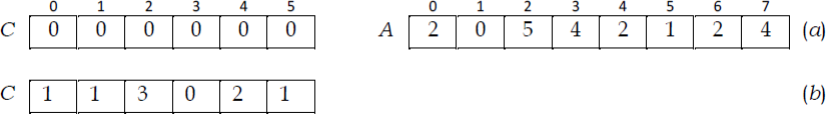
**end\_for**

**for j🡨 n-1 downto 0 do**

**C[A[k]] 🡨 C[A[j]] -1**

**B[C[A [ j ]]] 🡨 A[j]**

**end\_for**



U trećoj for petlji se povećava vrijednost brojača C[i] sa vrijednošću brojača C[i-1], da bi se dobio broj elemenata koji su manji ili jednaki vrijednosti i (linije 7-9).

Na slici c je prikazan sadržaj niza C, nakon završetka treće for petlje. Na primjer, element C[2] ima vrijednost 5, što pokazuje da u ulaznom nizu ima ukupno 5 elemenata koji su manji ili jednaki vrijednosti 2.

Nakon izvođenja treće for petlje je završena faza algoritma u kojoj je pripremljen niz C, te se u zadnjoj for petlji može oblikovati sortirani niz B. U zadnjoj (četvrtoj) for petlji se svaki element A[j] stavlja na odgovarajuću poziciju u nizu B (linije 10-13).



Oblikovanje izlaznog sortiranog niza započinje od zadnjeg elementa A[n-1] (linija 10).

U primjeru na slici zadnji element je broj 4. Budući da je C[4]=7, to znači da ima ukupno 7 elemenata koji su manji ili jednaki elementu 4. Drugim riječima, element 4 se treba pozicionirati na sedmu poziciju u nizu B, a to je pozicija sa indeksom 6 (slika d).

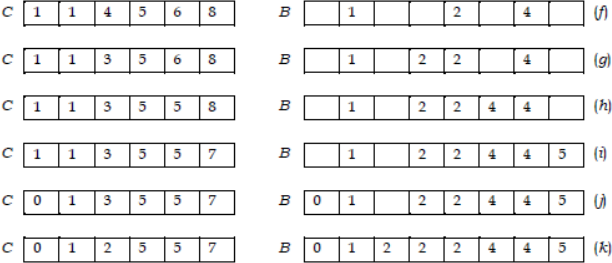


Drugi element koji se smješta na odgovarajuću poziciju je element 2. Pošto je C[2]=5, to znači da ima ukupno 5 elemenata koji su manji ili jednaki elementu 2.

Prema tome, element 2 se treba smjestiti na poziciju s indeksom 4 (slika e).



### Ostali koraci



Primijetimo da se u zadnjoj for petlji pored generiranja niza B također ažurira i niz C, tako da se odgovarajuća vrijednost C[A[j]] prvo dekrementira, da bi se element A[j] smjestio na odgovarajuće mjesto, a to je pozicija s indeksom C[A[j]] (linija 11-12).

Osim toga, važno je istaknuti da se brojač C[A[j]] dekrementira i zbog još jednog razloga.

Naime, ako u ulaznom nizu postoji još jedan element sa istom vrijednošću A[i], onda se taj element

treba pozicionirati na mjesto neposredno ispred prethodno pozicioniranog elementa sa istom

vrijednošću.

## Definicije i terminologija

**(Predavanje 15) Grafovi**

Potrebna je struktura pomoću koje se mogu modelirati relacije između dva objekta iz nekog skupa

objekata.

Struktura kojom se može modelirati skup, za dati problem relevantnih objekata i skup za dati

problem relevantnih relacija između dva objekta, se naziva graf.

Primjeri područja u kojima se primjenjuju grafovi su:

modeliranje povezanosti u računarskim i komunikacijskim mrežama;

modeliranje protoka u transportnim mrežama;

modeliranje organizacijskih situacija u kojima je potrebno na prihvatljiv način kreirati redoslijed

podzadataka u okviru kompleksnih aktivnosti;

modeliranje odnosa između objekata u različitim znanstvenim, poslovnim i inženjerskim

primjenama.

Pri primjeni grafova u različitim područjima, objekti koji su u određenim odnosima se prikazuju čvorovima, dok se odnosi između pojedinih objekata prikazuju granama.

Formalno, graf G je par:

G=(V, E)

gdje je:

V konačan neprazan skup čvorova;

E skup grana koje povezuju čvorove iz V.

## Usmjereni graf

Ako je E skup uređenih parova (u, v), onda se graf G naziva usmjereni graf ili digraf (eng. directed

graph).

Primjer usmjerenog grafa je prikazan na slici a.

Za usmjereni graf na slici a imamo:

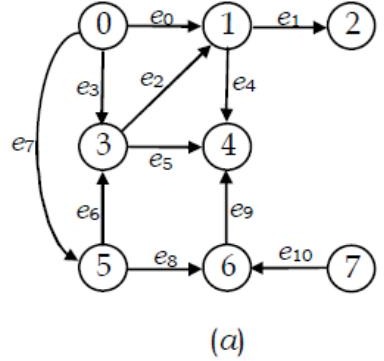
V= {0, 1, 2, 3, 4, 5, 6, 7};

E={e0, e1, e2, e3, e4, e5, e6, e7, e8, e9, e10},

gdje je:

e0= (0, 1), e1=(1, 2), e2=(3, 1), e3=(0, 3), e4=(1, 4), e5=(3, 4), e6=(5, 3), e7=(0, 5), e8=(5, 6), e9=(6, 4),

e10=(7, 6).



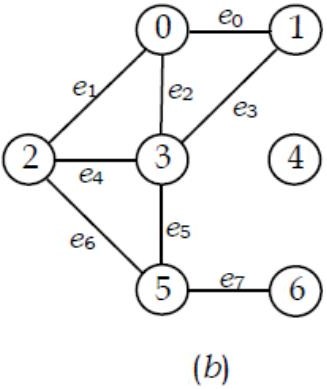
## Neusmjereni graf

S druge strane, ako je E skup neuređenih parova, onda se graf naziva neusmjereni graf.

Primjer neusmjerenog grafa je prikazan na slici b. Za graf na slici b imamo

V= {0, 1, 2, 3, 4, 5, 6};

E={e0, e1, e2, e3, e4, e5, e6, e7}, gdje je:

e0={0, 1}, e1={0, 2}, e2={0, 3}, e3={1, 3}, e4={2, 3}, e5={3, 5}, e6={2, 5}, e7={5, 6}.

Za čvorove u i v grafa G kažemo da su incidentni s granom e=(u, v), koja ih spaja.

Isto tako, za granu e kažemo da je incidentna čvorovima u i v.

Za graf sa slike b, čvorovi 5 i 6 su incidentni s granom e7, dok su čvorovi 2 i 3 incidentni s granom e4.

Također, grana e7 je incidentna čvorovima 5 i 6, a grana e4 je incidentna čvorovima 2 i 3. Budući da su kod usmjerenog grafa grane orijentirane, onda incidentne grane u odnosu na neki čvor mogu biti ulazne i izlazne.

Dva čvora u i v u neusmjerenom grafu, incidentna s nekom granom e={u, v}, se nazivaju susjedni

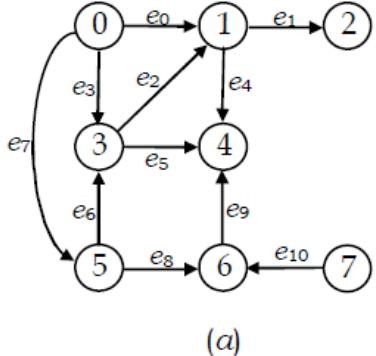
čvorovi

S druge strane, ako su čvorovi u i v incidentni nekoj grani e=(u, v) u usmjerenom grafu, onda je samo čvor v susjedan čvoru u, jer relacija susjedstva kod usmjerenih grafova nije simetrična.

Broj incidentnih grana čvora v∈V se naziva stupanj čvora.

Na primjer, stupanj čvora 5 na slici b je 3, jer je taj čvor incidentan sa 3 grane: e5, e6 i e7.

## Usmjereni graf

Kod usmjerenog grafa se dodatno mogu definirati ulazni stupanj čvora, kao broj grana koje završavaju u tom čvoru, te izlazni stupanj, kao broj grana koje izlaze iz tog čvora.

Na primjer, za čvor 5 na slici a, ulazni stupanj je 1, dok je izlazni

stupanj 2.

Ukupni stupanj je jednak zbiru ulaznog i izlaznog stupnja.

Ukoliko postoji grana koja spaja neki čvor sa samim sobom,

tada se takva grana naziva petlja.

Pošto je graf između ostalog definiran i skupom grana E, onda je prirodno da se ne dozvoljava da postoji više grana koje spajaju dva ista čvora.

Međutim, u određenim primjenama, ipak ima smisla dozvoliti da postoji više grana između dva ista čvora, a te grane se tada nazivaju paralelne grane.

Graf koji sadrži paralelne grane se naziva multigraf

## Neusmjereni graf

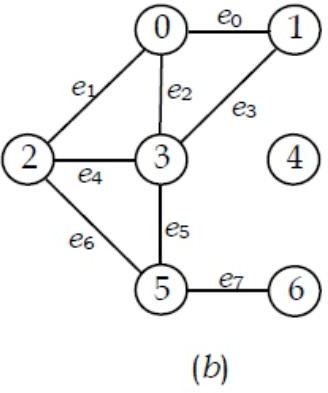
Put između čvorova u i v, u nekom grafu G=(V, E), definiramo kao niz čvorova (v0, v1, v2, ..., vk), gdje

je u=v0, v=vk, (vi-1, vi)∈E, 1 ≤ i ≤ k.

Broj k se naziva dužina puta.

Put je prost, ako su svi čvorovi na putu različiti.

Prema tome, svaka dva susjedna čvora vi-1 i vi na putu predstavljaju granu ei=(vi-1, vi) ∈ E, 1 ≤ i ≤ k.



Na primjer, u grafu koji je prikazan na slici b, postoji put (2, 3, 5, 6), od čvora 2 do čvora 6.

Ako postoji put od čvora u do čvora v, onda se kaže da je čvor v dostižan iz čvora u.

Za neki put kažemo da je zatvoren, ako ima dužinu veću od nule, a početni i krajnji čvor se

podudaraju.

Primjer zatvorenog puta u grafu prikazanom na slici 12.1b je (2, 3, 1, 0, 3, 5, 2).

Zatvoreni put kod kojeg su prvi čvor i unutrašnji čvorovi različiti se naziva ciklus. Primjer ciklusa u

grafu prikazanom na slici b je (3, 1, 0, 2, 3).

Ako graf sadrži barem jedan ciklus, kao što je to slučaj sa grafom na slici b, onda je graf cikličan.

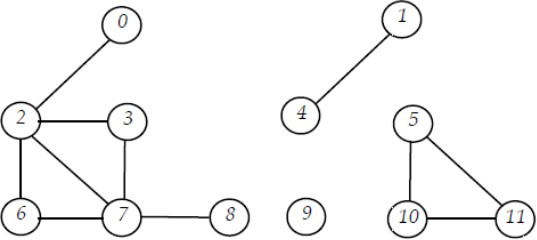
U suprotnom, ako graf ne sadrži niti jedan ciklus, onda se takav graf naziva acikličan graf ili šuma.

Neusmjereni graf je povezan, ako između svaka dva čvora postoji put.

Ako to nije ispunjeno, onda se graf sastoji od više komponenti povezanosti.

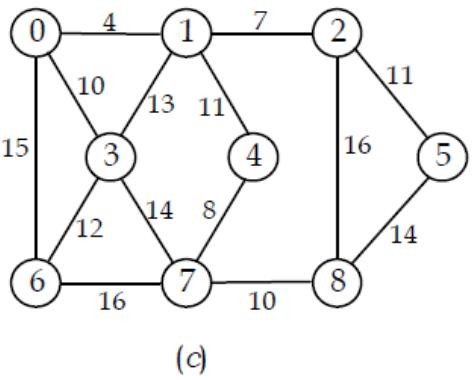
Na slici je prikazan primjer nepovezanog grafa koji se sastoji od četiri komponente povezanosti. Čvorovi 0, 2, 3, 6, 7 i 8, čine prvu komponentu povezanosti, čvorovi 1 i 4 čine drugu komponentu povezanosti, dok čvorovi 5, 10 i 11, čine treću komponentu povezanosti.

Čvor 9, sam po sebi, čini četvrtu komponentu povezanosti.



Ako granama grafa pridružimo težine w(u, v), onda se graf naziva težinskim.

Na slici c je prikazan primjer težinskog grafa



Važna vrsta grafova su tzv. slobodna stabla, kod kojih ne postoji niti jedan istaknuti čvor koji ima

svojstva korijena.

Slobodno stablo je povezan neusmjeren graf, koji ne sadrži cikluse (acikličan graf).

Ekvivalentna definicija bi mogla biti da je slobodno stablo povezan neusmjeren graf, koji ima |V|-1 grana. Neka od bitnih svojstava slobodnog stabla su:

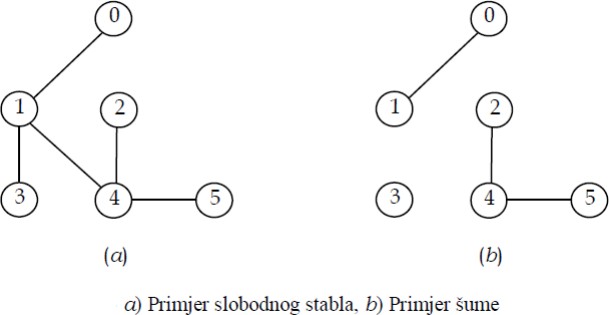
dodavanjem bilo koje grane se pojavljuje ciklus u grafu;

uklanjanjem bilo koje grane, graf postaje nepovezan;

između dva čvora ne postoji više grana;

između bilo koja dva čvora postoji jedinstven put

Slika prikazuje primjer jednog slobodnog stabla (slika a) i primjer šume koja sadrži tri stabla (slika b).



