Elektrotehnički fakultet Univerziteta u Beogradu

ALGORITMI I STRUKTURE PODATAKA

Materijali za vežbu i pripremu ispita

verzija 2010-01-03

Moguće je da ovi materijali sadrže greške.
Zbog toga ih ne treba koristiti kao jedini izvor znanja.

Sve uočene greške prijaviti Đorđu Đurđeviću na adresu zorz@etf.rs

Dnevnik izmena dokumenta

Verzija 2010-01-03
Izmene u zadatku 12.10: preciziranje postavke i dopuna komentara

Verzija 2009-12-30
Uklonjen ključ 33 iz zadatka 12.3

Verzija 2009-12-17
Otklonjena greška u numerisanju strana.
Izmena u preglednoj tabeli algoritama za sortiranje (poglavlje 12) za najgori slučaj kod metode uređivanja primenom stabla binarnog pretraživanja.

Verzija 2009-10-21
Dodat zadatak 1.3
Dodat zadatak 8.5

Verzija 2009-10-07
Dopunjen zadatak 9.2

1. Uvod

1.1 Generatori slučajnih brojeva

Generator slučajnih brojeva je (u opštem slučaju) uređaj sposoban da proizvede sekvencu brojeva takvu da se u njoj ne može uočiti obrazac. Postoje dva kategorije generatora slučajnih brojeva:

1. Generator koji se bazira na merenju fizičke veličine za koju se očekuje da je slučajne prirode
2. Generator koji se bazira na primeni algoritma i celobrojne aritmetike

Zbog toga što se baziraju na primeni algoritma, generatori druge kategorije se preciznije nazivaju generatori **pseudoslučajnih** brojeva jer sekvence brojeva koje generišu zapravo nisu slučajne: postoji determinizam u načinu na koji se određuje sledeći broj u sekvenci pa se isti u principu može predvideti. Ipak, sve dok je nepoznat primenjen algoritam i tekuće stanje generatora (na osnovu kojeg se generiše sledeći broj u sekvenci, odnosno sledeće stanje), ovi generatori se naizgled ponašaju kao da daju sasvim slučajne sekvence brojeva. U nastavku će pažnja biti posvećena softverskim realizacijama generatora pseudoslučajnih brojeva.

Kvalitetan generator pseudoslučajnih brojeva treba da poseduje dve osobine:

1. dobre statističke karakteristike
2. što je moguće duže sekvence brojeva

Pod dobrim statističkim karakteristikama se podrazumeva da statističke analize dobijenih sekvenci ne smeju da pokažu nikakvu zakonitost. Raspodela generisanih brojeva, u idealnom slučaju, treba da bude **uniformna**, tj. verovatnoća generisanja svakog broja treba da bude podjadnaka i nezavisna od prethodno generisanih brojeva. Dužina generisane sekvence je ograničena prirodom postupka (mašinska reč ograničenog broja bita), pa je nakon generisanja poslednjeg broja u sekvenci prvi naredni broj zapravo prvi broj sekvence. Poznato je da se mnoge bibliotečke funkcije za generisanje slučajnih brojeva koje se isporučuju uz programski prevodilac (poput rand() koja se isporučuje uz programski jezik C) ne mogu svrstati u kategoriju kvalitetnih generatora.

Primene generatora slučajnih brojeva su raznolike:

- simulacija procesa (fizičkih ili drugih)
- kriptografija
- statističke analize
- zabava (video-igre, kockanje)
- sastavni su deo generatora pseudoslučajnih brojeva po drugim raspodelama (npr. eksponecijalnoj)

Linearan kongruentan generator (LKG) periode m

$X_{n+1} = (a \cdot X_n + c) \bmod m$

$0 < m$: moduo

$0 \leq a < m$: množilac

$0 \leq c < m$: inkrement

$0 \leq X_0 < m$: "klica" (eng. *seed*)

Donald Knuth je proučavao ovaj generator i došao do sledećih uslova da bi perioda generatora bila m:
- c i m su uzajamno prosti
- a-1 je deljivo svim prostim činiocima m
- a-1 je umnožak 4 ako je m umnožak 4

Predlažu se sledeći parovi brojeva *a* i *c* za 32-bitni računar ($m=2^{32}$):
a = 429493445, *c* = 907633385 (Knuth)
a = 1664525, *c* = 1013904223 (Numerical Recipes in C)

BBS generator

Ovaj generator je dobio ime prema svojim autorima: Lenore Blum, Manuel Blum, Michael Shub. Izraz koji se nalazi u osnovi generatora sekvence je:

$$X_{n+1} = (X_n)^2 \bmod M$$

gde je $M=p*q$, a p i q su prosti brojevi kongruentni sa 3 po modulu 4.

Primer: za vrednosti p=11, q=19, $X_0=3$, M=p*q=209, dobija se sledeća sekvenca:
=> $X_1=9$, $X_2=81$, $X_3=82$, $X_4=36$, $X_5=42$, $X_6=92$...

Kod ovog generatora, međutim, sledeći broj u sekvenci nije X_{n+1}, već se od X_{n+1} uzima nekoliko najmanje značajnih bitova ili odgovarajući bit parnosti. Zbog toga je neophodno više puta obaviti računanje datog izraza pre nego što se sakupi dovoljno bita da bi se formirao naredni broj u sekvenci. Zbog toga je sporiji od drugih generatora i nije pogodan za simulacije. Međutim ima dobre osobine koje su potrebne za kriptografiju.

Fibonačijev generator sa kašnjenjem (Lagged Fibonacci Generator)

Ideja za ovaj generator se zasniva na izrazu za rekurzivno određivanje Fibonačijevih brojeva: $F_n = F_{n-1} + F_{n-2}$.
Izraz za određivanje sledećeg broja u sekvenci je sledeći:

$$S_n = (S_{n-j} \otimes S_{n-k}) \bmod m, 0 < j < k$$

$\otimes$ je neka operacija (+ - · / xor)

Perioda ovog generatora zavisi od odabrane operacije. Vrednosti za j i k se ne mogu proizvoljno birati, već moraju da zadovolje određene uslove. Ovaj generator je problematičan zbog svoje inicijalizacije i osetljivosti na početne uslove.

Zadatak 1.1

Napisati funkciju na programskom jeziku C za generisanje slučajnog realnog broja u opsegu od 0 do 1 primenom LKG na 32-bitnom računaru.

```
unsigned long u;
unsigned long iurng(void)
{
      return u = 429493445 * u + 907633385;
}

void SetSeed(unsigned long seed)
{
      u = seed;
}

double Randomd(void)
{
          return( iurng() / 4294967296.);
}
```

Tražena funkcija je Randomd() koja vraća realan broj dvostruke preciznosti. Ova funkcija koristi funkciju iurng() koja vraća 32-bitnu celobrojnu vrednost primenom LKG sa vrednostima za a i c koje je predložio Knuth. Klica se postavlja pozivom funkcije SetSeed().

S obzirom na to da delilac u izrazu za određivanje povratne vrednosti funkcije Randomd() ima vrednost 2^{32}, bilo je neophodno koristiti realan broj dvostruke preciznosti jer jedino on ima dovoljno veliku mantisu da bi se vrednost 2^{32} tačno prikazala. Treba primetiti da u funkciji iurng() ne postoji operacija dohvatanja ostatka prilikom deljenja po modulu 2^{32} jer se prilikom smeštanja rezultata u promenljivu u automatski dobija ostatak prilikom deljenja po modulu 2^{32}.

Zadatak 1.2

Koristeći LKG generator implementirati funkciju `coin()` koja sa približno jednakom verovatnoćom vraća vrednosti 0 ili 1. Koristeći funkciju `coin()` implementirati funkciju `cube()` koja simulira bacanje kocke, tj. vraća vrednost od 1 do 6 sa približno podjednakim verovatnoćama.

```
void InitCoin(unsigned long seed)
{
    SetSeed(seed);
}

int coin()
{
    double i=Randomd();
    if(i<0.5) return 0;
    else return 1;
}

int cube()
{
    int i=coin();
    i = i*2 + coin();
    i = i*2 + coin();
    if(i >= 6) return cube();
    else return i+1;
}
```

Generator slučajnih brojeva se koristi da bi simulirao bacanje novčića kod kojeg je verovatnoća padanja na bilo koju stranu ista (tzv. "pošten" novčić). Ideja za realizaciju funkcije `cube()` jeste da se slučajno generiše svaki bit u binarnoj reprezentaciji broja dobijenog bacanjem kockice. Potrebno je 3 bita, pa se zbog toga novčić "baca" 3 puta. Naravno, može se dogoditi da se dobije nedozvoljena vrednost kao rezultat. Tada se, rekurzivnim pozivom, obavlja novo "bacanje" kockice. Inicijalizacija procesa "bacanja" novčića je obavljena u funkciji `InitCoin()` i svodi se na inicijalizaciju generatora slučajnih brojeva.

U test programu, u 60.000 poziva funkcije `cube()`, povratna vrednost iznosila je:

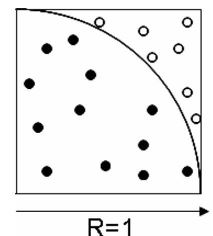
1 – 9.989 puta	4 – 9.972 puta
2 – 10.000 puta	5 – 9.924 puta
3 – 10.047 puta	6 – 10.068 puta

Zadatak 1.3

Napisati program koji izračunava približnu vrednost broja π primenom generatora slučajnih brojeva.

Rešenje:

Generator slučajnih brojeva će se koristiti za generisanje koordinata (X i Y) tačaka u ravni unutar kvadrata stranica dužine 1, u prvom kvadrantu. U graničnom slučaju, kada broj generisanih tačaka teži beskonačno, odnos ukupnog broja generisanih tačaka (sve se moraju nalaziti unutar pomenutog kvadrata!) T_U i broja generisanih tačaka koje se nalaze unutar četvrtine kružnice jediničnog poluprečnika T_K (popunjene crno na slici) teži ka odnosu površine kvadrata i četvrtine kružnice. Iz dobijenog odnosa se na sledeći način može odrediti tražena (približnu) vrednost π:



$$\frac{P_{\circ}}{P_{\square}} = \frac{\frac{1}{4}r^2\pi}{r^2} = \frac{1}{4}\pi \approx \frac{T_K}{T_U} \Rightarrow \pi \approx \frac{4T_K}{T_U}$$

```
PROGRAM RAND(output);
CONST MAX = 10000;
VAR i, j : longint;

FUNCTION pi(i : longint) : real;
VAR x, y,d : real;
    u_krugu, j : longint;
BEGIN
  u_krugu := 0;
  d := MAX;
  FOR j := 0 TO i DO
  BEGIN
    x := random(MAX)/d;
    y := random(MAX)/d;
    if sqr(x)+sqr(y)<=1.0 THEN
       u_krugu := u_krugu+1;
  END;
  pi := 4.0 * u_krugu / i
END;

BEGIN
  j := 1;
  FOR i := 1 to 7 DO
  BEGIN
    j:=j*10;
    writeln(output, pi(j));
  END
END.
```

Rezultati testa:

i	TP 7.0	VC++ 9
10	4	3.6
100	3.16	3.08
1000	3.152	3.084
10000	3.1552	3.1132
100000	3.14324	3.13984
1000000	3.141272	3.14014

Treba primetiti da vrednost veoma sporo konvergira ka stvarnoj vrednosti broja π.

1.2 *Run-length* kompresija

Kompresija po dužini sekvence (run-length) je veoma jednostavan metod kompresije koji se zasniva na pronalaženju sekvence istovetnih simbola u nizu podataka nad kojima se vrši kompresija. Osnovna ideja je da se u kodiranoj poruci takve sekvence predstave parom (broj_ponavljanja, simbol). Očigledno, ovakav pristup predstavlja kompresiju bez gubitaka, kod koje je niz podataka nakon dekompresije identičan ulaznom nizu podataka (na osnovu kojeg je izvršena kompresija). Takođe, ovakav pristup može da postigne dobre rezultate samo u slučaju da postoje dugačke sekvence identičnih simbola, što se može uočiti kod digitalnih slika (poznati formati za zapis digitalnih slika koji koriste ovu vrstu kompresije su BMP i ILBM).