## Prikaz grafova

Postoje različiti načini za prikaz grafova.

Za prikaz i implementaciju grafova u nekom programskom jeziku se najčešće koriste: • matrice susjedstva;

liste susjedstva.

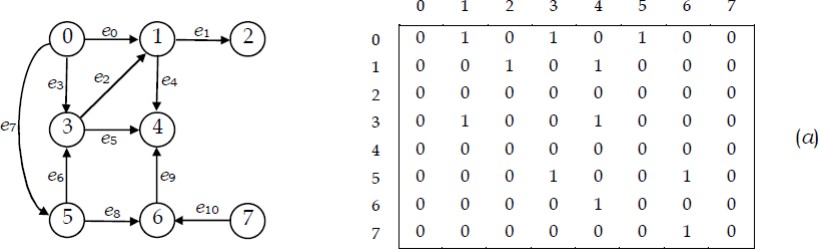
## Matrice susjedstva

Matrica susjedstva za graf G=(V, E) je matrica |V|x|V|, kod koje su elementi aij definirani na sljedeći način:

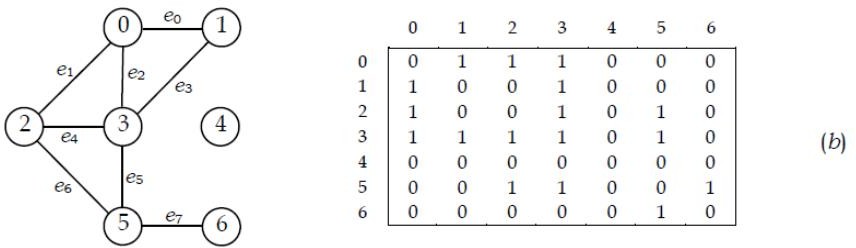
aij=1 ako grana (vi,vj) ∈ E;

aij=0 ako grana (vi, vj) ∈ E.

Na slici a je prikazan primjer matrice susjedstva za usmjereni graf



Na slici b je prikazan primjer matrice susjedstva za neusmjereni graf



Redoslijed čvorova v0, v1,...v|V|, je proizvoljan, pa je moguće za isti graf kreirati n! matrica

susjedstva.

Broj jedinica u redu, koji pripada nekom čvoru vi, je jednak izlaznom stupnju tog čvora. Na primjer, izlazni stupanj čvora 3 je 2, pa se u redu, koji pripada tom čvoru, nalaze dvije jedinice.

Broj jedinica u koloni, koja odgovara nekom čvoru vi, je jednak ulaznom stupnju tog čvora.

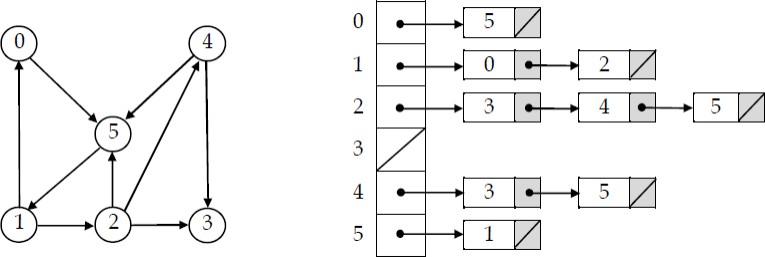
Na primjer, ulazni stupanj čvora 4 je 3, pa se u koloni koja pripada tom čvoru nalaze tri jedinice.

Budući da je kod neusmjerenog grafa grana e={u, v}, koja spaja čvorove u i v, zapravo isto što i grana e={v, u}, koja spaja čvorove v i u, onda je matrica susjedstva A jednaka transponiranoj matrici, A=A T

To znači da je matrica susjedstva simetrična u odnosu na glavnu dijagonalu. S druge strane, u općem slučaju, matrica susjedstva kod usmjerenih grafova nije simetrična.

## Liste susjedstva

Lista susjedstva je niz povezanih listi. Niz ima dužinu |V|, pri čemu se na poziciji i nalazi pokazivač na povezanu listu grana koje su incidentne čvoru vi. Odnosno, povezana lista, kojoj je pokazivač na čelo smješten u nizu na poziciji i, sadrži čvorove koji su susjedi čvora vi. Primjer liste susjedstva za usmjereni graf je prikazan na slici



Na primjer lista, koji odgovara čvoru 1, sadrži dva elementa (0 i 2), jer u grafu postoje dvije grane koje izlaze iz čvora 1, pri čemu jedna ide prema čvoru 0, dok druga grana ide prema čvoru 2

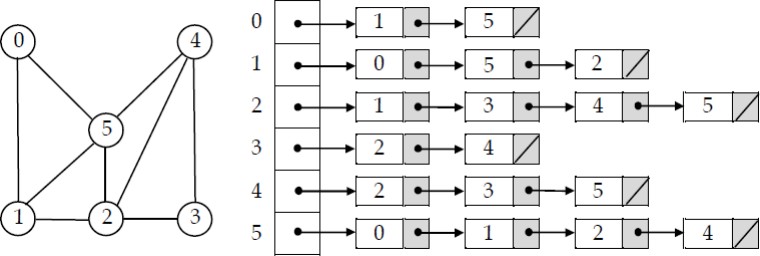
Zahtjevi za memorijskim prostorom kod liste susjedstva ovise i o broju grana i o broju čvorova u grafu. Za svaki čvor je potrebna jedna lokacija u nizu pokazivača, a za svaku granu je potreban po jedan čvor u listi. Odnosno, zahtjeve za memorijskim prostorom možemo izraziti kao O(|V|+|E|).

Primijetimo da je u nizu pokazivača potrebna lokacija čak i ako taj čvor nema niti jednog susjeda, kao što je to na slici ilustrirano za slučaj čvora 3.

Na slici je prikazano korištenje liste susjedstva za prikaz neusmjerenih grafova. Svaka grana u

neusmjerenom grafu, koja spaja čvorove vi i vj, može se tretirati kao dvije grane u usmjerenom

grafu: (vi, vj) i (vj, vi). To znači da će u listi susjedstva, za prikaz neke neusmjerene grane (vi, vj), biti potrebna dva čvora: jedan čvor u listi sa čelom na poziciji i, te jedan čvor u listi sa čelom na poziciji j.



## Matrice susjedstva vs. Lista susjedstva

Koji prikaz grafa je efikasniji u memorijskom smislu, ovisi od broja čvorova i grana u grafu.

Lista susjedstva sadrži informacije samo za one grane koje se pojavljuju u grafu, dok matrica susjedstva zahtijeva memorijski prostor za svaku potencijalnu granu, bez obzira da li dotična grana postoji.

Nadalje, lista susjedstva, za razliku od matrice susjedstva, zahtijeva dodatni prostor za smještanje pokazivača

Taj prostor može biti značajan, posebno ako uzmemo u obzir da se postojanje ili nepostojanje neke grane u grafu može zapravo prikazati sa samo jednim bitom.

Za grafove koji imaju relativno veliku povezanost, matrice susjedstva postaju sve efikasniji način prikaza. S druge strane, što graf ima manji stepen povezanosti, njegov prikaz listom susjedstva postaje sve efikasniji.

Vremenska složenost algoritama, koji kao ulazni objekat imaju grafove prikazane matricom susjedstva, često je veća nego što bi to bio slučaj da je za prikaz grafa korištena lista susjedstva.

Naime, algoritamskim procesom je potrebno posjetiti sve potencijalne susjede svakog čvora u grafu, što rezultira vremenskom složenošću O(|V|2 ). S druge strane, uz korištenje liste susjedstva,

obrađuju se samo stvarno prisutne grane, što rezultira vremenskom složenošću O(|V|+|E|).

Međutim, budući da se kod popunjenih grafova, broj grana |E| približava maksimalnom broju mogućih grana |V|2 -|V|, onda zaključujemo da veća efikasnost memorijskog prikaza grafa listom susjedstva dolazi do izražaja samo kod grafova koji imaju relativno mali broj grana.

## Obilazak čvorova grafa

Obilazak grafa podrazumijeva da se na sistematičan način po jednom posjete svi čvorovi, da bi se nad njima izvršila neka obrada.

Budući da kod grafa, za razliku od stabla, nema nekog istaknutog čvora od kojeg bi započinjao obilazak, potrebno je prije operacije obilaska zadati početni čvor.

Pretpostavimo da se organizirani način obilaska realizira pozivom procedure TRAZI (G, s), gdje s

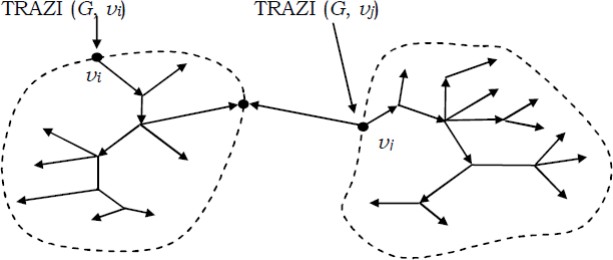
označava čvor od kojeg započinje obilazak.

Međutim, potrebno je naglasiti da u općem slučaju svi čvorovi u grafu nisu dostižni iz izabranog početnog čvora, što je karakteristično za grafove koji nisu povezani. Takve situacije su posebno izražene kod obilaska usmjerenih grafova.

Na slici je ilustrirana situacija u kojoj se graf sastoji od dvije komponente povezanosti. Iz početnog čvora s=vi se mogu, pozivom procedure TRAZI (G, vi) i slijeđenjem veza u grafu, posjetiti samo

čvorovi koji pripadaju prvoj komponenti, jer u grafu ne postoji grana koja omogućuje prelazak iz prve u drugu komponentu. Zato se za nastavak obilaska izabire novi početni čvor iz skupa neposjećenih

čvorova, te se obilazak nastavlja novim pozivom procedure TRAZI (G, vj).



Prema tome, da bi se omogućilo posjećivanje svih čvorova i u takvim situacijama, operacija obilaska se nastavlja tako da se ponovno izabire početni čvor iz skupa neposjećenih, pa se operacija ponavlja sve dok se ne posjete svi čvorovi u grafu. Osim toga, grafovi mogu sadržavati cikluse, pa je potrebno na neki način osigurati da ti ciklusi ne prouzrokuju ulazak algoritma obilaska čvorova u beskonačnu petlju

Prethodno navedeni potencijalni problemi pri obilasku grafa se mogu riješiti pridruživanjem oznake

posjecen[v] čvoru v.

Pri prvom posjećivanju čvora v, oznaka posjecen[v] se postavlja na vrijednost true.

Ako se algoritam obilaska susretne sa tako označenim čvorom ponovno, taj čvor se jednostavno preskače.

**OBIDJI-GRAF (G)**

**1 for each v =V[G] do**

**2 posjecen[v] <-- false**

**3 end\_for**

**4 for each v =V[G] do**

**5 if (posjecen[v] != true) then**

**6 TRAZI (G, v)**

**7 end\_if**

**8 end\_for**

Na početku algoritma obilaska svi čvorovi se označe kao neposjećeni, tako što se oznakama posjecen[v] pridruži vrijednost false.

Nakon što algoritam kompletira obilazak komponenti povezanosti grafa, ispitivanjem vrijednosti oznaka posjecen[v] se provjerava da li su ostali čvorovi posjećeni.

Ako neki pronađeni čvor nije označen kao posjećen, proces obilaska nastavljamo od tog neposjećenog čvora.

Ovaj pristup se koristi kako za obilazak usmjerenih, tako i za obilazak neusmjerenih grafova.

Algoritam kojim se obilaze svi čvorovi grafa je opisan procedurom OBIDJI-GRAF u tabeli.

U prvom koraku se svi čvorovi označe kao neposjećeni (linije 1-3), a zatim se u for petlji primjenjuje neki specifičan način organiziranog obilaska pojedinih komponenti povezanosti prisutnih u grafu (linije 4-8).

Postoje dvije uobičajene metode koje se koriste za realizaciju procedure TRAZI:

pretraživanje po dubini (eng. depth-first search-DFS);

pretraživanje po širini (eng. breadth-first search-BFS).

Obilazak po dubini započinje od zadatog početnog čvora, a zatim se od tog čvora slijedi put u jednom pravcu, što je više moguće.

Naime, nakon zadatog početnog čvora se posjećuje i označava jedan njegov susjed, nakon toga se posjećuje i označava susjedov susjed, itd. Na ovaj način se posjećuju čvorovi u jednom pravcu, sve dok se ne dođe do čvora čiji su svi susjedi već prethodno posjećeni ili do čvora koji nema susjeda.

Kada se tim putem više ne mogu posjećivati čvorovi, algoritam se vraća na zadnji prethodno posjećeni čvor, koji ima još neposjećenih susjeda.

Od tog čvora algoritam ponovno slijedi neki drugi put u grafu, posjećujući prethodno neposjećene

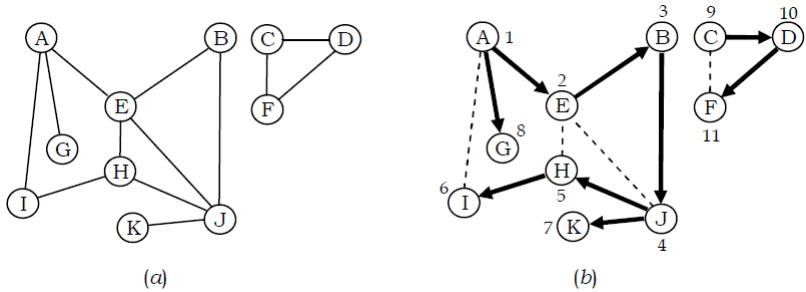
čvorove. Algoritam obilaska čvorova po dubini se završava kada se utvrdi da nema više neposjećenih čvorova, koji su dostižni iz prethodno posjećenih čvorova.

Proces obilaska neusmjerenog grafa po dubini je ilustriran na slici 12.8. Prikazani neusmjereni graf

sadrži dvije komponente povezanosti.

Obilazak započinje iz početnog čvora A posjećivanjem njegovog prvog susjeda, zatim susjedovog susjeda, sve dok se ne dođe do nekog čvora u dubini grafa koji više nema neposjećenih čvorova.

Prema tome, nakon čvora A posjećuju se čvorovi sljedećim redoslijedom: E, B, J, H i I (slika b).



Brojevi pored čvorova prikazuju redni broj pojavljivanja dotičnog čvora u poretku obilaska.

U ovom trenutku obilazak je kreirao sljedeći poredak: A→E→B→J→H→I.

Posjećivanjem čvora I, algoritam utvrđuje da su oba njegova susjeda (A i H) već posjećeni, pa se algoritam vraća na čvor H, te se utvrđuje da su također posjećeni i svi njegovi susjedi (E, I, J).

Algoritam se vraća na čvor J, te se posjećuje njegov prvi neposjećeni susjed K. Pošto čvor K nema neposjećenih susjeda, algoritam se opet vraća do čvora J, te se utvrđuje da čvor J nema više

neposjećenih susjeda. Algoritam obilaska se nastavlja vraćanjem prvo do čvora B, a zatim i do čvora E, te se utvrđuje da i ovi čvorovi također nemaju više neposjećenih susjeda.

Međutim, vraćanjem do čvora A, algoritam pronalazi i posjećuje njegovog prvog neposjećenog susjeda, a to je čvor G.

Čvor G nema više susjeda, pa se algoritam opet vraća do čvora A i utvrđuje da i taj čvor više nema neposjećenih susjeda. Na taj način je kreiran sljedeći obilazak prve komponente:

A→E→B→J→H→I→K→G.

Pošto u prvoj komponenti nema više neposjećenih čvorova, potrebno je iz skupa čvorova izabrati sljedeći neposjećeni čvor, te ponoviti isti proces i za drugu komponentu.

Prvi neposjećeni čvor je C, pa se iz datog čvora na prethodno opisani način kreira obilazak druge komponente: C→D→F. Prema tome, sveukupni obilazak je kreirao sljedeći poredak:

A → E → B → J→ H → I → K → G → C → D → F

Prethodno prikazani obilazak po dubini je opisan procedurom TRAZI PO-DUBINI. Argument v

predstavlja zadati početni čvor.

Kada god se posjeti neki čvor v, ova procedura rekurzivno posjećuje sve njegove susjede.

Mehanizam rekurzivnih poziva na različitim nivoima rekurzije osigurava smještanje posjećenih čvorova na stek.

**TRAZI-PO-DUBINI (G, L)**

**1 posjecentul true**

**2 for {u, (v, u) = E} do**

**3 if (posjecen[u] = true) then**

**4 TRAZI-PO-DUBINI (G, u)**

**5 end\_if**

**6 end\_for**

Kada je slijeđenjem grana u grafu po dubini onemogućeno daljnje napredovanje, tada mehanizam rekurzivnih poziva zapravo osigurava efekat vraćanja do prvog čvora na pređenom putu iz kojeg se može doći do još neposjećenih čvorova.

Procedura TRAZI-PO-DUBINI će osigurati da se obiđu čvorovi komponente povezanosti grafa kojoj pripada čvor v.

Za obilazak grafa koji se sastoji od više komponenti povezanosti, potrebno je pozvati proceduru

OBIDJI-PO-DUBINI.

Na početku se svi čvorovi označe kao neposjećeni (linije 1-3), a zatim se poziva rekurzivna procedura TRAZI-PO-DUBINI sa ulaznim argumentom v, pri čemu se poziv ostvaruje u for petlji, onoliko puta koliko u grafu ima komponenti povezanosti (linije 4-8).

**OBIDJI-PO-DUBINI (G)**

**1 for each v =V[G] do**

**2 posjecen v <-- false**

**3 end\_for**

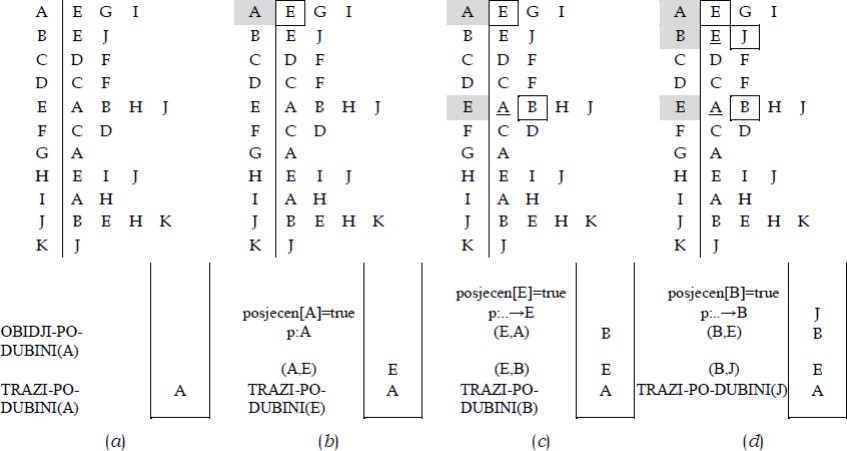
**4 for each v =V[G] do**

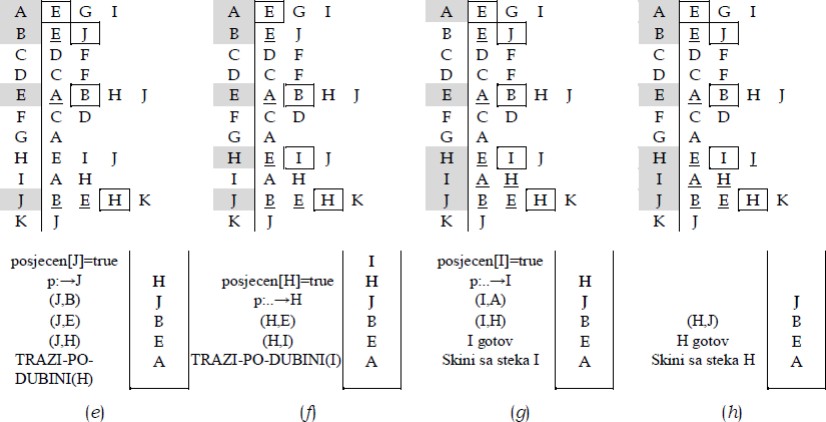
**5 if (posjecen[v] != true) then**

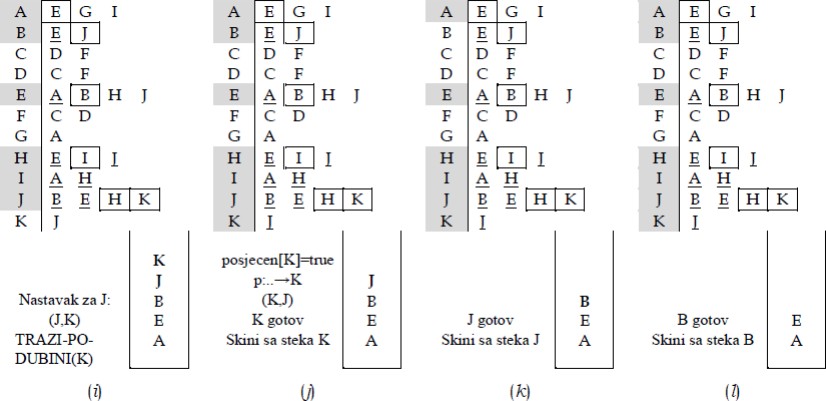
**6 TRAZI-PO-DUBINI (G, v)**

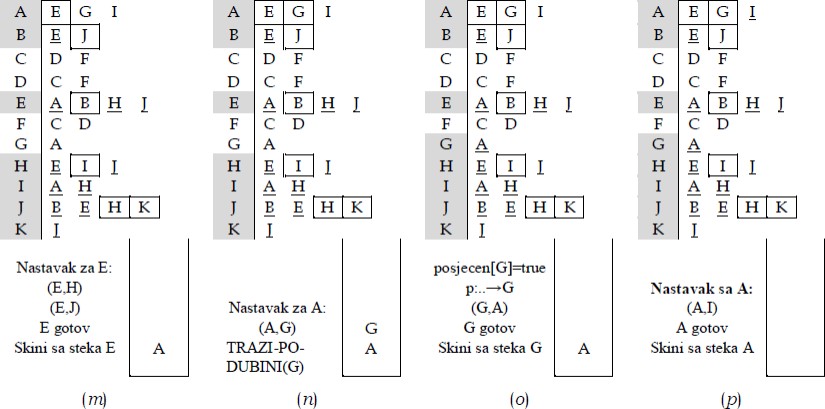
**7 end\_if**

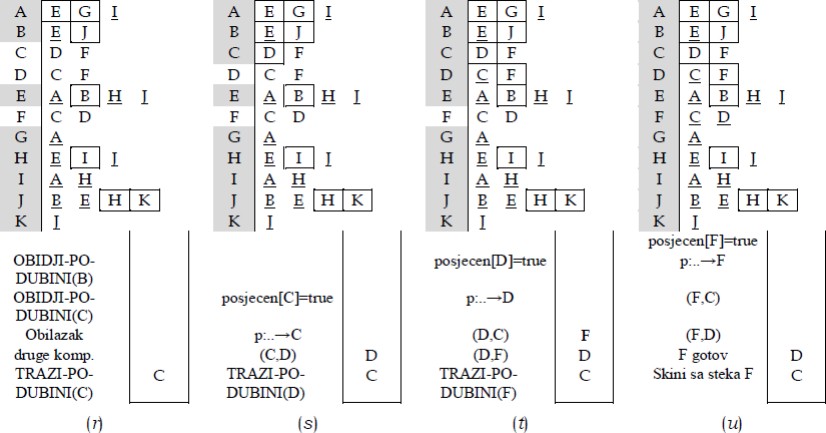
**8 end\_for**

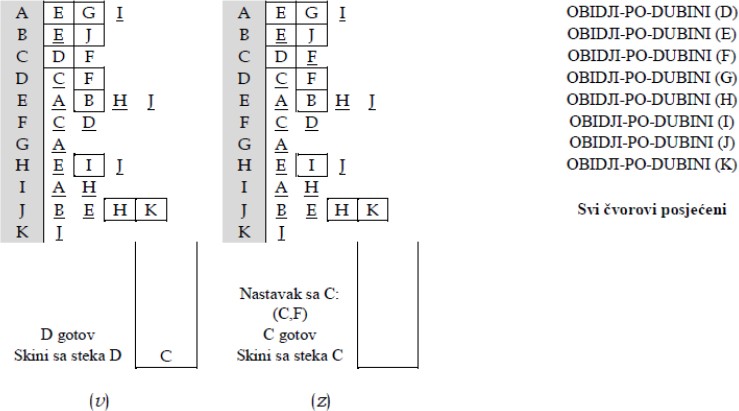












## Obilazak čvorova grafa po širini

Operacija obilaska po širini započinje od početnog čvora posjećivanjem svih njegovih susjeda.

Zatim se operacija nastavlja posjećivanjem svih neposjećenih susjedovih susjeda, sve dok se širenjem posjećivanja susjeda ne posjete svi neposjećeni susjedi.

Zapravo, možemo reći da se susjedi posjećuju po nivoima iste udaljenosti od početnog čvora.

Naime, ako sa d označimo udaljenost, koja predstavlja minimalan broj grana preko kojih se od

početnog čvora dolazi do datog čvora, onda se prvo posjećuju svi čvorovi na udaljenosti d, a zatim se prelazi na čvorove sa udaljenošću d+1.

Očito je da će za ovakav pristup biti potrebno voditi računa o redoslijedu elemenata, pa se za razliku

od obilaska po dubini, koji je koristio rekurzijski stek, kod obilaska po širini koristi red

Algoritam obilaska po širini je opisan procedurom TRAZI-PO-SIRINI gdje v označava zadati početni čvor.

Procedura TRAZI-PO-SIRINI će osigurati da se obiđu čvorovi komponente povezanosti grafa kojoj pripada čvor v.

**TRAZI-PO-SIRINI (G, v)**

**1 posjecen[v] <-- true**

**2 STAVI-U-RED (Q, U)**

**3 while (JE-LI-PRAZAN (Q) != true) do**

**4v<-- IZVADI-IZ-REDA (Q)**

**5for each {u: (v, u) = E} do**

**6if (posjecen[u] = false) then**

**7posjecen[u] <-- true**

**8STAVI-U-RED (Q, u)**

**9end\_if**

**10end\_for**

**11end\_while**

Za obilazak grafa koji se sastoji od više komponenti povezanosti, potrebno je pozvati proceduru

OBIDJI-PO-SIRINI.