Primer: AAABBBBBAAB ⇨ 3A5B2A1B

Problem ovog pristupa je postojanje dugačkih sekvenci uzastopno različitih simbola. Takve sekvence smanjuju stepen kompresije, a u teoretski najgorem slučaju bi mogle povećati dužinu poruke.

Primer: ABAB ⇨ 1A1B1A1B (gde se umesto originalnih 4 šalje 8 simbola)

Rešenje uočenog problema se svodi na poseban način tretiranja sekvenci različitih simbola: takve sekvence se direktno prepisuju u komprimovanu poruku. Usvaja se sledeća konvencija o obeležavanju dužine sekvence simbola:

- dužina sekvence istovetnih simbola se obeležava pozitivnim brojem
- dužina sekvence različitih uzastopnih simbola se obeležava negativnim brojem

Na osnovu toga, prilikom dekompresije poruke jednostavno se odlučuje da li naredni simbol treba ponoviti određen broj puta ili određen broj narednih simbola treba prepisati.

Obično se usvaja (mada nije neophodno) da jedan simbol ulaznog niza nad kojim se vrši kompresija odgovara jednom bajtu, kao i da je ceo broj kojim se obeležava dužina sekvence takođe predstavljen jednim bajtom. Time je dužina sekvence ograničena na najviše 128 znakova, nakon čega nova sekvenca mora da se inicira. S obzirom na to da je najveća vrednost pozitivnog broja predstavljenog jednim bajtom 127, deluje da postoji asimetrija u najvećoj mogućoj dužini sekvence istih i uzastopno različitih simbola. Ipak, ako se ima u vidu da se dužina sekvence od jednog izolovanog simbola predstavlja negativnim brojem, kao i to da dužina najkraće sekvence istih simbola mora biti 2, moguće je dužine sekvenci istih simbola označiti odgovarajućom vrednošću umanjenom za 1. Na primer, dužina sekvenca koja ima 2 simbola bi se označila vrednošću 1, a dužina sekvence od 128 simbola vrednošću 127. Time je zapravo postignuta simetrija u najvećim dužinama obe vrste sekvence. Vrednost 0 se u tom slučaju može rezervisati za oznaku kraja poruke.

Zadatak 1.2

Napisati na programskom jeziku C funkcije za vršenje run-length kompresije nad zadatim znakovnim nizovima.

```
#include <stdio.h>
#include <string.h>

void kompresija(char *poruka, char *izlaz)
{
char *pointer;
char c=*poruka;
int broj;

  while(c)
  {
     broj=0;
     if(c==*(++poruka))
     {
        while(*poruka && c==*poruka && broj<127)
        {
            broj++;
            poruka++;
        }
        *izlaz++=(char)broj;
        *izlaz++=c;
        c=*poruka;
     }
     else
     {
        pointer=izlaz++;
        while(*poruka && c!=*poruka && broj<127)
        {
           broj++; *izlaz++=c; c=*poruka++;
        }
        if(!*poruka)
        {
           broj++;
           *izlaz++=c;
           c=*poruka;
        }
        else poruka--;

        *pointer=(char)(-broj);
     }
  }
  *izlaz=0;
}
```

```
int dekompresija(char *poruka, char *izlaz)
{
char c = *poruka;
int i, n;

  while( *poruka )
  {
     if( *poruka > 0 )
     {
        n = *poruka++;
        if( ! *poruka )   return 0;

        for(i = 0; i <= n; i++)
        *izlaz++ = *poruka;
        poruka++;
     }
     else
     {
        n = -*poruka++;
        for(i = 0; i < n; i++)
        {
           if( ! *poruka ) return 0;
           *izlaz++ = *poruka++;
        }
     }
  }
  *izlaz = 0;
  return 1;
}

void main()
{
char ulaz[256];
char kompr[256];
char dekompr[256];

  printf("Unesite string za kompresiju: ");
  gets(ulaz);

  kompresija(ulaz, kompr);
  if( ! dekompresija(kompr, dekompr) )
     printf("Greska u dekompresiji\n");
  else
  {
     printf("Ulazna poruka: %s\n", ulaz);
     printf("Nakon dekompresije: %s\n", dekompr);

     if( ! strcmp(ulaz, dekompr) )
     {
         printf("Poruke su identicne\n");
         printf("Duzina originalne poruke je %d,"
                " a komprimovane poruke je %d\n", strlen(ulaz), strlen(kompr));
     }
     else
         printf("Greska u kompresiji/dekompresiji!\n");
  }
}
```

1.3 *Burrows-Wheeler* transformacija (BWT) [1994]

BWT je reverzibilna transformacija kojom se delimično može ublažiti problem nedostatka dužih sekvenci identičnih simbola. Prvenstveno se odnosi na transformaciju teksta, čijim je proučavanjem i nastala, ali se uspešno može primeniti i na ostale vrste podataka.

Osnovna ideja ove transformacije je da se ulazni niz simbola preuredi tako da se sekvence simbola koje se često pojavljuju grupišu. Zahvaljujući ovom grupisanju, moguće je izvršiti efikasniju kompresiju podataka.

Direktna transformacija

1. napraviti sve rotacije ulaznog niza simbola i smestiti u matricu (svaka vrsta matrice je jedna od rotacija)
2. izvršiti sortiranje matrice po vrstama
3. rezultat transformacije se nalazi u poslednjoj (desnoj) koloni matrice

Primer
Transformisati reč XBANANA primenom BWT.

Rešenje:
Reči koja se kodira najpre mora da se doda simbol za kraj stringa. U ovom primeru, kao znak za kraj stringa će se usvojiti simbol @. Dakle, ulazni niz koji treba kodirati je XBANANA@. Kasnije će biti objašnjeno kako se uvođenje simbola za kraj stringa može izbeći.

Reč koja se obrađuje	Rotacije	Sortiranje po vrstama	Rezultat
XBANANA@	XBANANA@ @XBANANA A@XBANAN NA@XBANA ANA@XBAN NANA@XBA ANANA@XB BANANA@X	ANANA@X**B** ANA@XBA**N** A@XBANA**N** BANANA@**X** NANA@XB**A** NA@XBAN**A** XBANANA**@** @XBANAN**A**	BNNXAA@A

U prethodnoj tabeli, u koloni "Sortiranje po vrstama" se može prepoznati ideja ove transformacije: u reči XBANANA se ispred slova N više puta pojavljuje slovo A, a takođe ispred slova A se više puta pojavljuje pojavljuje slovo N. U matrici nakon sortiranja se vidi da dve od tri reči koje počinju slovom A se završavaju sa N a obe reči koje počinju slovom N se završavaju slovom A. Zbog toga što su u matrici prisutne sve rotacije ulazne reči, poslednja kolona (rezultat transformacije) sadrži sve simbole koje sadrži i ulazna reč. Primetiti da je simbol za kraj stringa prisutan u rezultatu transformacije: dužina transformisane reči je povećana za jedan simbol u odnosu na polaznu reč.

Inverzna transformacija

1. polazi se od ulazne reči koja se postavlja kao poslednja (desna) kolona prazne kvadratne matrice
2. matrica se sortira po vrstama
3. ulazna reč se dodaje kao prva nepopunjena kolona matrice (posmatrano s desna)
4. koraci 2. i 3. se ponavljaju sve dok se ne popune sve kolone matrice
5. rezultat se nalazi u onoj vrsti matrice koja se završava simbolom za kraj stringa

Primer
Izvršiti inverznu BWT reči BNNXAA@A.

Rešenje:

Reč koja se obrađuje	Dodavanje kolone u matricu	Sortiranje	Dodavanje kolone u matricu	Sortiranje
BNNXAA@A	B N N X A A @ A	A A A B N N X @	BA NA NA XB AN AN @X A@	AN AN A@ BA NA NA XB @X

Dodavanje kolone u matricu	Sortiranje	Konačan izgled matrice	Rezultat
BAN NAN NA@ XBA ANA ANA @XB A@X	......	ANANA@XB ANA@XBAN A@XBANAN BANANA@X NANA@XBA NA@XBANA **XBANANA@** @XBANANA	**XBANANA@**

Kao što se može primetiti rekonstruisana matrica je identična matrici dobijenoj prilikom direktne transformacije, nakon sortiranja. Rezultat inverzne transformacije se nalazi u onoj vrsti matrice koja se završava simbolom za kraj stringa i predstavlja polazni niz simbola.

Simbol za kraj stringa je uveden da bi se objasnila transformacija. Nije neophodno (šta više, nepoželjno je) da u implementaciji ove transformacije postoji simbol za kraj stringa. Naime da bi neki simbol mogao da se koristi kao znak za kraj stringa on ne sme da pripada skupu simbola koji se mogu naći u ulaznom nizu. To znači da bi tabela simbola morala da se proširi dodatnim simbolom, čime bi mogla da se poveća dužina svakog simbola a time i dužina poruke. Umesto simbola za kraj poruke, prilikom direktne transformacije je dovoljno zapamtiti ceo broj koji označava na kojoj poziciji bi se taj simbol nalazio. Na primer, za transformisanu reč XBANANA, rezultat transformacije je BNNXAA@A, odnosno 7BNNXAAA.

2. Nizovi i matrice

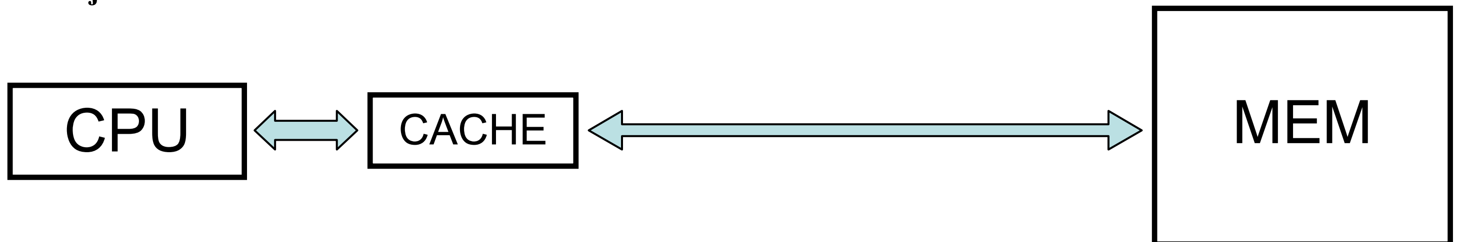
Zadatak 2.1

Potrebno je izvršiti sumiranje svih elemenata matrice M[32,16] čiji svaki element staje u jednu memorijsku reč. Matrica je u memoriji smeštena po vrstama. Neka je keš memorija za podatke kapaciteta 256 reči podeljena na 16 blokova dužine 16 reči. Keš memorija je direktno mapirana, a matrica je smeštena tako da se prvi element matrice mapira u prvu reč prvog bloka. Podaci nisu pre početka u keš memoriji. Koliki je faktor uspešnosti kod sledeća dva primera?