**OBIDJI-PO-SIRINI (G)**

**1 for each v =V[G] do**

**2 posjecen[v] <-- false**

**3 end for**

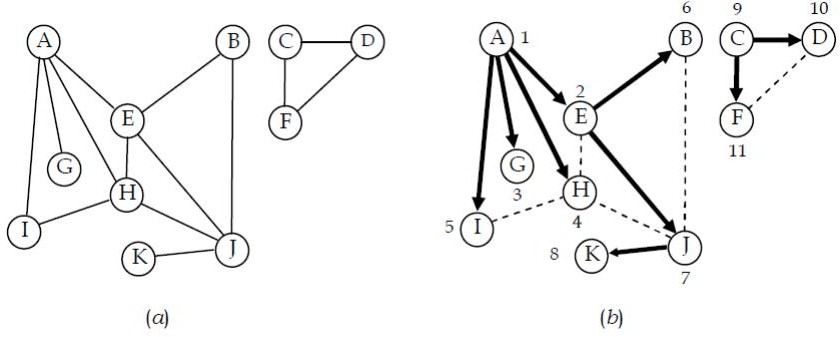
**4 for each v= V[G] do**

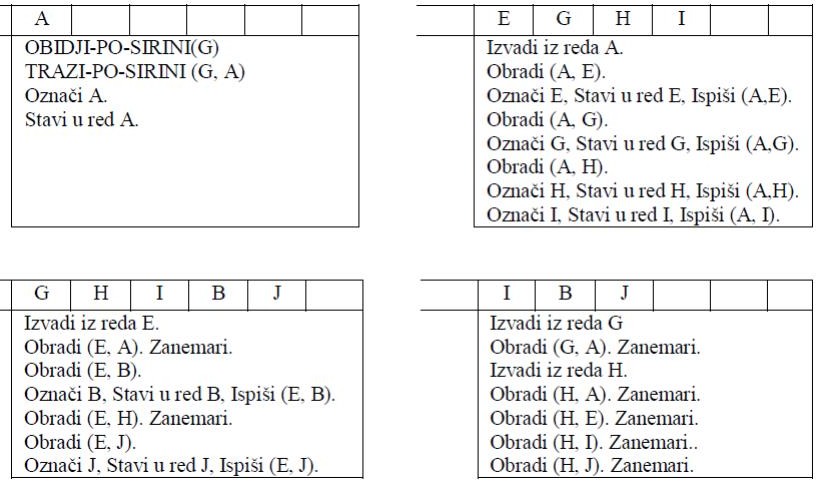
**5 if (posjecen[U] != true) then**

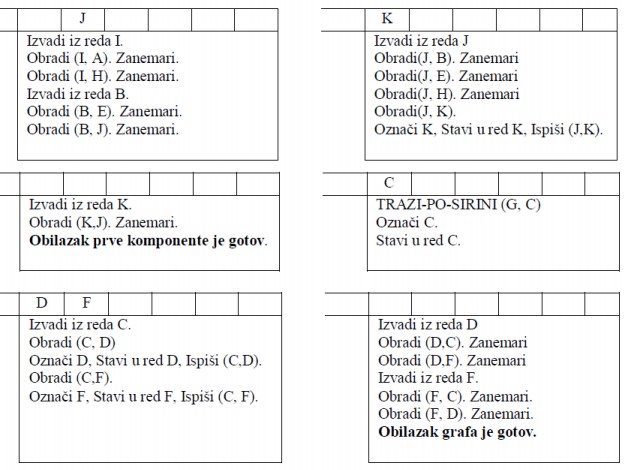
**6 TRAZI-PO-SIRINI (G, v)**

**7 end\_if**

**8 end\_for**





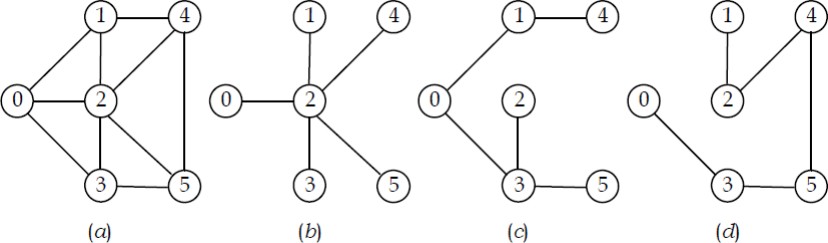


## Minimalno obuhvatno stablo

Neka je dat povezan i neusmjeren graf G = (V, E). Obuhvatno stablo grafa G je podgraf ST=(V, Est), gdje je Est ⊆ E, koji povezuje sve čvorove grafa G, tako da se ne pojavljuju ciklusi.

Prema tome, obuhvatno stablo ST grafa G sadrži sve čvorove iz tog grafa, kao i dovoljan broj grana, tako da se ti čvorovi povežu bez ciklusa.

Općenito, obuhvatna stabla nisu jedinstvena. Između bilo koja dva čvora u obuhvatnom stablu

postoji samo jedan put Obuhvatna stabla

## Algoritam pronalažanja minimalnog obuhvatnog stabla

Često se u različitim područjima primjene pojavljuju problemi koji se mogu formalno opisati kao problem pronalaženja „najpovoljnijeg“ načina za povezivanje svih čvorova grafa, odnosno kao problem pronalaženja minimalnog obuhvatnog stabla.

Pretpostavimo da imamo sljedeći problem. Potrebno je na određeni broj mjesta isporučiti signal

kablovske televizije.

Troškovi povezivanja između različitih mjesta ovise o različitim faktorima, kao što su: udaljenost,

različiti troškovi postavljanja kabla, itd. Ako mjesta na koja je potrebno isporučiti signal modeliramo čvorovima u grafu, a cijenu povezivanja težinskom funkcijom, onda se optimalno rješenje

povezivanja svih mjesta zapravo svodi na pronalaženje minimalnog obuhvatnog stabla.

Dakako, problemi i u drugim znanstvenim i inženjerskim područjima se mogu modelirati na sličan način, kao što su: transportni problemi, telekomunikacijske mreže, itd

Minimalno obuhvatno stablo MST grafa G=(V, E), sa težinskom funkcijom w:

E->R, je njegov podgraf MST = (V, Emst), Emst ⊆ E, koji sadrži sve čvorove iz V i samo one grane iz E, koje su dovoljne da svi čvorovi budu povezani, uz najmanji mogući zbir težina njegovih grana

Algoritmi pronalaženja minimalnog obuhvatnog stabla, koji će biti opisani na ovom mjestu, koriste tzv. „pohlepni pristup“.

Ovaj pristup se temelji na povećavanju minimalnog obuhvatnog stabla za jednu po jednu granu

Algoritam manipulira skupom grana A tako da se održava sljedeća invarijanta petlje:

prije svake iteracije, skup A predstavlja podskup nekog minimalnog obuhvatnog stabla.

U svakom koraku određujemo granu (u, v) koja se može dodati u skup A, a da se nakon dodavanja ne

naruši ova invarijanta na način da je A ∪ {(u, v)} također podskup minimalnog obuhvatnog stabla.

Takva grana se naziva sigurna grana za skup A, zato što se sigurno može dodati u skup A, a da se pri tome očuva invarijanta. Ključni dio algoritma je u načinu izbora grane, koja se dodaje u postepeno rastuće obuhvatno stablo

Invarijantu petlje koristimo na sljedeći način:

Inicijalizacija: Nakon prve linije, skup A je prazan, pa trivijalno zadovoljava invarijantu petlje.

Održavanje: Unutar while petlje (linije 2- 5) se invarijanta održava, jer se u skup A dodaju samo

sigurne grane.

Završetak: Sve grane koje se dodaju u skup A se nalaze u minimalnom obuhvatnom stablu, pa

procedura GENERIC-MST mora vratiti minimalno obuhvatno stablo

**GENERIC-MST (G, w)**

**1 A<--()**

**2 while (A ne formira obuhvatno stablo) do**

**3 pronaći granu (u, v) koja je sigurna za A**

**4 A<-- A U {(u, v)}**

**5 end\_while**

**6 return A**

Prethodno je već rečeno da je ključni dio algoritma način pronalaženja sigurne grane (linija 3).

Jedna sigurna grana mora postojati, zato što invarijanta diktira da nakon pronalaženja grane postoji obuhvatno stablo, takvo da je A⊆MST. Unutar while petlje, A mora biti pravi podskup od MST (A ⊂ MST), pa time mora postojati grana (u, v) ∈ MST, takva da (u, v) ∉ A, i da je (u, v) sigurna za A.

Prema tome, generički algoritam za određivanje minimalnog obuhvatnog stabla dodaje jednu po jednu sigurnu granu u postepeno rastuću šumu. Da bi specificirali neki konkretni algoritam, moramo odrediti za svaku iteraciju koju sigurnu granu treba dodati i kako je identificirati.

Da bismo lakše opisali algoritme za pronalaženje minimalnog obuhvatnog stabla, prvo ćemo uvesti neke pojmove kao što su rez i laka grana.

Rez (U, V-U) nekog neusmjernog grafa G=(V, E) predstavlja podjelu čvorova iz skupa V. Za neku

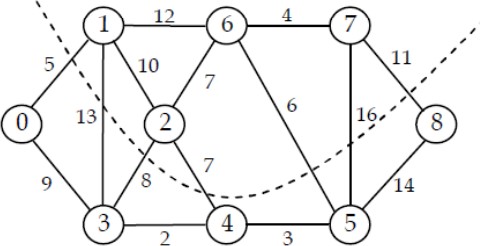
granu (u, v) koja pripada E kažemo da presijeca rez (U, V-U), ako jedan od krajnjih čvorova pripada skupu U, a drugi čvor pripada skupu V-U.

Ako skup A ne sadrži granu koja presijeca rez, onda kažemo da rez poštuje skup A.

Nadalje, za granu kažemo da je laka grana koja presijeca rez, ako je njena težina najmanja od svih

grana koje presijecaju rez.

Na primjer, na slici je ilustriran rez (U, V-U), gdje je U={0, 3, 4, 5, 8}, V-U={1, 2, 6, 7}. Grane koje presijecaju rez (U, V-U) su: (0, 1), (1,3), (2,3),(2,4), (5, 6), (5, 7) i (7, 8). Grana (0, 1) je laka grana, jer ima najmanju težinu od svih grana koje presijecaju dati rez



Postupak za prepoznavanje sigurnih grana je sljedeći:

Neka je G=(V, E) povezani, neusmjereni graf sa realnom težinskom funkcijom w, koja je definirana na

svim granama iz skupa E.

Nadalje, neka je A podskup skupa E, koji je uključen u neko minimalno obuhvatno stablo grafa G, neka je (U, V-U) bilo koji rez grafa G koji poštuje A, te neka je (u,v) laka grana koja presijeca rez (U, V- U).

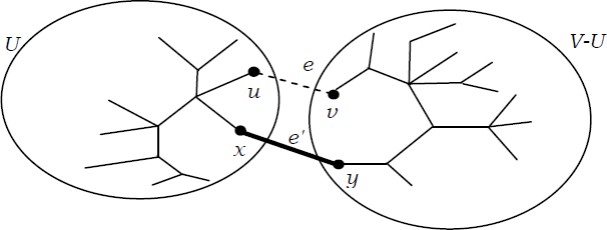
Onda je grana (u, v) sigurna za A.

Većina algoritama za određivanje minimalnog obuhvatnog stabla koristi jedno važno svojstvo tog stabla. To svojstvo je sljedeće:

Minimalno obuhvatno stablo sadrži svaku sigurnu granu

Ovo svojstvo možemo dokazati tako pokažemo nemogućnost suprotne pretpostavke.

Dakle, pretpostavit ćemo da minimalno obuhvatno stablo grafa G ne sadrži sigurnu granu. Neka je MST neko minimalno obuhvatno stablo grafa G. Pošto to stablo ne sadrži sigurnu granu e=(u, v), onda njeno uključivanje u MST formira ciklus, koji sadrži i ovu granu. Zato u stablu mora postojati neka grana e'=(x, y), x pripada U i y pripada (V-U), koja spaja skupove U i V-U (vidjeti sliku)



Kada ne bi postojala ta grana, ne bi se mogao zatvoriti ciklus od u do v, a da se pri tome grana e=(u,

v) ne prođe dva puta.

Ako uklonimo granu e'=(x, y), tada bi bio prekinut ciklus, pa bismo dobili drugo stablo MST', koje je također obuhvatno stablo, jer sadrži sve čvorove i sve grane, kao i stablo MST, osim što je umjesto grane e'=(x, y) u stablu MST' prisutna grana e=(u, v), te se može garantirati da ne postoje ciklusi.

Pošto smo pretpostavili da je grana e=(u, v) sigurna grana, onda je w(e') ≥ w(e), pa obuhvatno stablo MST' nema veću težinu od stabla MST. Drugim riječima, stablo MST', koje sadrži sigurnu granu e=(u, v), mora biti minimalno obuhvatno stablo.

Međutim, to je kontradiktorno početnoj pretpostavci, pa možemo zaključiti da je svojstvo time

dokazano.

## Kruskalov i Primov algoritam

Upravo na ovom svojstvu minimalnog obuhvatnog stabla se temelje i najpoznatiji algoritmi za

njegovo određivanje: Kruskalov i Primov algoritam.

Svaki od ova dva algoritma koristi specifičan način određivanja sigurne grane. Kod Kruskalovog algoritma, skup A je šuma. Sigurna grana koja se dodaje u A je uvijek grana sa najmanjom težinom u grafu, koja povezuje dvije odvojene komponente.

S druge strane, kod Primovog algoritma skup A formira jedno rastuće stablo. Sigurna grana, koja se dodaje u A, je uvijek grana sa najmanjom težinom u grafu, koja povezuje rastuće stablo sa čvorom koji u datom trenutku još nije u stablu.

## Kruskalov algoritam

Kruskalov algoritam pronalazi sigurnu granu za dodavanje u rastuću šumu, tako što među svim granama koje povezuju bilo koja dva stabla u šumi, pronalazi granu (u, v) sa najmanjom težinom.

Ovaj algoritam polazi od šume (T1, ...Tn), koja ne sadrži niti jednu granu, a sadrži n nepovezanih čvorova grafa.

Algoritmom se dodaje jedna po jedna grana, sve dok se šuma ne pretvori u minimalno obuhvatno

stablo grafa G.

Kruskalov algoritam je opisan funkcijom MST-KRUSKAL.

**MST-KRUSKAL (G, w)**

**1 A<--()**

**2 for each v =V[G] do**

**3 STVORI-SKUP (v)**

**4 end for**

**5 sortirati grane u E po težini u neopadajućem poretku**

**for each (u, v) = E, uzeto u neopadajućem poretku po težini do**

**if (NADJI-SKUP(u) != NADJI-SKUP(v)) then**

**A<-- A U {(u, v)}**

**UNIJA (u, v)**

**10 end\_if**

**11 end\_for**

**12 return A**

Implementacija Kruskalovog algoritma koristi disjunktne skupove podataka.

Svaki skup sadrži neki reprezentativni element, na temelju kojeg se jednoznačno može identificirati dotični skup.

Uvedimo sljedeće funkcije za rad nad disjunktnim skupovima, koje će se u nastavku koristiti za opis

Kruskalovog algoritma

STVORI-SKUP (x) - funkcija koja kreira novi skup čiji je jedini element x, koji je ujedno i

reprezentativni element.

UNIJA (x, y) - funkcija koja pravi uniju skupa Sx koji sadrži element x i skupa Sy koji sadrži element y. Pretpostavljamo da su x i y pripadali disjunktnim skupovima prije izvršavanja ove funkcije.

Reprezentativni element rezultirajućeg skupa može biti bilo koji element iz tog skupa.

NADJI-SKUP (x) - vraća pokazivač na reprezentativni element skupa koji sadrži x.

Na početku procedure MST-KRUSKAL se prvo inicijalizira skup A kao prazan skup (linija 1), a zatim se

kreira |V| stabala, tako da svako stablo inicijalno sadrži po jedan čvor iz skupa V (linije 2- 4).

Zatim se u neopadajućem poretku sortiraju sve grane po svojoj težini (linija 5). Nakon toga, slijedi for petlja za svaku granu (u, v), u kojoj se pomoću funkcije NADJI-SKUP provjerava da li krajnji čvorovi u i v pripadaju istom stablu (linija 7).

Ako je NADJI-SKUP (u) = NADJI-SKUP (v), onda čvorovi u i v pripadaju istom stablu, pa se grana (u, v) ne može dodati u skup A, jer bi se dodavanjem te grane formirao ciklus.

Ako čvorovi u i v pripadaju različitim stablima, grana (u,v) se dodaje u skup A (linija 8), a zatim se skupovi ta dva stabla spajaju u jedan skup (linija 9).

Vrijeme izvršavanja Kruskalovog algoritma za graf G=(V, E) ovisi od implementacije strukture podataka za prikaz disjunktnih skupova. Pretpostavimo da je implementacija šume disjunktnih skupova sa najbržom mogućom vremenskom složenošću.

Inicijalizacija skupa A uzima O(1) vremena, a vrijeme potrebno za sortiranje grana je O(Elog2 E).

Druga for petlja obavlja O(E) operacija NADJI-SKUP i UNIJA, nad šumom disjunktnih skupova. Nadalje, pošto imamo |V| operacija STVORI-SKUP, imamo O((V+E)α(V)) vremena, pri čemu je α vrlo sporo rastuća funkcija.

Budući da pretpostavljamo da je graf povezan, imamo da je |E| ≥ |V|-1, tako da operacije sa disjunktnim skupovima uzimaju O(E α(V)) vremena.

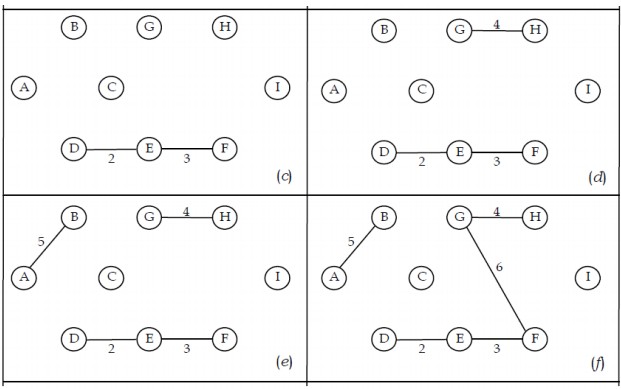
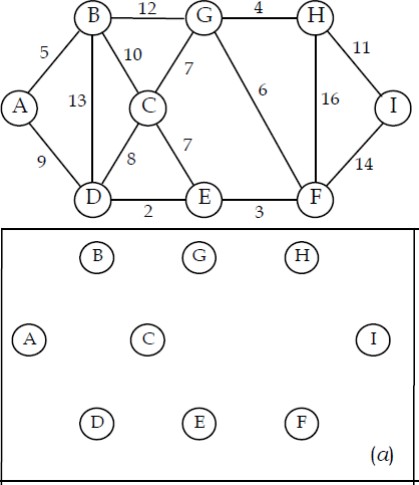
Pošto je α(|V|)=O(log2 V)=O(log2 E), ukupno vrijeme izvršavanja Kruskalovog algoritma je O(E log2

E). Nadalje, ako uzmemo da je |E| < |𝑉| 2 , onda imamo da je log2 |E| =O(log2 V).

Prema tome, Kruskalov algoritam se može implementirati na način da njegovo izvođenje ima vremensku složenost O(E log2 V ).

## Analiza stabla - primjer

Proces određivanja minimalnog obuhvatnog stabla započinjemo kreiranjem šume, koja se sastoji od svih čvorova iz skupa V.



Proces rasta obuhvatnog stabla nastavljamo dodavanjem grana (E, F), (G, H), (A, B) i (F, G) - a ti koraci su ilustrirani na slikama c-f.

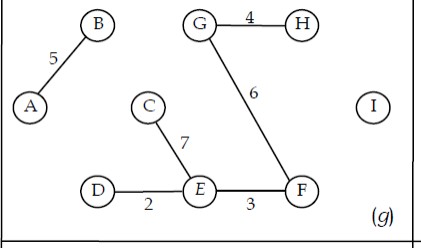
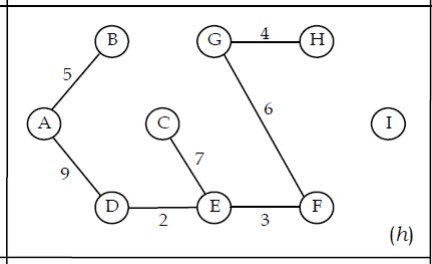
Skupovi povezanih čvorova su trenutno: {A, B}, {C}, {D, E, F, G, H} i {I}.

U sljedećem koraku utvrđujemo da postoje dvije grane u grafu G sa najmanjom težinskom vrijednošću 7, a to su grane (C, G) i (C, E).

Dodavanjem bilo koje od navedene dvije grane neće biti formiran ciklus, jer se kod obe grane krajnji čvorovi nalaze u različitim skupovima. Prema tome, možemo izabrati bilo koju od te dvije grane i dodati je u skup A.

U ovom primjeru je izabrana grana (C, E), pa je trenutno stanje u procesu određivanja minimalnog

obuhvatnog stabla prikazano na slici g.

Sljedeća grana koju je potrebno razmotriti je grana (C, D), jer ima najmanju težinsku vrijednost.

Međutim, utvrđujemo da su čvorovi C i D u istom skupu, što znači da bi se dodavanjem grane (C, D) kreirao ciklus, pa tu granu izostavljamo i dodajemo granu sa narednim najmanjim težinskim koeficijentom, a to je grana (A, D).

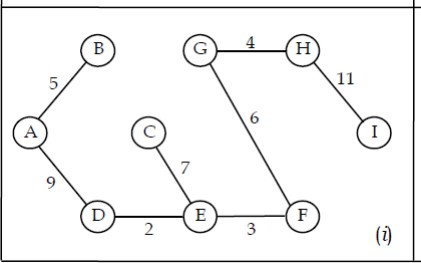
Trenutno stanje procesa formiranja obuhvatnog stabla je prikazano na slici h.

Sljedeća grana sa najmanjim težinskim koeficijentom je grana (B, C). Ponovno utvrđujemo da su

čvorovi B i C u istom skupu, što znači da bi se dodavanjem i grane (B, C) kreirao ciklus, pa i ovu granu izostavljamo i dodajemo granu sa narednim najmanjim težinskim koeficijentom, a to je grana (H, I).

U ovom trenutku imamo samo jedan skup povezanih čvorova {A, B, C, D, E, F, G, H, I}, pa

zaključujemo da su svi čvorovi povezani i da je proces formiranja obuhvatnog stabla završen. Na slici i je prikazan konačni izgled tog stabla



## Primov algoritam

Primov algoritam je opisan funkcijom MSTPRIM

Na početku izvršavanja algoritma početni čvor s ulazi u stablo, time što se skupovi U i E' (koji čine stablo) inicijaliziraju tako da je jedini element skupa U čvor s, dok se skup E' inicijalizira kao prazan (linije 1-2).

U svakoj iteraciji petlje while se bira grana sa najmanjom težinom od onih grana koje povezuju

čvorove koji su već sadržani u skupu U i čvorove iz skupa V-U. Čvor v pripada (V-U) i izabrana grana

(u, v) se uključuju u skupove U i E‘, tj. uključuju se u minimalno obuhvatno stablo MST (linije 5-6).

**MST-PRIM (G, S)**

**1 U<-- {s}**

**2 E<--()**

**3 while (U!= V) do**

**4 pronaći (u, v) => min {w(u, v): (u=U) and (v = (V-U))}**

**U<--U + {v}**

**E' <-- E'+{(u, v)}**

**end\_while**

**MST (U, E')**

**return MST**

Postupak formiranja rastućeg obuhvatnog stabla se završava kada svi čvorovi grafa G=(V, E) uđu u

obuhvatno stablo, tj. kada bude U = V.

Minimalno obuhvatno stablo MST je određeno formiranim skupovima U i E' (linija 8), pa algoritam na kraju kao rezultat vraća to stablo (linija 9).

Za efikasnu implementaciju algoritma MST-PRIM je ključno koristiti efikasan način pronalaženja grane najmanje težine, koja povezuje skupove U i V-U. Jedan od često korištenih pristupa je

korištenje prioritetnog red

**MST-PRIM-PQ (G, w, r)**

**1 for each u=V[G] do**

**2 kljuc[u] <--beskonacno**

**3 P[u] <-- NIL**

**4 end\_for**

**5 kljuc[r] <--0**

**6 Q<-- V[G]**

**while (Q!=()) do**

**u<--IZVADI-IZ-PRIOR-REDA (Q)**

**for each v e Adj[u] do**

**if (v= Q and w(u, v) < kljuc[v]) then**

**P[u] <-- U**

**kljuc[v] <-- w(u, v)**

**end\_if**

**end for**

**15end while**

Procedura MST-PRIMPQ opisuje jedan od mogućih pristupa koji koristi red s prioritetom minimuma.

Ulaz u algoritam su povezani graf G i korijen minimalnog stabla r.

U toku izvršavanja algoritma svi čvorovi v, koji još nisu u stablu se nalaze u redu Q sa prioritetom

minimuma ključa (kljuc[v]).

Za svaki čvor v, kljuc[v] je minimalna težina bilo koje grane koja povezuje čvor v sa čvorom u stablu. Kada ne postoji takva veza, vrijedi kljuc[v]=∞. U promjenljivoj P[v] se čuva roditelj čvora v u stablu.

U prvoj for petlji se postavlja ključ svakog čvora na ∞, dok se roditelji svih čvorova inicijalno

postavljaju na NIL (linije 1-4).

Ključ od korijena se postavlja na 0, što osigurava da se korijen prvi obrađuje (linija 5).

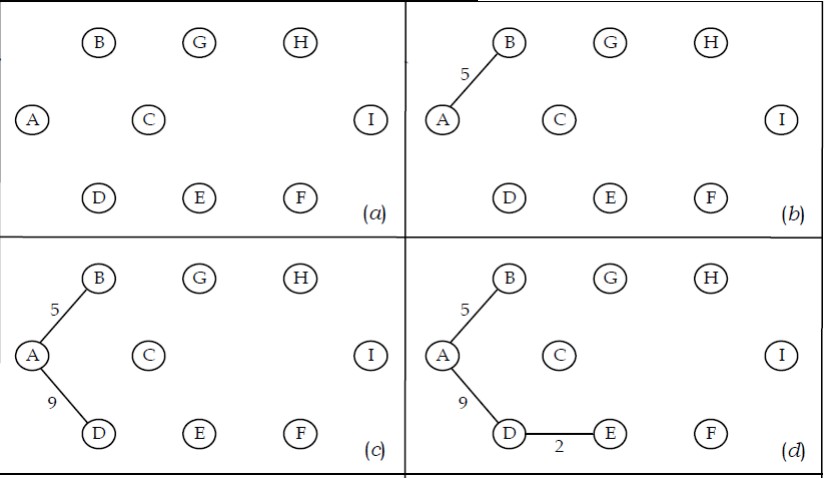
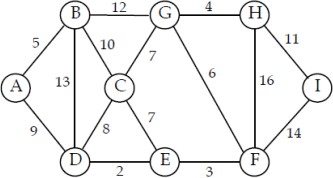
Red Q s prioritetom minimuma se inicijalizira tako da sadrži sve čvorove (linija 6).

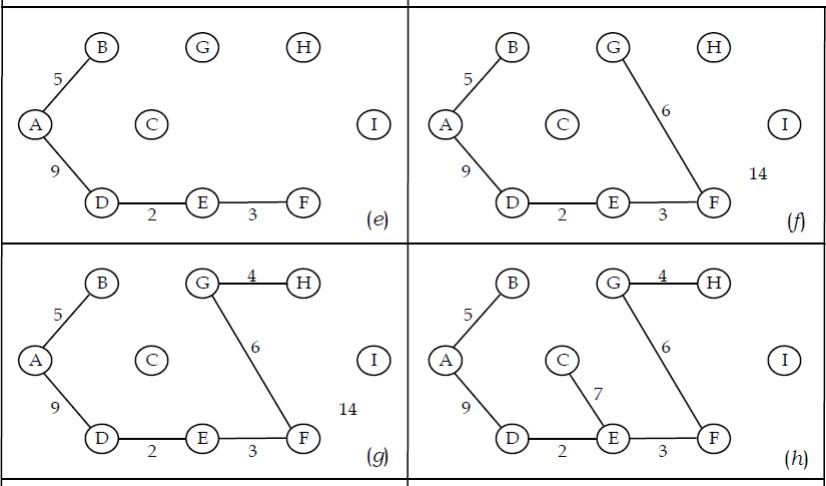
Unutar while petlje se iz reda vadi čvor u koji pripada Q, koji je na najmanjem rastojanju od stabla A

(linija 8).