SUM1
```
sum=0
  for i=1 to 32 do
          for j=1 to 16 do
                  sum=sum+ M[i,j]
          end_for
  end_for
```

SUM2
```
sum=0
  for j=1 to 16 do
          for i=1 to 32 do
                  sum=sum+ M[i,j]
          end_for
  end_for
```

Rešenje:



Keš (cache) memorija
- veoma brza memorija
- veoma mali kapacitet (u odnosu na Operativnu Memoriju - MEM)
- hardversko rešenje (radi bez softverske intervencije)

Uloga
- smanjiti komunikaciju CPU↔MEM
- približiti podatke CPU (kraće vreme dohvatanja podataka)

Opšti scenario:
1. CPU traži podatak na adresi A
2. CACHE presreće zahtev i proverava da li već ima taj podatak.
   - 2.1. Ako ga ima – vraća ga.
   - 2.2. Ako ne, prosleđuje zahtev do MEM, ali ujedno traži i onoliko podataka sa nekoliko narednih adresa da bi popunila jedan svoj blok (A, A+1, A+2, A+3, ...)
     - 2.2.1. MEM odgovara na zahtev i šalje tražene podatke jedan za drugim
     - 2.2.2. CACHE prihvata podatke i smešta ih u predviđen blok. Paralelno sa prihvatanjem podataka, vraća CPU traženi podatak čim postane raspoloživ
3. CPU traži podatak sa adrese A+1
4. CACHE presreće zahtev, detektuje da ima taj podatak i vraća ga CPU

U slučaju SUM1, podacima se pristupa u redosledu u kojem su smešteni u memoriju. Keš memorija je inicijalno prazna, tako da će prvi zahtev mikroprocesora (M[1,1]) biti prosleđen do operativne memorije. Tom prilikom će iz operativne memorije u keš memoriju biti dovučeni svi podaci prve vrste matrice. Sledeći zahtev mikroprocesora (M[1,2]) će biti vraćen od strane keš-memorije jer je u prethodnom pristupu dovučen iz operativne memorije. Kada mikroprocesor bude pristupio podatku M[2,1], ovaj se neće nalaziti u keš memoriji,

pa će se prema prethodnom scenariju svi podaci druge vrste matrice dovući u keš memoriju. Prilikom dovlačenja 17. vrste iz matrice, ona će u keš memoriji biti smeštena u bloku gde je do tada bila smeštena prva vrsta matrice. Međutim, prva vrsta matrice je već obrađena i neće joj se pristupati do kraja obrade, tako da to ne remeti efikasnost izvršenja programa. S obzirom na to da matrica ima 32 vrste, dogodiće se 32 promašaja od ukupno 512 pristupa pristupa keš memoriji. **Zbog toga je faktor uspešnosti q = (512-32)/512 = 0.938**

U slučaju SUM2, pristup elementima matrice se vrši po kolonama. Nakon dovlačenja prvih 16 vrsta matrice, 17. vrsta matrice će biti smeštena u istom bloku gde je bila smeštena 1. vrsta. Kada se bude pristupalo 2. koloni matrice, biće neophodno ponovno dovlačenje svih vrsta matrice jer se one neće nalaziti u keš memoriji. Zbog toga će svaki zahtev mikroprocesora naići na promašaj u keš memoriji, odnosno svi podaci će biti dovučeni iz operativne memorije. **Zbog toga je faktor uspešnosti q = 0**.

Zadatak 1.2

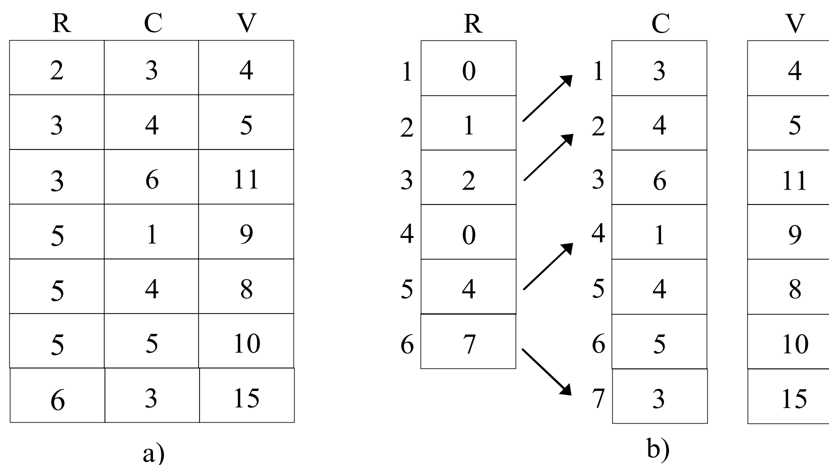
Prikazati vektorsku prezentaciju retke matrice sa slike:

a) sa jednim vektorom zapisa od po tri polja,

b) sa tri posebna vektora.

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 11 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 8 & 10 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Rešenje:



R	C	V
2	3	4
3	4	5
3	6	11
5	1	9
5	4	8
5	5	10
6	3	15

a)

	R
1	0
2	1
3	2
4	0
5	4
6	7

	C	V
1	3	4
2	4	5
3	6	11
4	1	9
5	4	8
6	5	10
7	3	15

b)

Zadatak 1.3

Šta se dobija na izlaznoj jedinici računara nakon izvršenja sledećeg FORTRAN programa:

```
PROGRAM PRIMER
  DIMENSION K(3,4)
  DO I=1,3
       DO J=1,4
            K(I,J)=I+J
       ENDDO
  ENDDO
  CALL MISSMATCH(K)
PRINT *, ((K(I,J), J=1,4), I=1,3)
END

SUBROUTINE MISSMATCH(K)
  DIMENSION K(1,1)
  K(2,3)=0
  RETURN
END
```

Rešenje:
Adresa elementa:
$A_{i,j} = A_{l1,l2} + ((j - l_2)(u_1 - l_1 + 1) + i - l_1)s$
$i=2, j=3, l_2=1, l_1=1, u_1=1$
$A_{2,3}=A_{1,1}+((3-1)(1-1+1)+2-1)s$

$$K = \begin{bmatrix} 2 & 0 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

Zadatak 1.4

Objasniti postupak smeštanja i izvesti adresnu funkciju pri pristupu proizvoljnom elementu gornje trougaone matrice smeštene po kolonama. Smatrati da se jedan element matrice smešta u tačno jednu memorijsku reč.

Rešenje:

$$A_{ij} = A_{11} + (j(j-1)/2 + i - 1)*s$$

Objašnjenje: treba uočiti element u *i*-toj vrsti i *j*-toj koloni. Ukupan broj elemenata u matrici pre kolone u kojoj se nalazi uočen element iznosi $\sum_{k=1}^{j-1} k = \frac{j(j-1)}{2}$. U koloni u kojoj se nalazi uočen element ima ukupno *i*-1 elemenata pre uočenog elementa. *s* je veličina memorijske reči.

Zadatak 1.5

Tridijagonalna matrica je matrica reda nxn, gde je A[i,j] = 0, ako je |i-j|>1.

a) Koliki je maksimalan broj nenultih elemenata ?
b) Ako se matrica linearizuje po vrstama, izvesti adresnu funkciju.

Rešenje:

a) $n_{nz} = 3*n - 2$
b) $A_{ij} = A_{11} + (2*i + j - 3)*s$, ako je $|i-j| \leq 1$

Objašnjenje: jedan od načina da se izračuna adresna funkcija u ovom slučaju je da se doda fiktivni element u prvu vrstu (koji bi bio u nepostojećoj koloni 0). Kao i u prethodnom zadatku, treba uočiti neki element u *i*-toj vrsti i *j*-toj koloni. Zahvaljujući fiktivnom elementu, postignuto je da svaka vrsta matrice, pre vrste u kojoj se nalazi uočen element ima tačno 3 elementa. Taj član bi u izrazu za adresnu funkciju bio 3*(*i*-1). U *i*-toj vrsti, pre uočenog elementa postoji *j*-*i*+1 element, jer je |i-j| ≤ 1. Na kraju, treba oduzeti 1, zbog fiktivnog elementa.
Ukupan izraz je : 3*(i-1)+j-i+1-1, što se svodi na dato rešenje.

j

i					
	a_{11}	a_{12}			
	a_{21}	a_{22}	a_{23}		
i		a_{32}	a_{33}	a_{34}	
			a_{43}	a_{44}	a_{45}
				a_{54}	a_{55}

Savet: prilikom određivanja adresne funkcije, nacrtati tabelu koja prikazuje redosled u kojem se elementi matrice smeštaju u memoriju i pomeraj u odnosu na adresu prvog elementa (tj. indeks). Koristiti tabelu za proveru

a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	...
0	1	2	3	4	5	6	...

Ako se uoči da adresna funkcija linearno zavisi od broja vrste i broja kolone, tj. pomeraj=A*vrsta + B*kolona + C, zadatak se može rešiti i rešavanjem sistema linearnih jednačina. Pomeraj u odnosu na prvi element se određuje tražeći koeficijente A, B i C. Pažljivim izborom tri elementa poznatih pomeraja dobija se sistem linearnih jednačina. Na primer:
A+B+C=0 (element a_{11})
2A+3B+C=4 (element a_{23})
3A+2B+C=5 (element a_{32})
Rešavanjem sistema od ove tri jednačine dobijaju se koeficijenti A=2, B=1, C=-3, što daje ranije pronađeno rešenje.
Napomena: u ovom postupku neophodno je voditi računa o izboru elemenata za koje će odgovarajuće jednačine biti linearno nezavisne. Za kakav izbor elemenata će jednačine biti linearno zavisne?

1. Zadatak za vežbu: (Generalizacija gornjeg primera): Data je matrica A_{ij} - to je matrica reda nxn koja ima popunjenu po a-1 dijagonalu ispod i iznad glavne dijagonale. Naći adresnu funkciju ako se prvo smešta krajnje donja dijagonala, pa dijagonala iznad nje, itd.

2. Zadatak za vežbu: Data je matrica reda nxn, gde su svu elementi A[i,j] jednaki 0, osim onih za koje važi n+2 ≥ i+j ≥ n. Koliki je maksimalan broj nenultih elemenata? Ako se matrica linearizuje po vrstama uzimajući prvo elemente sa manjim rednim brojem kolona, izvesti adresnu funkciju.

Zadatak 1.6 (IR2ASP - Januar 2007)

Usvojiti efikasnu implementaciju kvadratne matrice realnih brojeva za koju se zna da je simetrična u odnosu na glavnu dijagonalu, a zatim napisati kompletne potprograme (na jeziku C ili C++) za efikasno čitanje proizvoljnog elementa, upisivanje proizvoljne vrednosti elementa i određivanje broja nenultih elemenata matrice. Obavezno komentarisati program.

Rešenje

Data matrica se najjednostavnije i najefikasnije može predstaviti u vidu trougaone matrice. Pretpostavka: posmatra se gornja trougaona matrica, elementi matrice su smešteni po vrstama. Adresna funkcija je $A_{i,j} = A_{0,0} + i \cdot n + j - \frac{i \cdot (i+1)}{2}$ (jer je 0 indeks prvog element niza u C/C++). [*Uporediti sa donjom trougaonom matricom*]. Klasa u jeziku C++ koja realizuje traženu funkcionalnost je prikazana ispod. Napomena: deo koda koji predstavlja rešenje zadatka je obojen sivo.

```
// MATRICA.H
#ifndef MATRICA_H_
#define MATRICA_H_
#include <iostream>
using namespace std;
class Matrica
{
protected:
	double		*m_matrica;
	unsigned int m_dim;
	unsigned int napravi(unsigned int dim);
	void kopiraj(const Matrica &m);
public:
	Matrica(unsigned int dim);
	Matrica(const Matrica &m) { kopiraj(m); }
	~Matrica()	{ delete []m_matrica; }

	Matrica &operator=(const Matrica &m)
	{
		if( this != &m )
		{
			delete []m_matrica;
			kopiraj(m);
		}
		return *this;
	}
	//--------------------------

	double & dohvati(unsigned int v, unsigned int k);
	const double & dohvati(unsigned int v, unsigned int k) const
	{
		return const_cast<Matrica *>(this)->dohvati(v,k);
	}

	unsigned int brojNenultih() const;


	friend ostream & operator<<(ostream &o, const Matrica &m);
};
```

```
// MATRICA.CPP
#include "Matrica.h"
unsigned int Matrica::napravi(unsigned int dim)
{
	m_dim = dim;
	unsigned int duz = m_dim * (m_dim+1) / 2;
	m_matrica = new double[duz];
	return duz;
}
void Matrica::kopiraj(const Matrica &m)
{
  unsigned int duz = napravi(m.m_dim);
  for(unsigned int i = 0; i < duz; m_matrica[i] = m.m_matrica[i++]);
}
Matrica::Matrica(unsigned int dim)
{
	unsigned int duz = napravi(dim);
	for(unsigned int i = 0; i < duz; m_matrica[i++] = 0);
}
double & Matrica::dohvati(unsigned int v, unsigned int k)
{
unsigned int vrsta, kolona;
	if( v > k )  vrsta = k, kolona = v;
	else         vrsta = v, kolona = k;
	unsigned int pomeraj = vrsta*m_dim + kolona - vrsta*(vrsta+1)/2;
	return m_matrica[pomeraj];
}
unsigned int Matrica::brojNenultih() const
{
unsigned int broj = 0;
	for(unsigned int i = 0; i < m_dim; i++)
		for(unsigned int j = i; j < m_dim; j++)
			if( dohvati(i, j) != 0 )
				broj += i == j ? 1 : 2;
	return broj;
}

ostream & operator<<(ostream &o, const Matrica &m)
{
	for(unsigned int i = 0; i < m.m_dim; i++ )
	{
		for( unsigned j = 0; j < m.m_dim; j++ )
			o << m.dohvati(i, j) << " ";
		o << endl;
	}
	return o;
}
// GLAVNI PROGRAM
#include "Matrica.h"
#include <iostream>
using namespace std;

void main()
{
Matrica mat(5);
	for(unsigned int i = 0; i < 5; i++)
		mat.dohvati(i,0) = i+1;
	cout << mat << mat.brojNenultih() << endl;
}

Izlaz nakon pokretanja datog programa:

1 2 3 4 5
2 0 0 0 0
3 0 0 0 0
4 0 0 0 0
5 0 0 0 0
```

2. ULANČANE LISTE

Zadatak 2.1

Realizovati operaciju INVERT(list) koja menja poredak čvorova u listi na čiji prvi čvor ukazuje pokazivač *list*, tako da prvi čvor postane poslednji, drugi pretposlednji, itd.