U for petlji se ažuriraju promjenljive kljuc i P za svaki čvor v, koji je povezan sa čvorom u, ali nije u stablu (linije 9-14).

## Primjer

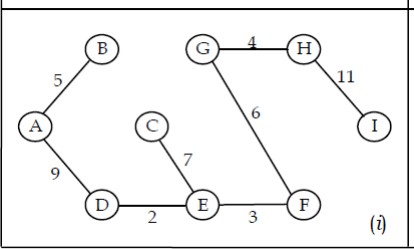




Primov algoritam se može implementirati sa ukupnim vremenom izvršavanja



što je jednako kao i kod Kruskalovog algoritma

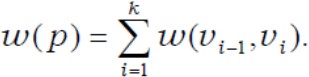


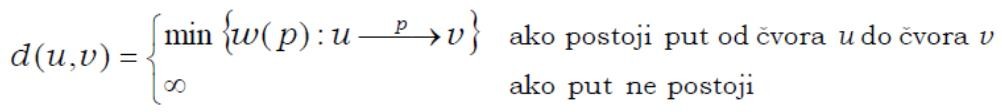
## Najkraći put u grafu

Rješavanje nekih problema koji se modeliraju težinskim grafovima se svodi na određivanje najkraćih puteva između čvorova.

Kod problema određivanja najkraćeg puta, dat je težinski usmjereni graf G=(V, E) sa težinskom funkcijom w. Težina puta p=(v0, v1, , vk) je definirana kao zbir težina grana od kojih se put p

sastoji



Težinu najkraćeg puta od čvora u do čvora v definiramo na sljedeći način

Najkraći put od čvora u do čvora v se definira kao bilo koji put čija je težina w(p)=d(u, v). Algoritmi određivanja najkraćeg puta se obično temelje na svojstvu da najkraći put između dva čvora sadrži najkraće puteve između svih čvorova koji se nalaze na tom putu.

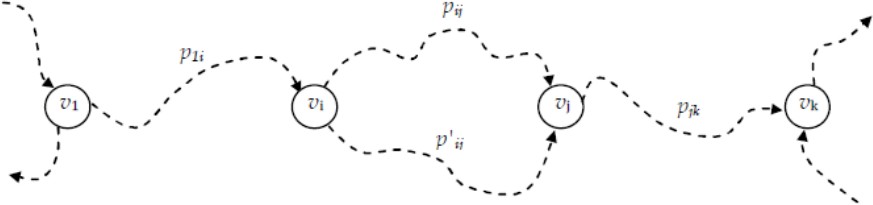
Neka je p=(v1, ...,vk) najkraći put od čvora v1 do čvora vk, te neka je podput pij=(vi, vi+1,. , vj) puta p

od čvora vi do čvora vj za svako i i j, takvo da vrijedi 1 <= i <=j <= k. Tada je pij najkraći put od čvora vi do čvora vj.

Ovo svojstvo možemo dokazati tako da put podijelimo na puteve p1i, pij i pjk (slika). Ukupna težina puta p je w(p1k)=w(p1i)+w(pij)+w(pjk). Pretpostavimo da postoji drugi put pij' između čvorova vi i vj, koji je najkraći.

Tada je ukupna težina puta w(p1k) > w(p1i) + w(p'ij) + w(pjk), što je kontradiktorno početnoj

pretpostavci. Dakle, zaključujemo da najkraći put između čvorova v1 i vk mora sadržavati najkraće puteve između svih čvorova vi i vj, koji se nalaze na tom putu, 1 <=i <= j <= k.



Algoritmi nalaženja najkraćeg puta se temelje na tzv. principu relaksacije.

Za svaki čvor v koji pripada V se čuva promjenljiva d[v], koja prikazuje gornju granicu težine puta od

polazišta s do čvora v.

Promjenljiva d[v] se naziva procjena najkraćeg puta. Procjene najkraćeg puta i prethodnike

inicijaliziramo procedurom INITIALIZE-SINGLE-SOURCE

**INITIALIZE-SINGLE-SOURCE (G, s)**

**1 for each v =V[G] do**

**2 d[v]<--beskonacno**

**3 pi <-- NIL**

**4 end\_for**

**5 d[s]<--0**

Proces relaksacije neke grane (u, v) se sastoji u ispitivanju da li je moguće skratiti dosadašnji najkraći put do čvora v, prolaskom kroz čvor u.

Ako je to moguće, onda se ažuriraju promjenljive d[v] i π[v].

Korak u relaksaciji može smanjiti vrijednost procjene najkraćeg puta d[v] i promjeniti prethodnika

čvora v, koji se čuva u π[v].

Relaksacija na grani (u,v) se obavlja pozivom procedure RELAX (u, v, w)

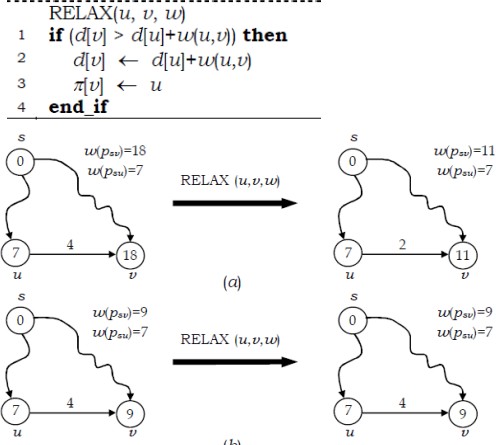
**RELAX(u,v,w)**

**1 if(d[]>d[u]+w(u,v)) then**

**2 d[v]<--d[u]+w(u,v)**

**3 pi[v]<--u**

**4 end\_if**



## Dijkstra algoritam

Na ovom mjestu ćemo opisati Dijkstra algoritam pomoću kojeg se određuje najkraći put iz jednog polazišta na težinskom usmjerenom grafu G=(V, E), u slučaju kada su težine svih grana nenegativne.

Dakle, pretpostavljamo da je w(u,v) ≥ 0, za svaku granu (u, v) koja pripada E

Dijkstra algoritam čuva skup S čvorova čije su konačne težine najkraćih puteva od polazišta s već određene. Ovaj algoritam proširuje skup u petlji, tako što izabire čvor u koji pripada V-S sa najmanjom procjenom najkraćeg puta, te relaksira sve grane koje polaze iz čvora u.

Algoritam je opisan procedurom DIJKSTRA

**DIJKSTRA (G, s)**

**1 INITIALIZE-SINGLE-SOURCE (G, s)**

**2 S<--()**

**3 Q<--V[G]**

**4 while (Q!=()) do**

**5 u<-- EXTRACT-MIN (Q)**

**6 S<--S U {u}**

**7 for each v = Adj[u] do**

**RELAX (u, v, w)**

**end\_for**

**end\_while**

Na početku algoritma se procedurom INITIALIZE SINGLE-SOURCE inicijaliziraju nizovi d i π (linija 1), a

zatim se skup S inicijalizira kao prazan skup (linija 2).

Skup S sadrži čvorove za koje je do određenog trenutka pronađen najkraći put od polaznog čvora.

Nakon toga se inicijalizira skup Q, tako da sadrži sve čvorove grafa (linija 3).

U skupu Q se nalaze ostali čvorovi za koje postoje trenutne procjene najkraćih rastojanja od

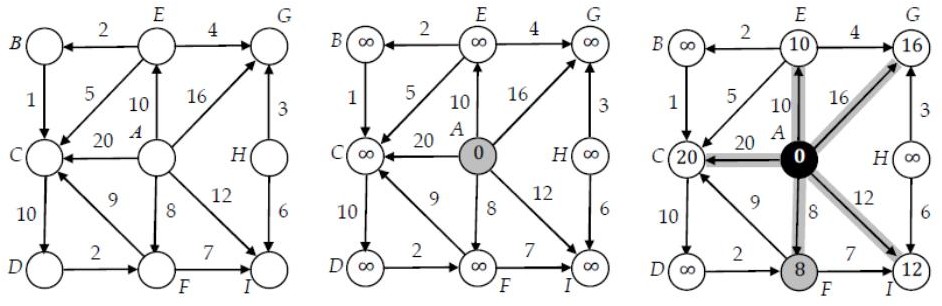
polaznog čvora, koje se kasnije mogu eventualno ažurirati, ako se u algoritamskom procesu pronađu kraći putevi do tih čvorova. Skup Q zapravo sadrži čvorove iz skupa V-S.

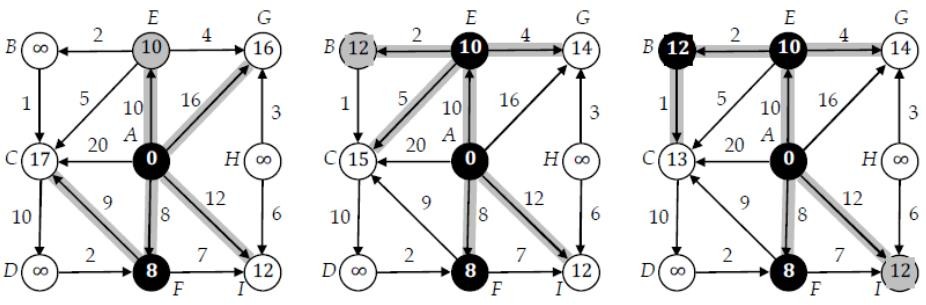
Slijedi while petlja, u kojoj se pri svakoj iteraciji pozivom funkcije EXTRACT-MIN (Q) iz skupa Q izabire čvor u (linija 5), koji ima najmanju procjenu najkraćeg puta od svih čvorova koji se nalaze u skupu Q, odnosno od svih čvorova iz skupa V-S.

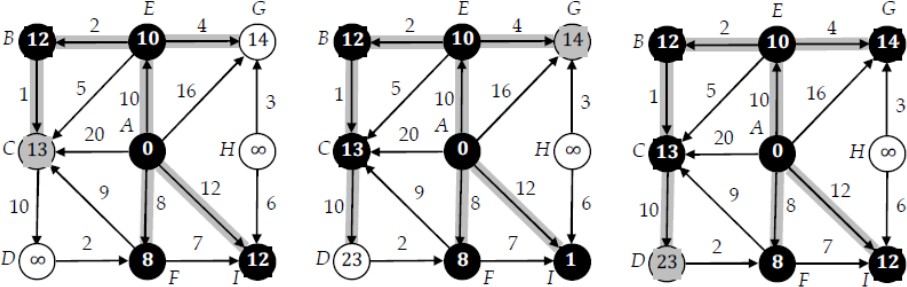
Čvor u postaje element skupa S (linija 6).

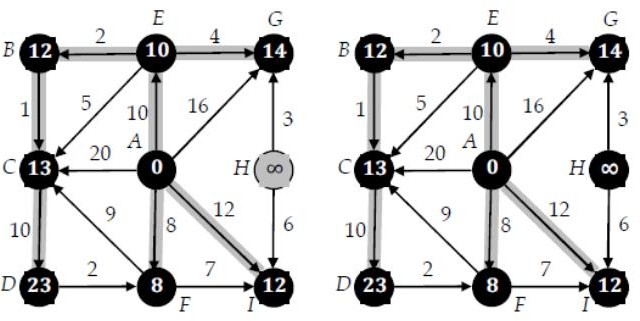
Na kraju se relaksiraju sve grane (u,v), koje polaze iz čvora u (linije 7-9).

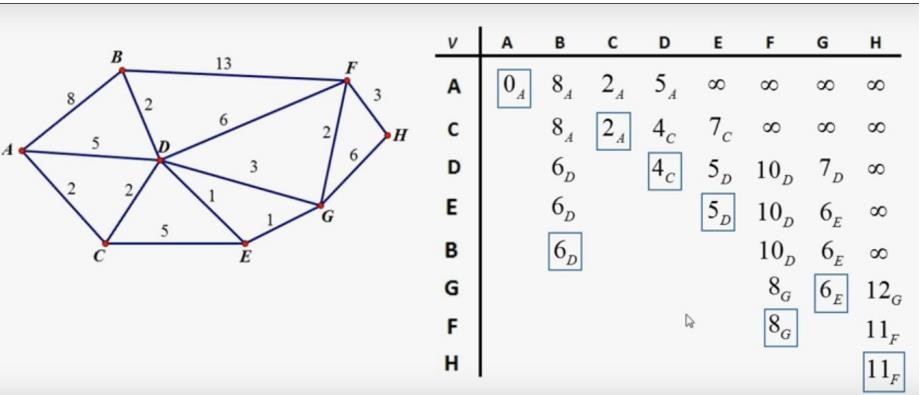
Tom relaksacijom se za sve čvorove v, koji su susjedni čvoru u, ažurira procjena d[v] i prethodnici π[v], ako se može postići kraći put do čvora v preko čvora u.











# (Predavanje VII)

**Algoritmi pretraživanja**

• Pod pretraživanjem podrazumijevamo utvrđivanje da li se zadata tražena vrijednost pojavljuje u nekom

skupu vrijednosti.

• U ovom poglavlju će biti opisani osnovni algoritmi pretraživanja, a to su algoritmi sekvencijalnog i binarnog pretraživanja.

• Složeniji algoritmi se temelje na ovim algoritmima

## Sekvencijalno pretraživanje

• Sekvencijalno pretraživanje je najjednostavniji algoritam pretraživanja.

• Algoritmi pretraživanja podrazumijevaju pronalaženje neke zadate vrijednosti.

• Niz se pretražuje sve dok se zadata tražena vrijednost ne pronađe, ili dok algoritam kao svoj izlaz ne odredi da tražena vrijednost nije u nizu

• Na primjer, pretpostavimo da su u nizu jednostavni podaci cjelobrojnog tipa, onda se tražena vrijednost uzastopno uspoređuje sa elementima iz niza, sve dok se ne pronađe podudaranje tih vrijednosti, ili dok se ne ispitaju svi elementi u nizu.

• Kod kompleksnijih tipova podataka obično se neki atribut definira kao ključni atribut. Tada se vrši poređenje tražene vrijednosti samo sa vrijednostima ključnog atributa, dok se ostali atributi

kompleksnijih tipova za potrebe pretraživanja ignoriraju.

• Na primjer, neki niz može sadržavati podatke o radnicima neke kompanije. Radnici mogu biti prikazani pomoću složenog tipa koji uključuje atribute kao što su: prezime radnika, ime radnika, adresa, telefonski broj, radno mjesto, itd.

• Nadalje, za potrebe pretraživanja može se definirati i šifra radnika koja je jedinstvena za svakog radnika u nizu. Šifra radnika se može koristiti kao ključni atribut, pa se kod pretraživanja kao tražena vrijednost tada koristi neka konkretna šifra radnika

• Sekvencijalni algoritam pretraživanja ostvaruje pronalaženjem traženog ključa tako da se uspoređuje jedna po jedna vrijednost ključa u nizu sa traženim ključem, počevši od prvog elementa u nizu, sve dok se traženi ključ ne pronađe ili dok se ne dođe do kraja niza.

• Algoritam kao znak uspješnog pretraživanja može vratiti indeks pronađenog elementa u nizu, ili neku predefinisanu vrijednost, ako je pretraživanje bilo neuspješno.

SEKVENC-PRETRAZI(K, k) SEKVENC-PRETRAZI-M(K, k)

i🡨0 K[n]🡨k

while(i <= n-1)do i🡨0

if(K[i] = k)then while(K[i] != k)do

return i i🡨i+1

else end\_while

i🡨i+1 if(i != n)then

end\_if return i

end\_while end\_if

return -1 return -1

• Pretpostavimo da su ključevi organizirani u niz K[0..n-1], koji sadrži n elemenata.

• Algoritam sekvencijalnog pretraživanja je opisan funkcijom SEKVENCPRETRAZI koja je prikazana na slici.

• Argumenti funkcije su niz K i traženi ključ k. Funkcija vraća poziciju traženog ključa u nizu ako je

pretraživanje završilo uspješno (pronalaženjem ključa), ili vrijednost -1 ako traženi ključ nije prisutan u nizu K.

• Ako niz K sadrži više traženih ključeva, funkcija će vratiti najmanji indeks lokacije sa traženim ključem k

• Algoritam sekvencijalnog pretraživanja SEKVENCPRETRAZI se može do određene mjere učiniti efikasnijim, ako se on modificira na način kako je to opisano na slici u obliku funkcije SEKVENC- PRETRAZI-M.

• Na početku algoritamskog procesa se na n-tu poziciju u nizu kao novi element dodaje traženi ključ (linija

1).

* Na taj način se postiže da se u nizu mora pronaći traženi ključ na nekoj od pozicija 0..n. Ako je traženi

ključ pronađen na zadnjoj (n-toj) poziciji u nizu (i=n), to zapravo predstavlja neuspješno pretraživanje, pa funkcija vraća -1 (linija 9).

* S druge strane, ako je traženi ključ pronađen na nekoj drugoj poziciji (i ≠ n), to znači da je on prisutan u

nizu, te funkcija vraća indeks pozicije u nizu na kojoj je ključ pronađen (linije 6-8).

* Prednost modificiranog sekvencijalnog algoritma pretraživanja ogleda se u činjenici da nije potrebno poslije svake usporedbe provjeravati da li je varijabla i izašla iz raspona 0..n- 1, kao kod običnog algoritma sekvencijalnog pretraživanja.
* Na taj način se dvostruko smanjuje broj operacija poređenja, što modificirani algoritam čini u određenoj

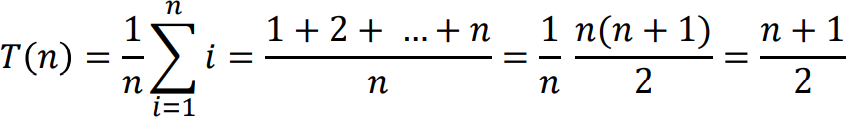
mjeri efikasnijim.

#### Efikasnost sekvencijalnog pretraživanja.

* Tri mjere za efikasnost sekvencijalnog pretraživanja:
* najbolji,
* najgori i
* prosječni slučaj.
* Najbolji slučaj je situacija u kojoj je traženi ključ pozicioniran na prvom mjestu u nizu, jer je za njegovo pronalaženje potrebna samo jedna operacija usporedbe, bez obzira na dužinu niza n.
* S druge strane, najgori slučaj je situacija u kojoj je traženi ključ pozicioniran na zadnjem mjestu u nizu ili ako je pretraživanje neuspješno, tj. ako traženi element nije uopće prisutan u nizu.
* Tada je potrebno vrijednost traženog ključa usporediti sa vrijednostima svih ključeva u nizu, što znači da

je potrebno n operacija usporedbi.

* Prosječan slučaj se dobija izračunavanjem prosječnog broja potrebnih operacija usporedbi tako da se razmotre sve moguće situacije pretraživanja, uzimajući u obzir sve moguće klase ulaza.
* Ako pretpostavimo da su svi ključevi jednako vjerojatni kao traženi ključevi, lako intuitivno možemo zaključiti da će u prosjeku biti potrebno pretražiti oko pola niza.
* To znači da možemo procijeniti da će biti potrebno oko n/2 operacija usporedbi. Ili, u cilju formalnijeg i preciznijeg razmatranja prosječnog slučaja kod sekvencijalnog pretraživanja, možemo reći da broj usporedbi kod uspješnog pretraživanja ovisi o poziciji ključa u listi.
* Uz pretpostavku da je traženi ključ na nekoj i-toj poziciji, tada je kod uspješnog pretraživanja potrebno i operacija usporedbi. Ako još pretpostavimo da su svi ključevi jednako vjerojatni, onda je pri uspješnom pretraživanju ukupan broj operacija poređenja T(n).



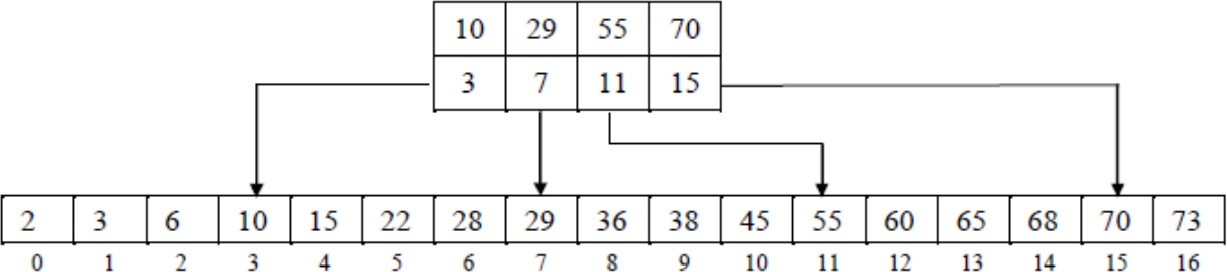
* Imajući u vidu da je stupanj rasta broja ostalih operacija koje se izvršavaju u sklopu while petlje jednak stupnju rasta operacija usporedbi, možemo zaključiti da je vremenska složenost algoritma

sekvencijalnog pretraživanja O(n), što nadalje znači da efikasnost ovog algoritma u prosječnom slučaju

nije dobra, pa je ovaj algoritam prihvatljiv samo za male vrijednosti n

#### Upotreba pomoćnih indeksnih struktura

* Efikasnost sekvencijalnog pretraživanja se može povećati kod sortiranih nizova. Jedan od načina je korištenje pomoćne indeksne strukture. Primjer povećanja efikasnosti pretraživanja sortiranog niza primjenom pomoćne indeksne strukture je ilustriran na slici.



* Pomoćna indeksna struktura sadrži određeni broj indeksa pozicije iz niza i pripadajuće vrijednosti

ključeva. Prilikom pretraživanja prvo se pretražuje indeksna struktura, da bi se pronašle dvije susjedne vrijednosti od kojih je jedna manja, a druga veća od traženog ključa. Nakon toga, pretraživanje se

nastavlja samo u onom dijelu niza koji je definiran donjom i gornjom granicom iz indeksne strukture ili se

odmah pronađe ključ u indeksnoj strukturi.

* Ako, na primjer, pretpostavimo da je traženi ključ 38, algoritam će u indeksnoj strukturi identificirati donju granicu d=29 i gornju granicu g=55, odnosno algoritam će suziti pretraživanje niza na pozicije u rasponu 8..10. Dakle, potrebne su tri operacije poređenja da se utvrdi raspon pozicija u nizu 8..10, te dodatne dvije operacije poređenja da se pronađe traženi ključ 38.
* Na taj način je potrebno ukupno 4 operacije usporedbi, dok bi za sekvencijalno pretraživanje bez korištenja indeksne strukture bilo potrebno 10 operacija usporedbi.

#### Upotreba indeksnih struktura sa dva i više nivoa

* Ako nizovi imaju izuzetno veliki broj elemenata, onda se radi dodatnog povećanja efikasnosti mogu

eventualno koristiti i indeksne strukture na dva hijerarhijska nivoa.

* Tada se indeksna struktura na višem hijerarhijskom nivou koristi da indeksira sadržaj indeksne strukture na nižem hijerarhijskom nivou, dok indeksna struktura na nižem nivou indeksira sadržaj elemenata niza

#### Poboljšanje efikasnosti pretraživanja preuređivanjem niza

* Spomenimo još da se efikasnost sekvencijalnog pretraživanja može poboljšati i u slučajevima kada su vjerojatnosti traženja pojedinih ključeva nejednake i unaprijed poznate
* Tada se ključevi u nizu mogu rasporediti tako da se ključevi sa većim vjerojatnoćama pretraživanja

stavljaju na pozicije bliže početku niza, dok se ključevi sa manjim vjerojatnoćama pretraživanja stavljaju na pozicije bliže kraju niza

* Ako vjerojatnoće nisu unaprijed poznate, jedna od mogućnosti bi mogla biti mjerenje frekvencije traženja pojedinih ključeva da bi se niz mogao dinamički preuređivati u skladu s izmjerenim frekvencijama. Međutim, brojanje frekvencija i dinamičko preuređivanje niza zahtijeva povećanje potrebnih operacija, što prethodno opisani pristup čini neprikladnim
* Jedan od mogućih pristupa bi mogao biti da pronađeni ključ zamijeni poziciju sa ključem koji je

pozicioniran neposredno ispred njega.

* Kod ovog pristupa efikasnost je dobra i pri preuređivanju povezanih listi, kao i pri preuređivanju nizova.
* Drugi mogući pristup bi mogao biti da se pronađeni ključ odmah pozicionira na prvo mjesto, a svi ključevi

koji su prije bili pozicionirani ispred pronađenog ključa bi se pomjerili za jedno mjesto ka kraju niza. Važno je istaknuti da je ovaj pristup efikasnije primijeniti na preuređivanju povezanih listi, nego na preuređivanju nizova

## Binarno pretraživanje - uvod

* Ključni nedostatak sekvencijalnog pretraživanja zasniva se činjenici da se pri svakoj neuspješnoj operaciji poređenja odbacuje samo jedan element niza koji se pretražuje.
* Ovaj nedostatak je čak prisutan i kod nizova koji su sortirani po ključevima, iako kod njih informaciju o

usporedbi traženog ključa i ključa u nizu pretraživanja možemo puno bolje iskoristiti,

* Ako se traženi ključ uspoređuje sa nekim ključem na poziciji i u nizu pretraživanja koji je sortiran u

rastućem poretku, te ako se utvrdi da je traženi ključ veći od ključa na poziciji i, onda lako zaključujemo da je traženi ključ veći i od svih ostalih ključeva na pozicijama 1 do i-1.

* To znači, da se svi ključevi na pozicijama 1 do i-1 mogu odbaciti u nastavku algoritma pretraživanja
* Najefikasnije je vršiti poređenje na polovini intervala niza pretraživanja, te nakon odbacivanja pola

elemenata niza ponovno vršiti polovljenje preostalog niza, sve dok se ne pronađe traženi ključ ili dok se ne utvrdi da traženi ključ nije prisutan u nizu.

* Ovakav pristup se naziva binarno pretraživanje
* Uvedimo oznake:
* k - traženi ključ;
* K - sortirani niz pretraživanja (u rastućem poretku);
* vrh - zadnja pozicija u preostalom dijelu niza nakon polovljenja;
* dno - prva pozicija u preostalom dijelu niza nakon polovljenja;
* srednji - srednja pozicija u nizu
* Varijabla srednji predstavlja indeks središnjeg ključa u nizu (K[srednji]), koji zapravo dijeli niz na dvije

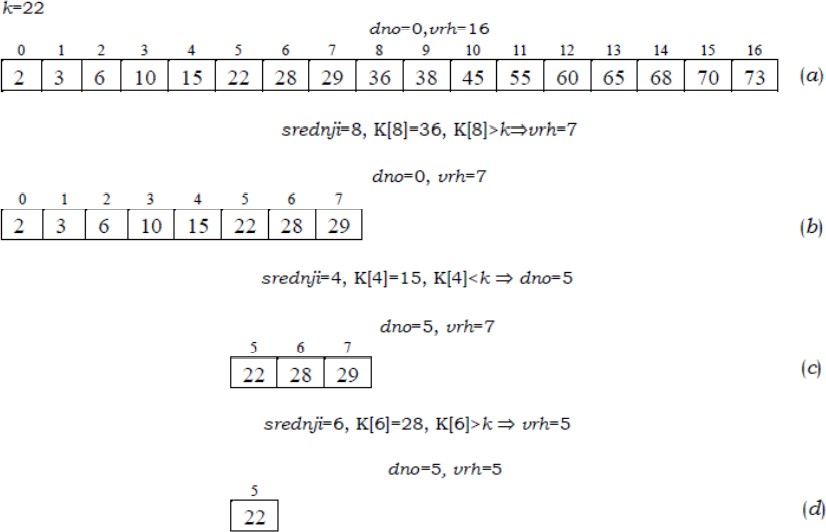
polovine: lijevu polovinu koja se sastoji od elemenata sa indeksima isrednji.