Rešenje:

```
INVERT(list)
p = list
q = nil
while (p ≠ nil) do
    r = q
    q = p
    p = next(p)
    next(q) = r
end_while
list = q
```

Zadatak 2.2

Realizovati operaciju CONCATENATE(list1,list2) koja nadovezuje listu na koju pokazuje *list2* na listu ukazanu sa *list1*:

a) ako su liste jednostruko ulančane,
b) sa kružno ulančanim listama.

Rešenje:

```
a) CONCATENATE(list1, list2)
if list1 = nil then
  list1 = list2
  return
else if list2 = nil then
  return
end_if
p = list1
while (next(p) ≠ nil) do
  p = next(p)
end_while
next(p) = list2
```

```
b) CONCATENATE-C(list1, list2)
if (list1 = nil) then
  list1 = list2
  return
else if (list 2 = nil) then
  return
end_if
p = next(list1)
next(list1) = next(list2)
next(list2) = p
list1 = list2
```

Zadatak 2.3

Napisati operacije inicijalizacije liste slobodnih ulaza, alokacije i dealokacije čvora u slučaju vektorske implementacije ulančanih lista:

Rešenje:

```
INIT
avail = 1
for i = 1 to n – 1 do
    L[i].next = i +1
end_for
L[n].next = 0


GETNODE
if (avail = 0) then
    ERROR(Nema prostora)
end_if
p = avail
avail = L[avail].next
return p


FREENODE(p)
L[p].next = avail
avail = p
```

Zadatak 2.4

Neka su dva skupa predstavljena **uređenim** jednostruko ulančanim listama **sa zaglavljima** na koja ukazuju pokazivači A i B. Realizovati funkciju koja vraća pokazivač na zaglavlje liste koja predstavlja presek ova dva skupa.

Rešenje:

```
INTERSECTION(A, B)
r = c= GETNODE
p = next(A)
q = next(B)
while (p ≠ nil) and (q ≠ nil) do
        if (info(p) = info(q)) then
                next(c) = GETNODE
                c = next(c)
                info(c) = info(p)
                p = next(p)
                q = next(q)
        else if (info(p) < info(q)) then
                        p = next(p)
                else
                        q = next(q)
                end_if
        end_if
end_while
return r
```

Zadatak 2.5

Neka su dva polinoma sa jednom nezavisnom promenljivom predstavljena uređenim jednostruko ulančanim kružnim listama sa zaglavljima na koja ukazuju pokazivači p1 i p2. Realizovati funkciju koja vraća pokazivač na zaglavlje liste koja predstavlja zbir ova dva polinoma.

Napomena: informacioni sadržaj zaglavlja, koji odgovara polju eksponenta kod ostalih čvorova liste, ima vrednost -1.

Rešenje:

```
POLY-ADD(p1, p2)
r = GETNODE
next(r) = r
exp(r) = -1
t = r
p = next(p1)
q = next(p2)
while (p ≠ p1) or (q ≠ p2) do
    if (exp(p) = exp(q)) then
        if (c(p) + c(q) ≠ 0) then
            INSERT-AFTER(t, c(p) + c(q), exp(p))
            t = next(t)
        end_if
        p = next(p)
        q = next(q)
    else if (exp(p) < exp(q)) then
            INSERT-AFTER(t, c(q), exp(q))
            q = next(q)
            t = next(t)
        else
            INSERT-AFTER(t, c(p), exp(p))
            p = next(p)
            t = next(t)
    end_if
end_while
return r
```

Zadatak za vežbu

Na programskom jeziku C napisati funkcije za rad sa jednostruko ulančanim listama sa generičkim sadržajem. Obezbediti mogućnost automatskog brisanja generičkog sadržaja prilikom brisanja liste. Napisati na programskom jeziku C glavni program koji demonstrira korišćenje ovih funkcija.

Rešenje

```
/* lista.h : jednostruko ulancana lista */

typedef struct ElListe
{
      void  *podatak;
      struct ElListe    *sledeci;
} ElementListe;

typedef struct lista
{
      ElementListe *prvi, *poslednji;
      void (*brisi)(void *);
      int br_elem;
} Lista;


typedef struct listIterator
{
      Lista       *lista;
      ElementListe      *tekuci, *prethodni;
} IteratorListe;


/* pravljenje, brisanje i praznjenje liste */
Lista *napraviListu( void (*brisi)(void *) );
void obrisiListu(Lista *l);
void isprazniListu( Lista *l );


/* dodavanje i uklanjanje */
int dodajUListu(Lista *l, void *podatak );
int ukloniElementBr(Lista *l, int br);
int ukloniElement(Lista *l, ElementListe *el);


/* razne operacije sa listom */
int brojElemenata(Lista *l);

/* funkcije iteratora */
void inicijalizujIterator(Lista *l, IteratorListe *i);
void naPocetak(IteratorListe *i);
void sledeciElement(IteratorListe *i);
int krajListe(IteratorListe *i);
```

```
#include "lista.h"
#include <stdlib.h>

/* pravljenje i brisanje liste */
Lista *napraviListu( void (*brisi)(void *) ) {
Lista *l = calloc(1, sizeof(Lista) );

      if( ! l )   return 0;
      l->brisi = brisi;
      return l;
}

void obrisiListu(Lista *l) {
      if( l ) {
            isprazniListu(l);
            free(l);
      }
}

void isprazniListu( Lista *l ) {
      if( l ) {
      ElementListe *el = l->prvi;

            while( el )  {
            ElementListe *stari = el;
                  el = el->sledeci;

                  if( l->brisi )
                        l->brisi(stari->podatak);

                  free(stari);
            }

            l->prvi = l->poslednji = 0;
            l->br_elem = 0;
      }
}


/* dodavanje i uklanjanje */
/* vraca broj elemenata u listi ili 0 ako je doslo do greske */
int dodajUListu(Lista *l, void *podatak )
{
      if( l && podatak ) {
      ElementListe *noviEl = calloc(1, sizeof(ElementListe) );

            if( ! noviEl )    return 0;
            noviEl->podatak = podatak;
            if( l->prvi )
                  l->poslednji->sledeci = noviEl;
            else
                  l->prvi = noviEl;

            l->poslednji = noviEl;
            return ++l->br_elem;
      }

      return 0;
}
```

```
/* uklanja element na koga ukazuje iterator, prototip ne postoji
   u lista.h fajlu, pa nije "javno" dostupna funkcija */
void ukloniElementIzListe(Lista *l, IteratorListe *il) {
      if( ! il->prethodni ) l->prvi = il->tekuci->sledeci;
      else                  il->prethodni->sledeci = il->tekuci->sledeci;

      if( il->tekuci == l->poslednji )
            l->poslednji = il->prethodni;

      if( l->brisi )    l->brisi( il->tekuci->podatak );

      free( il->tekuci );
}


/* uklanja element pod zadatim rednim brojem */
int ukloniElementBr(Lista *l, int br) {
      if( l && br > 0 && l->br_elem <= br ) {
      IteratorListe il;
      int i;

            inicijalizujIterator(l, &il);

            for( i = 0, naPocetak(&il);
                 i < br-1 && ! krajListe(&il);
                 i++, sledeciElement(&il) );

            ukloniElementIzListe( l, &il );
            return 1;
      }
      return 0;
}

/* uklanja zadati element iz liste */
int ukloniElement(Lista *l, ElementListe *el) {
      if( ! l )     return 0;
      else  {
      IteratorListe il;

         inicijalizujIterator(l, &il);
         for( naPocetak(&il); ! krajListe(&il) && il.tekuci != el; sledeciElement(&il));
         if( ! krajListe(&il) ) {
              ukloniElementIzListe(l, &il);
              return 1;
         }
     }
     return 0;
}

/* razne operacije sa listom */
int brojElemenata(Lista *l) {
      if( l )   return l->br_elem;
      return 0;
}

/* funkcije iteratora */
void inicijalizujIterator(Lista *l, IteratorListe *i) {
      i->lista = l;
      i->prethodni = i->tekuci = 0;
}
```

```
void naPocetak(IteratorListe *i) {
	i->prethodni = 0;
	i->tekuci = i->lista->prvi;
}

void sledeciElement(IteratorListe *i) {
	i->prethodni = i->tekuci;
	i->tekuci = i->tekuci->sledeci;
}

int krajListe(IteratorListe *i) {
	if( i->tekuci )   return 0;
	else              return 1;
}
```

```
#include "lista.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
	char *prezime, *ime;
	char jmbg[14];
} Osoba;

void brisiOsobu(void *o) {
Osoba *osoba = (Osoba *)o;

	if( osoba ) {
		free( osoba->prezime );
		free( osoba->ime );
		free( osoba );
	}
}

Osoba *citajOsobu()
{
char niz[128];
Osoba *o = calloc(1, sizeof(Osoba) );

	if( ! o )  return 0;

	printf("Unesite prezime osobe:");
	scanf("%s", niz);
	o->prezime = malloc( strlen(niz)+1 );
	if( ! o->prezime ) {
		brisiOsobu(o);
		return 0;
	}
	strcpy(o->prezime, niz);

	printf("Unesite ime osobe:");
	scanf("%s", niz);
	o->ime = malloc( strlen(niz)+1 );
	if( ! o->ime ) {
		brisiOsobu(o);
		return 0;
	}
	strcpy(o->ime, niz);
```

```
      printf("Unesite JMBG osobe:");
      scanf("%s", o->jmbg);
      return o;
}

void main() {
Lista *lista = napraviListu( brisiOsobu );
char odgovor = 'd';
IteratorListe il;

    if( ! lista )  {
        printf("Greska u alokaciji memorije!\nProgram se prekida.\n");
        exit(0);
    }

    do {
        printf("Unesite podatke o osobi:\n");
        if( ! dodajUListu(lista, citajOsobu() )) {
            printf("Doslo je do greske prilikom dodavanja osobe u listu."
                   "Unos se prekida.\n");
            break;
        }

        printf("Jos? (d/n) ");
        scanf("\n%c", &odgovor);
       } while( odgovor != 'n' && odgovor != 'N' );

      printf("Uneta lista:\n");
      inicijalizujIterator(lista, &il);
      for( naPocetak(&il); ! krajListe(&il); sledeciElement(&il) ) {
      Osoba *o = (Osoba *)il.tekuci->podatak;
            printf("%s %s %s\n", o->prezime, o->ime, o->jmbg);
      }
      obrisiListu(lista);
}
```

3. STEKOVI

Zadatak 3.1

Razmatra se sekvencijalna implementacija više stekova. Napisati funkcije brisanja elementa sa steka S_i i umetanje vrednosti x na stek S_i.