* Preostali dio niza K, koji se dobija polovljenjem niza iz prethodne iteracije, je definiran cjelobrojnim varijablama dno i vrh
* U svakoj iteraciji algoritma se izračunava indeks središnjeg ključa srednji=(vrh+dno)/2, te se provjerava da li je središnji ključ K[srednji] jednak ključu k kojeg tražimo
* Naravno, ova provjera najčešće pokazuje da središnji ključ nije traženi ključ, posebno u početnim

iteracijama, pa se algoritam nastavlja tako da se na temelju usporedbe središnjeg ključa K[srednji] i traženog ključa k odbacuje ili lijeva ili desna polovina niza

* Ako je ispunjeno K[srednji]>k, onda se odbacuje desna polovina niza. Inače, odbacuje se lijeva polovina niza. Nadalje, ako se odbacuje desna polovina niza, onda se ažurira varijabla vrh, tako da joj se pridruži vrijednost vrh=srednji-1. S druge pak strane, ako se odbacuje lijeva polovina niza, onda se ažurira

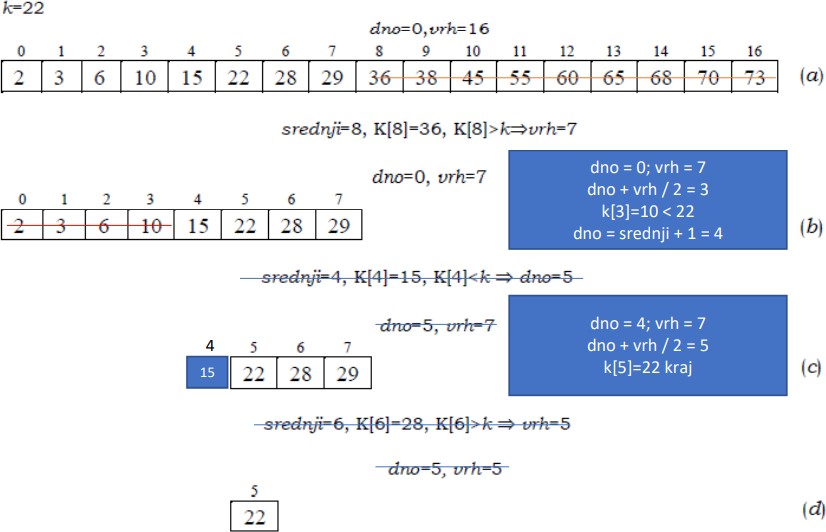
vrijednost varijable dno, tako da joj se pridruži vrijednost dno=srednji+1



* Na slici a je prikazan niz nad kojim se obavlja binarno pretraživanje, pri čemu je traženi ključ k=22, dok su početne vrijednosti varijabli dno=0 i vrh=16, jer je u prvoj iteraciji preostali niz jednak zapravo početnom nizu.
* U prvoj iteraciji se izračunava indeks središnjeg elementa srednji=8, te se uspoređuje vrijednost

elementa K[8]=36 sa traženim ključem k=22. Pošto je 36>22, to znači da se treba odbaciti desna polovina niza, odnosno potrebno je ažurirati varijablu vrh=srednji-1=7.

* Dakle, preostali dio niza u sljedećoj iteraciji će biti K[0..7], jer je dno=0 i vrh=7.



* Preostali dio niza u ovoj iteraciji je prikazan na slici b. Ponovno se izračunava indeks središnjeg elementa srednji=3. Sada je središnji element K[3]=10 manji od traženog ključa k=22, pa je potrebno odbaciti lijevu polovinu niza sa indeksima lokacija i <= 3.
* Taj dio niza se odbacuje tako što se ažurira varijabla dno=srednji+1=4. Prema tome, preostali dio niza će

biti K[4..7], jer je dno=4 i vrh=7.

* Na slici je prikazana funkcija BINPRETRAZI koja opisuje algoritam binarnog pretraživanja, koji kao ulaz

uzima niz ključeva K i traženi ključ k, a kao izlaz daje indeks pozicije u nizu na kojoj je ključ k pronađen.

* Ako traženi ključ k nije pronađen, funkcija vraća -1

BIN-PRETRAZI(K,k)

dno🡨0

vrh🡨duzina[K]-1

while(vrh >= dno)do

srednji🡨(dno+vrh)/2

if(K[srednji] = k)then

return srednji

else if(K[srednji] > k)then

vrh🡨srednji-1

else

dno🡨srednji+1

end\_if

end\_while

return -1

BIN-PRETRAZI-M(K, k)

dno🡨0

vrh🡨duzina[K]-1

while(vrh >= dno)do

srednji 🡨(dno+vrh)/2

if(K[srednji]<k)then

dno🡨srednji+1

else

vrh🡨srednji

end\_if

end\_while

if(vrh=-1)then

return -1

end\_if

if(K[vrh] = k)then

return vrh

else

return -1

end\_if

* Moguće je oblikovati i nešto drugačiji algoritam binarnog pretraživanja kod kojeg se u svakoj iteraciji, pri određivanju srednjeg elementa i polovljenja intervala, ne provjerava da li je traženi ključ pronađen.
* Tek na samom kraju izvršavanja algoritma, kada u nizu ostane samo jedan element, provjerava se da li je taj element traženi ključ. Ova druga varijanta binarnog pretraživanja je prikazana na slici u obliku funkcije BIN-PRETRAZI-M
* Iako funkcija BIN-PRETRAZI-M na prvi pogled izgleda manje efikasna u odnosu na funkciju BIN-PRETRAZI,

ipak u većini slučajeva je ustvari suprotno.

* Naime, funkcija BINPRETRAZI u svakoj iteraciji polovljenja intervala ima dva poređenja, za razliku od funkcije BIN-PRETRAZI-M, kod koje se u svakoj iteraciji polovljenja intervala obavlja jedna operacija usporedbe, te dodatna usporedba na kraju, kojom se provjerava da li je preostali element traženi ključ.
* Međutim, kod nizova sa većim brojem elemenata u većini slučajeva pretraživanja traženi ključ neće biti pronađen sve dok se niz polovljenjem ne reducira na svega nekoliko elemenata, a često i na samo jedan element.
* To znači, da će funkcija BINPRETRAZI u većini slučajeva izvršiti operacije usporedbi koje neće pronaći traženi ključ, što u određenoj mjeri ovu funkciju čini zapravo neefikasnijom u odnosu na funkciju BINPRETRAZI-M.

#### Efikasnost binarnog pretraživanja:

* Analizu efikasnosti binarnog pretraživanja možemo pojednostaviti tako da pretpostavimo da nizovi imaju dužinu jednaku potencijama broja 2, tj. da nizovi imaju dužine 2, 4, 8, 16, 32, itd.
* Nadalje, procjenu potrebnih operacija usporedbi ćemo napraviti za drugu varijantu binarnog

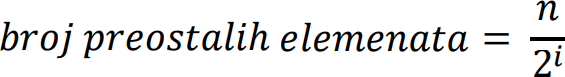
pretraživanja, koja je opisana funkcijom BIN-PRETRAZI-M. Na primjer, za niz dužine 8 će biti potrebna 3 polovljenja intervala, što znači da će biti potrebno ukupno 4 operacije usporedbi (3 operacije u koracima polovljenja i jedna završna operacija usporedbe).

* Za dužine nizova koje nisu jednake nijednoj potenciji broja 2, možemo procijeniti da će broj operacija usporedbi biti vrlo blizu broju operacija usporedbi za onaj niz čija je dužina najbliža dužini dotičnog niza i jednaka je potenciji broja 2.
* Na primjer, za niz dužine 1000, možemo procijeniti da će biti potrebno 10 operacija usporedbi da se niz polovi, sve dok u njemu ne ostane samo jedan element, te još jedna dodatna operacija usporedbe da se utvrdi da li je jedini preostali element traženi ključ
* Dakle, možemo procijeniti da će biti potrebno 11 operacija usporedbi da bi se pronašao neki ključ u nizu dužine 1000
* Općenito, za niz koji ima dužinu n jednaku n=2i, možemo zaključiti da će biti potrebna i+1 operacija usporedbi. Odnosno, možemo pisati da je ukupan broj operacija usporedbi

T(n): 𝑇 𝑛 = log2 𝑛 + 1

* Isto tako, umjesto analize kojom dobijamo broj potrebnih koraka usporedbe, možemo za određivanje vremenske složenosti binarnog pretraživanja procijeniti broj potrebnih koraka polovljenja intervala u ovisnosti od dužine n. Budući da se interval pretraživanja u svakom sljedećem koraku polovi, možemo pisati da poslije nekog i-tog koraka sljedeći izraz daje maksimalan broj elemenata koji su preostali u

intervalu pretraživanja



* Za obje prethodno opisane varijante binarnog pretraživanja možemo reći da algoritam sigurno završava kada preostane samo jedan element u intervalu pretraživanja.
* Dakle, nakon izjednačavanja izraza 8.2 sa 1, za maksimalan ukupan broj iteracija dobijamo

### 𝑖 = log2 𝑛

* Imajući u vidu da je stupanj rasta broja ostalih operacija (operacije dodjele, aritmetičke operacije) jednak stupnju rasta operacija usporedbi, zaključujemo da je vremenska složenost binarnog pretraživanja reda O(log n).

## Interpolacijsko pretraživanje - uvod

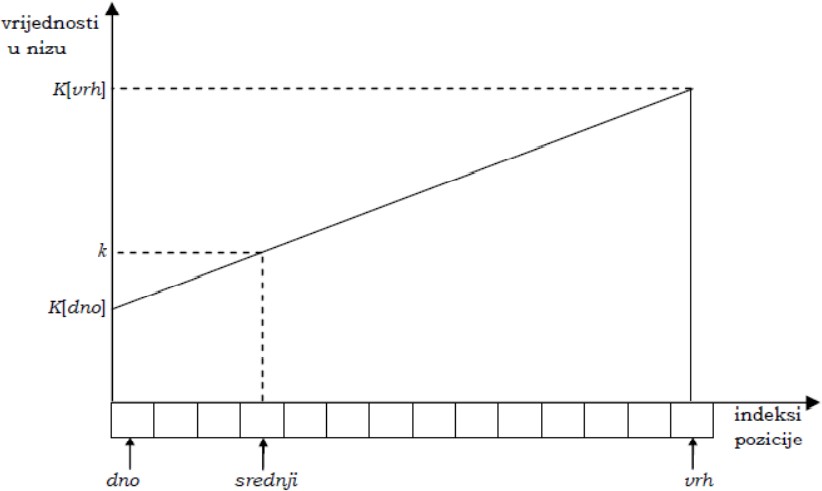
* Ako su vrijednosti ključeva u sortiranom nizu ravnomjerno raspoređene, onda se efikasnost algoritma binarnog pretraživanja može dodatno poboljšati korištenjem tzv. interpolacijskog pretraživanja
* Naime, umjesto da se u svakom koraku traženi ključ poredi sa elementom koji je pozicioniran na sredini

intervala na koji je usmjereno pretraživanje, opravdano je traženi ključ porediti sa elementom koji je bliži vrijednosti traženog ključa, koji se u općem slučaju ne nalazi na sredini intervala.

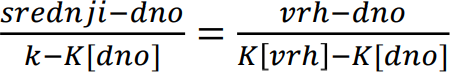
* Na primjer, ako je vrijednost traženog ključa k bliža vrijednosti ključa na donjoj granici intervala (K[dno]) od vrijednosti ključa na gornjoj granici intervala (K[vrh]), tada je efikasnije pretraživanje traženog ključa nastaviti u blizini donje granice intervala
* Prema tome, algoritam binarnog pretraživanja možemo modificirati tako da se pozicija polovljenja

intervala ne određuje kao sredina intervala, nego se ta pozicija interpolira na način kako je to ilustrirano

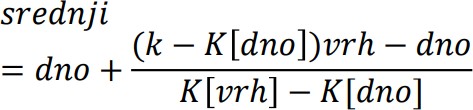
na slici



* Iz sličnosti truoglova se može napisati



* Slijedi



* Ako je traženi ključ jednako udaljen od vrijednosti ključeva na obje granice intervala, algoritam

interpolacijskog pretraživanja će prepoloviti interval na isti način kao i obično binarno pretraživanje.

* Algoritam interpolacijskog pretraživanja je isti kao i prethodno opisane varijante binarnog pretraživanja u obliku funkcija BINPRETRAZI i BINPRETRAZI-M, s jedinom razlikom da se pozicija središnjeg elementa srednji ne određuje kao što je to opisano naredbom u liniji 4
* Istaknimo još jednom da je interpolacijsko pretraživanje efikasnije u odnosu na binarno pretraživanje samo ako su vrijednosti ključeva u nizu ravnomjerno raspoređene.
* U suprotnom, efikasnost interpolacijskog pretraživanja je bitno smanjena. Isto tako, potrebno je uzeti u obzir da obično binarno pretraživanje ima jednostavniju cjelobrojnu aritmetiku, koja uključuje samo

indekse pozicija, dok interpolacijsko pretraživanje ima nešto složeniju aritmetiku, koja pored indeksa pozicija uključuje i vrijednosti ključeva, što u određenoj mjeri smanjuje vremensku efikasnost dotičnog algoritma