Rešenje:

```
POP-M(i)
    if (top[i] = b[i]) then
        return underflow
    else
        x = V[top[i]]
        top[i] = top[i] – 1
        return x
    end_if


PUSH-M(i, x)
    if (top[i] = b[i + 1]) then
        ERROR(Overflow)
    else
        top[i] = top[i] + 1
        V[top[i]] = x
    end_if
```

Zadatak 3.2

Izvršiti konverziju infiksnog izraza A+B*C-(D+E-F↑G↑H)*(I+(J-K)*L)/M u postfiksni

Rešenje:

next	*S*	*postfix*	*rank*
A		A	1
+	+	A	1
B	+	AB	2
*	+*	AB	2
C	+*	ABC	3
-	-	ABC*+	1
(	-(	ABC*+	1
D	-(	ABC*+D	2
+	-(+	ABC*+D	2
E	-(+	ABC*+DE	3
-	-(-	ABC*+DE+	2
F	-(-	ABC*+DE+F	3
↑	-(-↑	ABC*+DE+F	3
G	-(-↑	ABC*+DE+FG	4
↑	-(-↑↑	ABC*+DE+FG	4
H	-(-↑↑	ABC*+DE+FGH	5
)	-	ABC*+DE+FGH↑↑-	2
*	-*	ABC*+DE+FGH↑↑-	2
(	-*(	ABC*+DE+FGH↑↑-	2
I	-*(	ABC*+DE+FGH↑↑-I	3
+	-*(+	ABC*+DE+FGH↑↑-I	3
(	-*(+(	ABC*+DE+FGH↑↑-I	3
J	-*(+(	ABC*+DE+FGH↑↑-IJ	4
-	-*(+(-	ABC*+DE+FGH↑↑-IJ	4
K	-*(+(-	ABC*+DE+FGH↑↑-IJK	5
)	-*(+	ABC*+DE+FGH↑↑-IJK-	4
*	-*(+*	ABC*+DE+FGH↑↑-IJK-	4
L	-*(+*	ABC*+DE+FGH↑↑-IJK-L	5
)	-*	ABC*+DE+FGH↑↑-IJK-L*+	3
/	-/	ABC*+DE+FGH↑↑-IJK-L*+*	2
M	-/	ABC*+DE+FGH↑↑-IJK-L*+*M	3
		ABC*+DE+FGH↑↑-IJK-L*+*M/-	1

operator	*ipr*	*spr*	R
+ , -	2	2	-1
* , /	3	3	-1
↑	5	4	-1
(	6	0	-
)	1	-	-

Zadatak 3.3

Postoje tri štapa A, B i C, a na štapu A se nalazi *n* različitih diskova poređanih po veličini tako da je disk najvećeg poluprečnika na dnu, a najmanji na vrhu. Potrebno je da se diskovi prebace sa štapa A na štap C koristeći štap B kao pomoćni. Pritom se sa štapa može uklanjati samo disk sa vrha, a ni u jednom trenutku se ne dozvoljava da disk manjeg poluprečnika bude ispod diska većeg poluprečnika na bilo kom štapu. Rešiti problem:

a) pomoću rekurzije
b) bez upotrebe rekurzije

Rešenje:

a)

```
HANOI-R(n, from, to, aux)
if (n = 1) then
        PRINT(Prenesi disk 1 sa from na to)
        return
end_if
HANOI-R(n - 1, from, aux, to)
PRINT(Prenesi disk n sa from na to)
HANOI-R(n - 1, aux, to, from)
return
```

b)

```
HANOI-NR3(n, from, to, aux)
INIT-STACK(S, v)
loop
        while (n ≠ 1) do
           ar ← (n, from, to, aux)
           PUSH(S, ar)
           n = n - 1
           to ↔ aux
        end_while
        PRINT(Prenesi disk 1 sa from na to)
        if (STACK-EMPTY(S)) then
                return
        end_if
        ar = POP(S)
        (n, from, to, aux) ← ar
        PRINT(Prenesi disk n sa from na to)
        n = n – 1
        from ↔ aux
end_loop
```

4. REDOVI

Zadatak 4.1

Sa redom implementiranim kao ulančana lista na koju pokazuje pokazivač q, realizovati operaciju umetanja elementa sa vrednošću x i operaciju brisanja iz nepraznog reda.

Rešenje:

```
INSERT-L(q, x)
p = GETNODE
info(p) = x
next(p) = nil
if (rear[q] = nil) then
        q = p
else
        next(rear[q]) = p
end_if
rear[q] = p
```

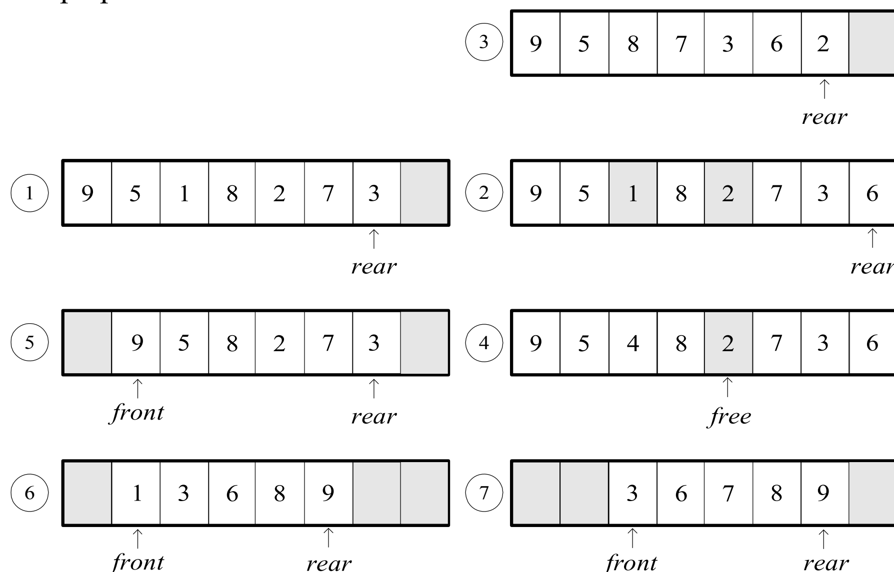
```
DELETE-L(q)
if (q = nil) then
        return underflow
else
        p = q
        x = info(p)
        q = next(p)
    if (q = nil) then
                rear[q] = nil
    end_if
        FREENODE(p)
        return x
end_if
```

Zadatak 4.2

Razmatra se prioritetni red. Objasniti moguće načine realizacije operacije brisanja (elementa najmanjeg prioriteta), ako je red implementiran kao vektor.

Rešenje:

1. Brisanje se najbrže obavlja ako se samo mesto uklonjenog elementa posebno markira kao nevažeći element koji se pri narednim pretraživanjima na najmanji element ignoriše.
2. S obzirom da su markirane pozicije praktično slobodne, gornji način može da se modifikuje tako što se, umesto iza *rear*, dozvoljava umetanje na poziciju prvog nevažećeg elementa (ako takav postoji).
3. Da bi se izbegla pojava markiranih pozicija, pri brisanju se mogu pomeriti svi elementi iza onog koji se briše za jednu poziciju nadole i dekrementirati *rear*.
4. Moguće je održavati red rastuće uređenim tako da fizički poredak elemenata odgovara njihovom logičkom poretku po prioritetu.



Zadatak 4.3

Realizovati operaciju umetanja elementa u prioritetni red implementiran kao uređena jednostruko ulančana lista. Mogu se koristiti već realizovane funkcije PUSH_L(S,x) – ubacivanje novog elementa na početak liste i INSERT_AFTER(q,x) – umetanje u listu novog čvora sa sadržajem x iza čvora čija je adresa q.

Rešenje:

```
PQ-INSERT(pq, x)
q = nil
p = pq
while (p ≠nil) and (x ≥ info(p)) do
        q = p
        p = next(p)
end_while
if (q = nil) then
        PUSH-L(pq, x)
else
        INSERT-AFTER(q, x)
end_if
```

Zadatak 4.4

Realizovati red koristeći dva steka implementirana u jednom vektoru.

```
INIT(S1,S2)
  ALLOCATE(V[1:n])
  b2 = b1=top1=top2=0
  return
```

```
INSERT(x)
  PUSH(S1, x)
  b2 = top1
  rear = top1
  return
```

```
DELETE(x)
  if(top1=top2) then
        return underflow
  end_if
  x=POP(S2)
  b1 = top2
  front = top2
  return x
```

Zadatak za vežbu

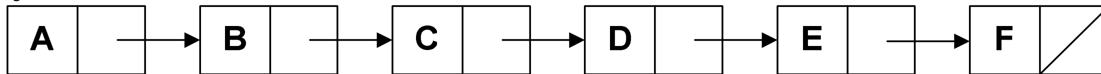
Realizovati stek koristeći dva reda implementirana u jednom vektoru.

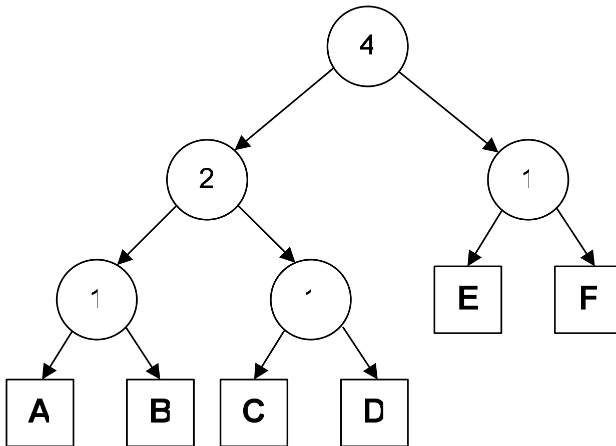
5. STABLA

Zadatak 5.1

Predstaviti jednostruko ulančanu listu pomoću binarnog stabla i realizovati algoritam za nalaženje k-tog elementa u datoj listi.

Rešenje:





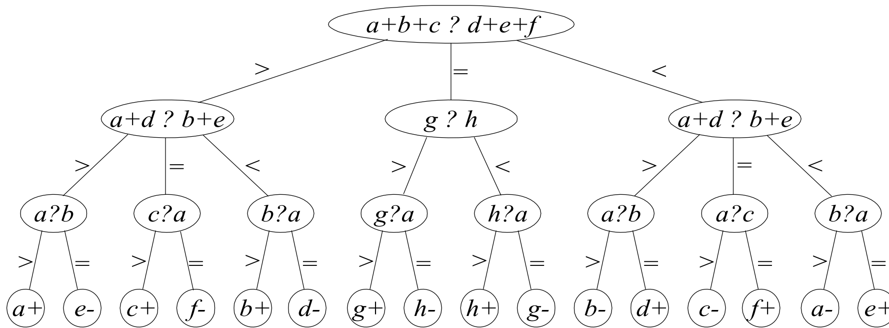
U unutrašnjim čvorovima stabla se beleži koliko elemenata liste ima u levom podstablu. Ako je indeks elementa koji se traži manji od date vrednosti, onda se taj element nalazi u levom podstablu. Ako je indeks elementa veći, onda se od njega oduzima data vrednost, a element se nalazi u desnom podstablu.

```
FIND(root, k)
r = k
p = root
while (left(p) ≠ nil or right(p) ≠ nil) do
  if (r ≤ lcount(p)) then
        p = left(p)
  else
        r = r – lcount(p)
        p = right(p)
  end_if
end_while
if (r ≠ 1) then p = nil
return p
```

Zadatak 5.2

Neka je dato osam novčića *a*, *b*, *c*, *d*, *e*, *f*, *g* i *h* od kojih je jedan lažan i ima različitu težinu od drugih. Na raspolaganju je vaga (terazije), a potrebno je odrediti koji je novčić lažan i da li je lakši ili teži od ostalih uz minimalan broj merenja. Optimalan proces odlučivanja realizovati u vidu stabla.

Rešenje:



Najjednostavnije rešenje je da se odabere jedan novčić (na pr. *a*) i da se njegova težina poredi sa težinom svih ostalih novčića. U najboljem slučaju bila bi potrebna najmanje 2 merenja, a najviše 7 – u proseku između 4 i 5 merenja.

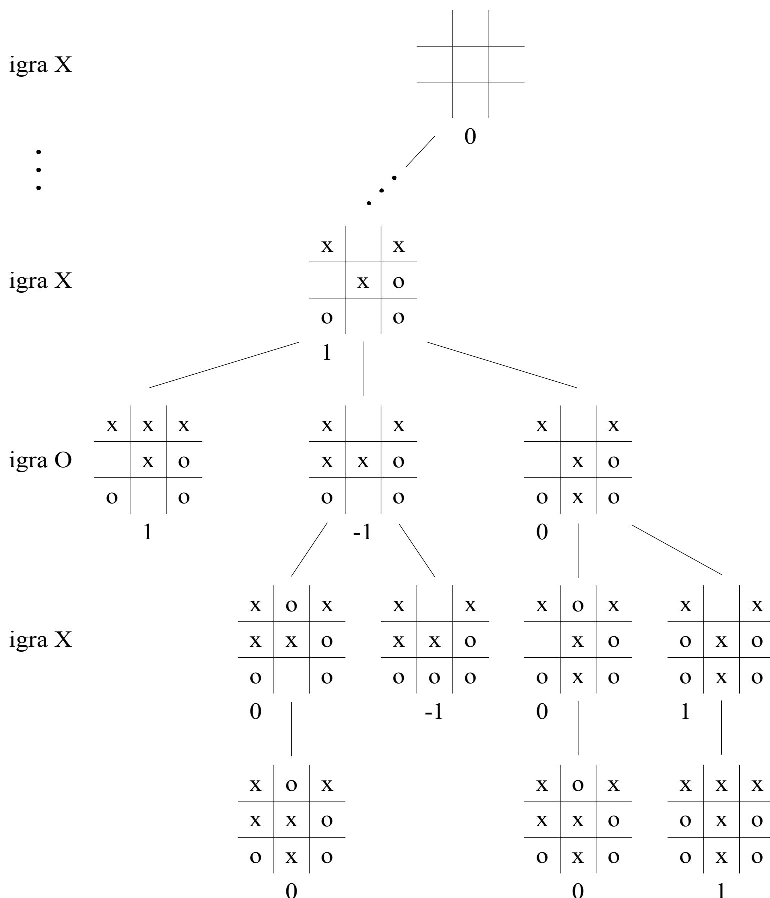
Na slici je ilustrovan proces odlučivanja koji dovodi do rešenja za tačno 3 merenja.

Treba primetiti da su u drugom merenju, za slučaj nejednake težine u prvom, novčići *b* i *d* zamenili tas. Ovo je neophodno uraditi iz sledećeg razloga: neka je, na primer *a*+*b*+*c* > *d*+*e*+*f*. Tada poređenje *a*+*b* ? *d*+*e* može da pruži samo podatak koja od grupa (*a*+*b* ili *c*+*d*) je teža, ali ne i koji novčić se razlikuje od ostalih.

Zadatak 5.3

Prikazati deo stabla za igru krstića i kružića X-OX (*tic-tac-toe*).

Rešenje: Izbor poteza u nekoj poziciji se zasniva na estimaciji: predviđaju se sve moguće pozicije i svakoj se dodeljuje vrednost koja označava njenu meru uspešnosti. Očekuje se da će svaki igrač u svom potezu povući za njega najbolji mogući potez, odnosno doći u najbolju moguću poziciju. Vrednost 0 označava da ako svaki igrač povlači najbolji mogući potez, nijedan od igrača ne može da pobedi. Vrednost 1 označava da iz date pozicije igrač "X" sigurno pobeđuje, a vrednost -1 označava da iz date pozicije igrač "O" sigurno pobeđuje. Pridruživanje vrednosti počinje od listova i vrši se propagacijom od čvorova–potomaka ka čvorovima–roditeljima. Za pozicije kada je igrač "X" na potezu, od interesa je vrednost 1 (igrač pobeđuje) ili vrednost 0 (igrač ne gubi), a za pozicije kada je igrač "O" na potezu, od interesa je vrednost -1 (igrač pobeđuje) ili vrednost 0 (igrač ne gubi).



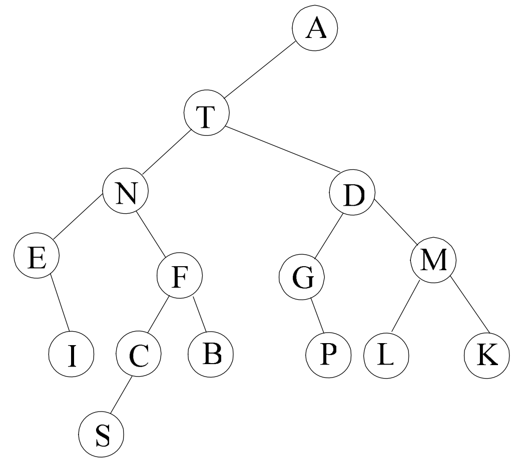
S obzirom na to da igrači naizmenično dobijaju pravo da igraju, cilj igrača "X" je da dođe do pozicije čija je vrednost 1 (ili 0) a da izbegne pozicije čija je vrednost -1. Cilj igrača "O" je da dođe do pozicije čija je vrednost -1 (ili 0) a da izbegne pozicije čija je vrednost 1.

Ova metoda traženja najboljeg mogućeg poteza se naziva **minimaks**: igrač "X" teži da u svom potezu maksimizuje, a igrač "O" da minimizuje vrednost pridruženu čvoru koji predstavlja narednu poziciju.

Zadatak 5.4

Šta je **preorder**, a šta **inorder** obilazak stabla? Ako za jedno binarno stablo **preorder** obilazak daje poredak ATNEIFCSBDGPMLK, a **inorder** obilazak daje poredak EINSCFBTGPDLMKA, rekonstruisati izgled ovog stabla i objasniti postupak.

Rešenje:

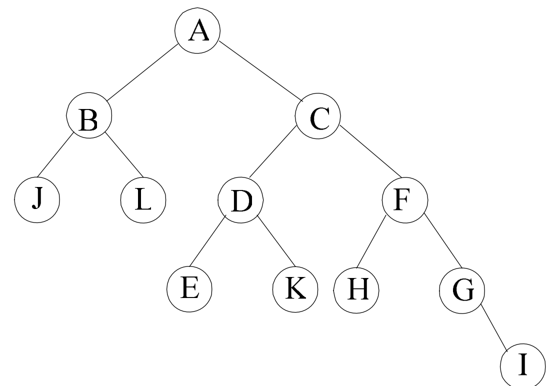


<u>*Preorder*</u>
1. Poseti se koren
2. Obiđe se levo podstablo na *preorder* način
3. Obiđe se desno podstablo na *preorder* način

<u>*Inorder*</u>
1. Obiđe se levo podstablo na *inorder* način
2. Poseti se koren
3. Obiđe se desno podstablo na *inorder* način

Zadatak 5.5

Šta je **inorder**, a šta **postorder** obilazak stabla? Ako za jedno binarno stablo **inorder** obilazak daje poredak **JBLAEDKCHFGI**, a **postorder** obilazak daje poredak **JLBEKDHIGFCA**, rekonstruisati izgled ovog stabla i objasniti postupak.



<u>*Postorder*</u>
1. Obiđe se levo podstablo na *postorder* način
2. Obiđe se desno podstablo na *postorder* način
3. Poseti se koren

<u>*Inorder*</u>
1. Obiđe se levo podstablo na *inorder* način
2. Poseti se koren
3. Obiđe se desno podstablo na *inorder* način

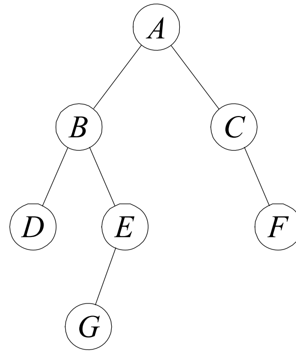
Zadatak 5.6

Napisati proceduru koja prikazuje iterativnu realizaciju inorder obilaska binarnog stabla.

```
INORDER-I(root)
next = root
loop
    while (next ≠ nil) do
        PUSH(S, next)
        next =left(next)
    end_while
    if (not STACK-EMPTY(S)) then
        next = POP(S)
        P(next)
        next = right(next)
    else
        return
    end_if
end_loop
```

Zadatak 5.7

Skicirati i objasniti iterativnu realizaciju inorder algoritma obilaska binarnog stabla. Ilustrovati rad algoritma po koracima na primeru sledećeg stabla.



Rešenje:

next	*Stack*	*inorder* poredak
A	*A*	-
B	*AB*	-
D	*ABD*	-
nil	*AB*	D
nil	*A*	*DB*
E	*AE*	*DB*
G	*AEG*	*DB*
nil	*AE*	*DBG*
nil	*A*	*DBGE*
nil		*DBGEA*
C	*C*	*DBGEA*
nil		*DBGEAC*
F	*F*	*DBGEAC*
nil		*DBGEACF*
nil		

Zadatak 5.8

U poruci se javlja 8 simbola (A, B, C, D, E, F, G, H) sa datim verovatnoćama pojavljivanja. Prikazati postupak izbora optimalnih prefiksnih kodova primenom standardnog **Huffman**-ovog algoritma.

Simboli	A	B	C	D	E	F	G	H
Verovatnoće	8	6	13	12	23	10	18	10

Rešenje:

Simboli	A	B	C	D	E	F	G	H
Kodovi	1101	1100	101	100	01	000	111	001

Zadatak 5.9

U jednom prenosnom sistemu poruke se sastoje od simbola A, B, C, D, E, F, G i H sa verovatnoćama pojavljivanja 0.29, 0.25, 0.2, 0.12, 0.05, 0.04, 0.03 i 0.02, respektivno. Kodirati simbole tako da prosečna dužena prenesene poruke bude minimalna i izračunati ovu dužinu. Obrazložiti postupak.

Simboli	A	B	C	D	E	F	G	H
Verovatnoće	0.29	0.25	0.2	0.12	0.05	0.04	0.03	0.02

Rešenje:

Simboli	A	B	C	D	E	F	G	H
Kodovi	11	01	00	100	10111	10110	10101	10100

Prosečna dužina: $2 \cdot 0.29 + 2 \cdot 0.25 + 2 \cdot 0.2 + 3 \cdot 0.12 + 5 \cdot 0.05 + 5 \cdot 0.04 + 5 \cdot 0.03 + 5 \cdot 0.02 = 2.54$ bita

Zadatak 5.10

U jednom prenosnom sistemu poruke se sastoje od simbola A, B, C, D, E, F, G i H sa verovatnoćama 0.18, 0.36, 0.15, 0.17, 0.07, 0.02, 0.03 i 0.02, respektivno. Kodirati simbole tako da prosečna dužina prenesene poruke bude minimalna i izračunati ovu dužinu. Obrazložiti postupak.

Simboli	A	B	C	D	E	F	G	H
Verovatnoće	0.18	0.36	0.15	0.17	0.07	0.02	0.03	0.02

Rešenje:

Simboli	A	B	C	D	E	F	G	H
Kodovi	111	0	101	110	1000	100110	10010	100111

Prosečna dužina: $3 \cdot 0.18 + 1 \cdot 0.36 + 3 \cdot 0.15 + 3 \cdot 0.17 + 4 \cdot 0.07 + 6 \cdot 0.02 + 5 \cdot 0.03 + 6 \cdot 0.02 = 2.53$ bita

6. Kompresija podataka

6.1 Dinamički (adaptivni) Huffman-ov kod

Motivacija za korišćenje dinamičkog (adaptivnog) koda:

- statistički podaci sekvence koju treba kodirati često nisu poznati
- kada su poznati, potrebno je i njih preneti do mesta dekodovanja (dodatno opterećenje)

Ideja je slična kao i kod statičkog koda: simboli koji se češće pojavljuju se kodiraju sa manje bita. Algoritam koristi binarno stablo kao osnovnu strukturu. Listovi stabla su simboli od kojih se sastoji poruka. Prijemnik mora da rekonstruiše identično stablo kao predajnik da bi uspešno dekodirao poruku.

Postoje razne implementacije algoritma (Knuth, FGK, Vitter, ...)

Objašnjenje algoritma

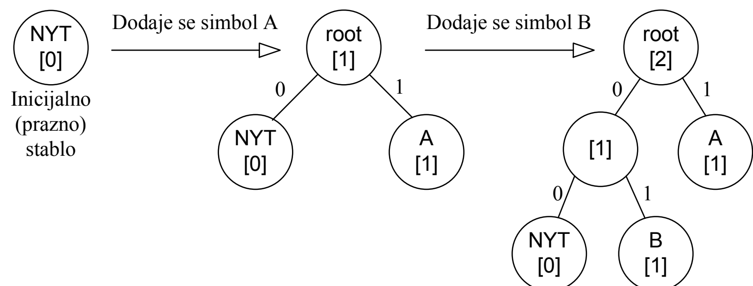
Svakom čvoru stabla se dodeljuje težina na sledeći način:

- težina lista je broj pojavljivanja datog simbola u do tada kodiranoj/dekodiranoj poruci
- težina nekog unutrašnjeg čvora stabla je suma težina direktnih potomaka (sinova) tog čvora
- specijalan čvor-list NYT (**N**ot **Y**et **T**ransmitted) težine 0

Kodiranje: pseudo-kod za dinamički Huffman-ov algoritam

```
DHuffman-encode
tree = CreateEmptyTree
InsertNode( tree, NYT )
next = getchar()
while next ≠ EOF
        if NodeExist(tree, next) then
            TransmitCode(tree, next)
        else
            TransmitCode(tree, NYT)
            TransmitSymbol(next)
            SpawnNode(tree, NYT, next)
        end_if
        UpdateTree(tree, next)
        next = getchar()
end_while
```

Na početku, stablo se sastoji od korena koji je ujedno čvor NYT. Dodavanje novog simbola u stablo stvara novi unutrašnji čvor stabla sa dva potomka: NYT (levi potomak) i list novog simbola (desni potomak). To je ilustrovano na sledećoj slici dodavanjem najpre simbola A a zatim simbola B. Težina čvora je označena unutar uglastih zagrada.



Ako već postoji čvor za simbol koji treba kodirati, onda se na izlaz šalje sekvenca bita koja označava putanju od korena do tog čvora (procedura *TransmitCode*). Na primer, za čvor B, ta sekvenca bi bila 01. Ako ne postoji čvor koji odgovara simbolu koji treba kodirati, onda se na izlaz šalje sekvenca bita koja označava putanju od korena do čvora NYT, a zatim se na izlaz šalje sam simbol (bez kodiranja).

Nakon obrade simbola (bilo da je već postojao odgovarajući čvor u stablu ili ne), vrši se ažuriranje stabla počevši od čvora koji predstavlja obrađen simbol. Ažuriranje se vrši u koracima. U svakom koraku se ažurira čvor roditeljski čvor od prethodno ažuriranog čvora na putu do korena stabla. Ažuriranje se prekida kada dođe do korena stabla. Generalno, ažuriranje stabla se svodi na pronalaženje čvora koji je bliži korenu a čija je težina jednaka težini čvora koji odgovara upravo obrađenom simbolu (a čiju težinu treba inkrementirati). Ako takav čvor postoji, onda se vrši zamena mesta datog čvora i čvora koji odgovara upravo obrađenom simbolu. **Treba obratiti pažnju da čvor ne može da zameni mesto sa svojim roditeljem, niti sa bilo kojim svojim pretkom**. **Takođe, koren stabla ne može da menja mesto sa bilo kojim drugim čvorom.**

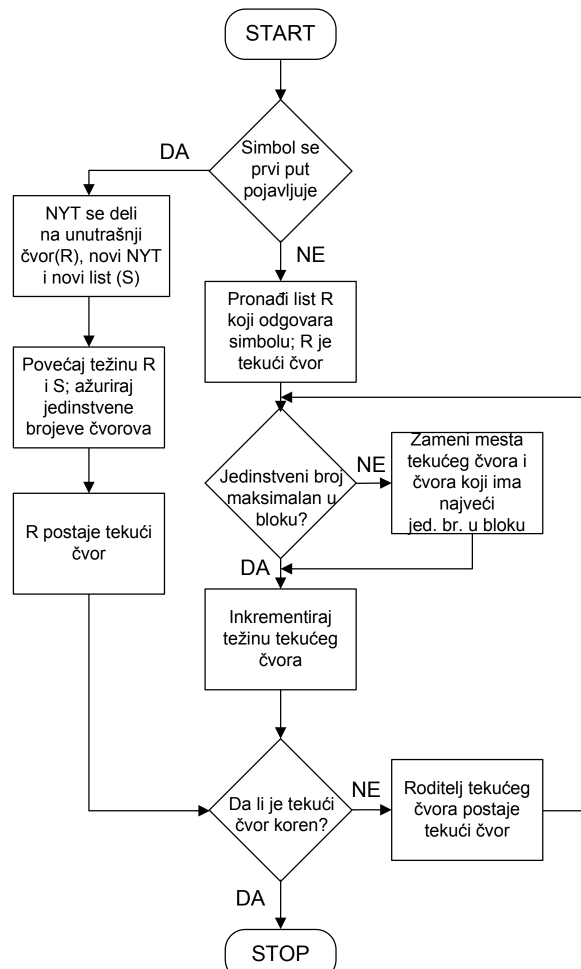
Razlikuju se dva slučaja ažuriranja stabla:

1. **Prvo pojavljivanje simbola**: dodat je simbol za koji u stablu nije postojao odgovarajući list. U ovom slučaju, list NYT se na ranije opisan način deli tako da se u stablo doda unutrašnji čvor, novi list NYT kao levi i novi list NEW koji odgovara dodatom simbolu kao desni potomak. Težina lista NEW je 1. Ažuriranje otpočinje od čvora roditelja čvora koji je prethodno bio NYT u oznaci R: u stablu se traži čvor najbliži korenu, iste težine kao i čvor R. Ako postoji, vrši se njihova zamena. Težina čvora R se inkrementira i ažuriranje se nastavlja od roditelja čvora R (novog ako je bilo zamene, starog ako nije) na isti način.
2. **Ponovno pojavljivanje simbola**: dodat je simbol za koji u stablu postoji odgovarajući list. U ovom slučaju, pre inkrementiranja težine datog lista se u stablu traži čvor, prema istom kriterijumu kao u prethodnoj tački, sa kojim bi dati čvor mogao da zameni mesto. Ako takav čvor postoji, vrši se zamena. Nakon toga, težina datog čvora se inkrementira i ažuriranje se dalje nastvlja na isti način od njegovog roditelja.

Neke implementacije ovog algoritma se baziraju na *jedinstvenom broju čvora* na sledeći način: svakom čvoru je dodeljen jedinstveni broj sa sledećim osobinama:

1. čvor veće težine ima veći jedinstven broj
2. roditeljski čvor ima veći jedinstven broj od svojih potomaka.

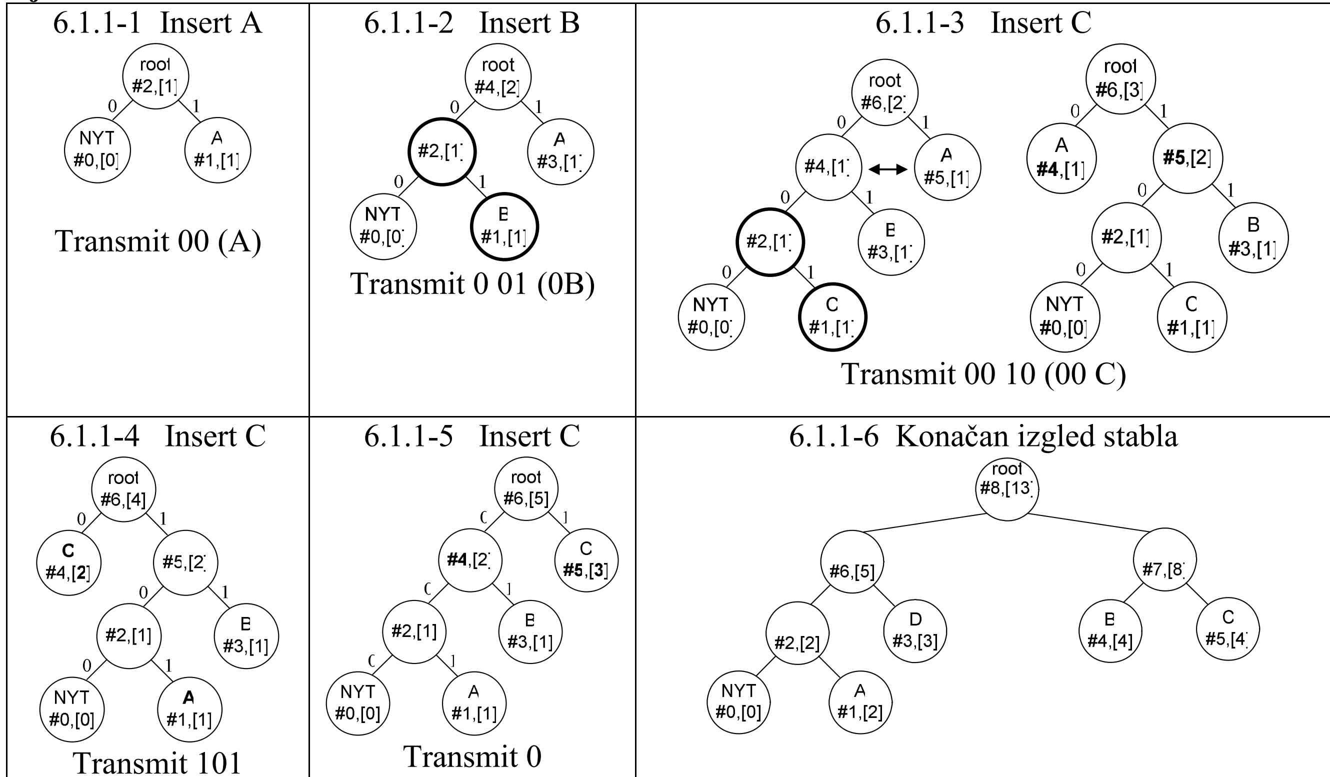
Ažuriranje stabla se obavlja tako da ove osobine budu očuvane nakon svakog ažuriranja. Takođe, u pomenutim implementacijama se uvodi pojam *bloka*: blok predstavlja skup čvorova tako da svi čvorovi iste težine pripadaju jednom bloku. Dijagram toka izvršavanja algoritma je dat na sledećoj slici, preuzetoj iz "Introduction to Data Compression" (Sayood Khalid).



Zadatak 6.1.1

Kodirati **dinamičkim Huffman**-ovim kodovima niz simbola ABCCCDDDCABBB ako se simboli A, B, C i D kodovima fiksne dužine kodiraju sa po dva bita 00, 01, 10, 11, respektivno.

Rešenje:



Objašnjenje: Jedinstvenim brojevima čvorova prethodi #, a u uglastim zagradama je obeležena težina čvorova. Prvi simbol za kodiranje je A. S obzirom na to da je stablo prazno (čvor NYT je koren) formira se stablo kao na slici 6.1.1-1) a na izlaz kodera se šalje sekvenca 00 (kod fiksne dužine za simbol A). Sledeći simbol koji se dodaje u stablo je simbol B. Odgovarajući čvor ne postoji u stablu (simbol se prvi put pojavljuje) pa se na izlaz šalje 0 (putanja od korena do lista NYT – videti sliku 6.1.1-1) a zatim 01 (kod fiksne dužine za simbol B). Izgled stabla nakon umetanja simbola B je dat na slici 6.1.1-2. Sledeći simbol koji se dodaje u stablo je simbol C. Postupak je isti kao i za simbol B. Međutim, prilikom ažuriranja stabla, konstatuje se da jedinstveni broj čvora #4 nije maksimalan u bloku čvorova težine 1, pa zato on i čvor najveće vrednosti jedinstvenog broja iz tog bloka (#5) zamenjuju mesto (slika 6.1.1-3 levo). Primetiti da su prilikom zamene mesta takođe razmenili i jedinstvene brojeve. Nakon zamene mesta, inkrementira se težina čvora, a zatim se ažuriranje nastavlja od čvora-roditelja (koji je ujedno koren stabla, pa se zapravo dalje ažuriranje prekida). Slika 6.1.1-3 desno prikazuje izgled stabla nakon umetanja simbola C. Sledeći simbol je ponovo C. On već postoji u stablu. Zato se na izlaz šalje putanja od korena do čvora koji predstavlja simbol C (101, videti sliku 6.1.1-3 desno), a zatim se vrši ažuriranje stabla. Pronalazi se čvor u bloku čvorova težine 1 sa najvećim jednistvenim brojem. To je čvor #4 (slika 6.1.1-3 desno) i čvor C sa njim menja mesto, a zatim se inkrementira težina čvora C. Izgled stabla nakon ažuriranja je prikazan na slici 6.1.1-4. Ponovno dodavanje simbola C u stablo uzrokuje da čvorovi #4 i #5 sa slike 6.1.1-4 zamene mesto, što je prikazano na slici 6.1.1-5.

Izlazna sekvenca je: 00 001 0010 101 0 00011 0101 10 0 101 1001 1101 01

Zadatak 6.1.2

Dekodirati niz bitova 1100001001001111011000111101 koji predstavlja **dinamički Huffmanov** kod za niz simbola sastavljen od simbola A, B, C i D čiji kodovi fiksne dužine se kodiraju sa po dva bita 00, 01, 10, 11, respektivno.

Rešenje: DAACDDCBB

Zadatak 6.1.3

Kodirati dinamičkim Huffman-ovim kodovima niz simbola BHCEHECCBBBBBB ako se simboli A, B, C, D, ..., H kodovima fiksne dužine kodiraju sa po tri bita 000, 001, 010, 011,..., 111 respektivno.

Rešenje: 001 0111 00010 100100 11 001 01 01 001 001 10 11 0 0

6.2 LZW Algoritam

Metoda je dobila naziv po autorima:
- Lempel i Ziv (1977) osnovni algoritam
- Welch (1984) unapređena verzija algoritma

LZW koncepcija:
- LZW algoritam kompresije zamenjuje **znakovne nizove** pojedinačnim kodovima
- Nema preliminarne analize ulaznog teksta (za razliku od statičkog Huffman-ovog koda)
- Svaki novi znakovni niz se dodaje tabeli znakovnih nizova
- Ušteda se postiže kada je izlaz jedan kod umesto znakovnog niza
- Kod znakovnog niza može biti proizvoljne dužine, ali mora biti duži od koda 1 znaka

Kompresija:

- LZW kompresija najpre pokuša da pronađe u tabeli kod za tekući znakovni niz
- Ako ne uspe – novi znakovni niz (i novi kod) se unose u tabelu, a izlaz je taj novi kod

```
STRING = getInputCharacter()
WHILE (not EOF(input)) DO
        CHARACTER = getInputCharacter()
        IF (TABLE.contains(STRING+CHARACTER)) then
            STRING = STRING+CHARACTER
        ELSE
            OutputCode(STRING)
            TABLE.add(STRING+CHARACTER)
            STRING = CHARACTER
        END_IF
END_WHILE
OutputCode(STRING)
```

Dekompresija:

- Algoritam dekompresije mora na osnovu niza kodova regenerisati originalni niz znakova
- LZW algoritam dekompresije kreira translacionu tabelu na osnovu komprimovanog sadržaja
- Ne mora se memorisati translaciona tabela (tabela znakovnih nizova)
- Polazi se od početne tabele koja sadrži samo skup osnovnih simbola, ali ne i znakovne nizove

```
OLD_CODE = getInputCode()
CHARACTER=TABLE.translate(OLD_CODE)
OutputStr(CHARACTER)
WHILE(not EOF(input)) DO
        NEW_CODE = getInputCode()
        if(not TABLE.contains(NEW_CODE))then
               STRING = TABLE.translate(OLD_CODE)+ CHARACTER
        else
               STRING = TABLE.translate(NEW_CODE)
        end_if
        outputStr(STRING)
        CHARACTER = firstCharacter(STRING)
        TABLE.add(TABLE.translate(OLD_CODE) + CHARACTER)
        OLD_CODE = NEW_CODE
 END_WHILE
```

Zadatak 6.2.1

Izvršiti kompresiju znakovnog niza "DADA_DA_DA_DA", ako je data početna tabela sa kodovima simbola.

simbol	Kod
D	0
A	1
_	2

Rešenje:

Input	STRING	CHARACTER	TABLE	Izlazni kod	Novi STRING
A	D	A	DA 3	0	A
D	A	D	AD 4	1	D
A	D	A	-	-	DA
_	DA	_	DA_ 5	3	_
D	_	D	_D 6	2	D
A	D	A	-	-	DA
_	DA	_	-	-	DA_
D	DA_	D	DA_D 7	5	D
A	D	A	-	-	DA
_	DA	_	-	-	DA_
D	DA_	D	-	-	DA_D
A	DA_D	A	DA_DA 8	7	A
EOF	A	-	-	1	-

Zadatak 6.2.2

Izvršiti dekompresiju ulaznog niza kodova ”0132571”, ako je data početna tabela sa kodovima simbola.

simbol	Kod
D	0
A	1
_	2

Rešenje:

input	NEW_CODE	STRING	CHARACTER	TABLE	OLD_CODE	output
0	-	-	-	-	0	D
1	1	A	A	DA 3	1	A
3	3	DA	D	AD 4	3	DA
2	2	_	_	DA_ 5	2	_
5	5	DA_	D	_D 6	5	DA_
7	7	DA_D	D	DA_D 7	7	DA_D
1	1	A	A	DA_DA 8	1	A

Napomena: prilikom dekodovanja ulazne sekvence, na ulazu će se pojaviti kod 7 pre nego što se odgovarajuća zamena pojavi u tabeli. Tada se zamena pravi od OLD_CODE + CHARACTER. Ovaj nedostatak osnovnog algoritma je uočio Welch.

Zadatak 6.2.3

Primeniti LZW kompresiju na primeru ulaznog stringa ”/WED/WE/WEE/WEB/WET”. Za pojedinačne simbole koristiti ASCII kodove.

Rešenje:

Input String = /WED/WE/WEE/WEB/WET			
Character Input	**Code Output**	**New code value**	**New String**
/W	/	256	/W
E	W	257	WE
D	E	258	ED
/	D	259	D/
WE	256	260	/WE
/	E	261	E/
WEE	260	262	/WEE
/W	261	263	E/W
EB	257	264	WEB
/	B	265	B/
WET	260	266	/WET
EOF	T		

Zadatak 6.2.4

Primeniti LZW dekompresiju na primeru ulaznog niza kodova ”/ W E D 256 E 260 261 257 B 260 T ”. Za pojedinačne simbole koristiti ASCII kodove.

Rešenje:

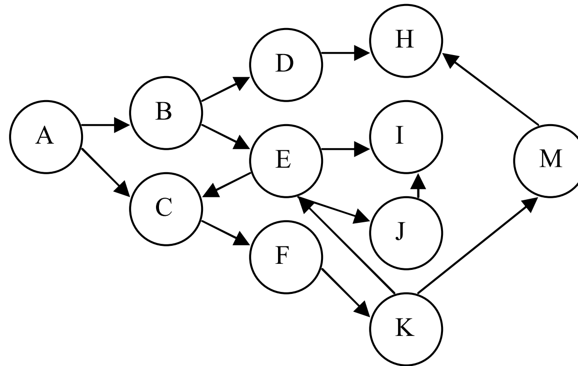
Input Codes: / W E D 256 E 260 261 257 B 260 T				
Input/ NEW_CODE	OLD_CODE	STRING/ Output	CHARACTER	New table entry
/	/	/		
W	/	W	W	256 = /W
E	W	E	E	257 = WE
D	E	D	D	258 = ED
256	D	/W	/	259 = D/
E	256	E	E	260 = /WE
260	E	/WE	/	261 = E/
261	260	E/	E	262 = /WEE
257	261	WE	W	263 = E/W
B	257	B	B	264 = WEB
260	B	/WE	/	265 = B/
T	260	T	T	266 = /WET

7. GRAFOVI

Zadatak 7.1

Na datom grafu ilustrovati:

a) obilazak grafa po širini
b) obilazak grafa po dubini



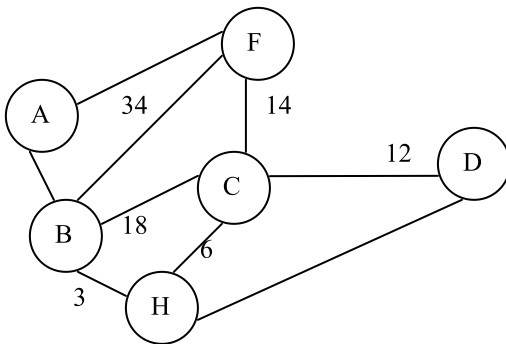
Rešenje:

a) ABCDEFHIJKM
b) ABDHEIJCFKM

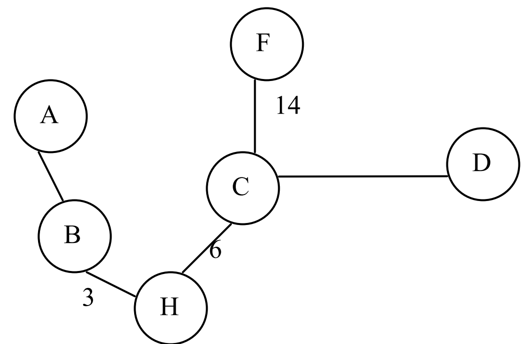
Zadatak 7.2

Za dati graf naći minimalno obuhvatno stablo:

a) upotrebom Prim-ovog algoritma
b) upotrebom Kruskal-ovog algoritma



Rešenje:

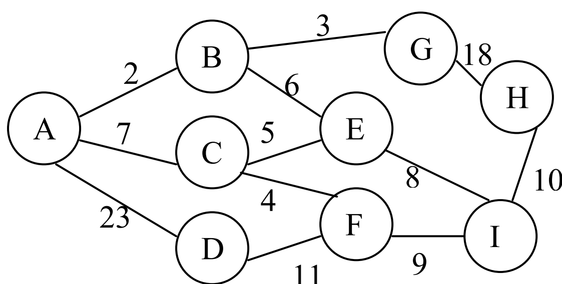


a) A-B, B-H, H-C, C-D, C-F
b) H-B, H-C, B-A, C-D, C-F

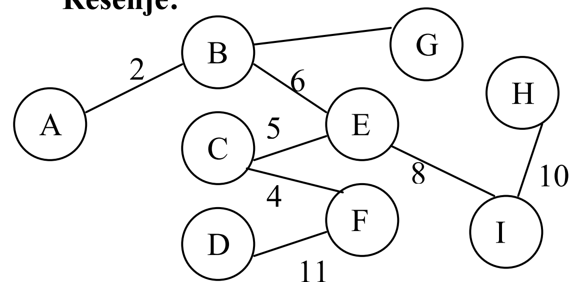
Zadatak 7.3

Za dati graf naći minimalno obuhvatno stablo:

a) upotrebom Prim-ovog algoritma
b) upotrebom Kruskal-ovog algoritma



Rešenje:



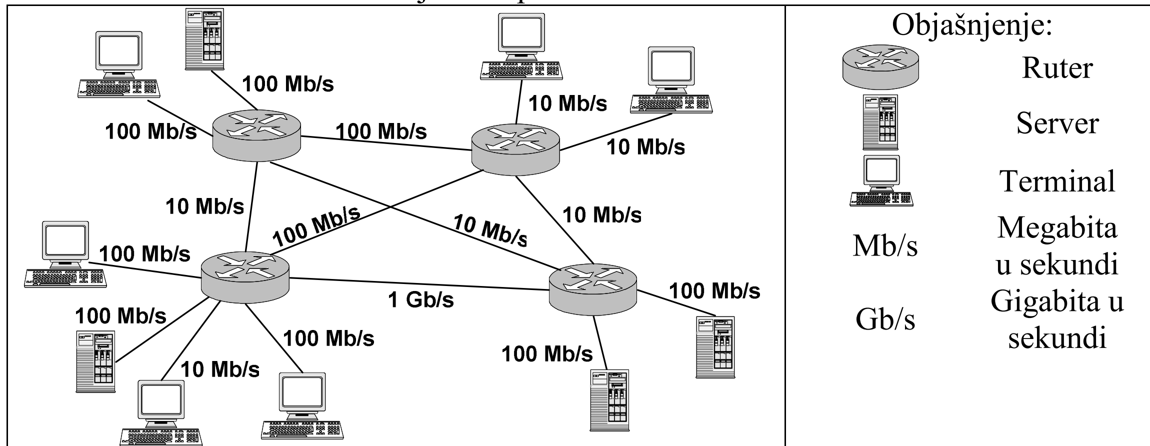
a) A-B, B-G, B-E, E-C, C-F, E-I, I-H, F-D
b) A-B, B-G, C-F, E-C, B-E, E-I, I-H, F-D

Zadatak 7.4

Na slici je prikazana šema jedne računarske mreže, koja se sastoji od **servera**, **terminala** i **rutera**. Uloga **rutera** je da prosleđuju komunikaciju između **servera** i **terminala** preko **najbržih** veza. Pri tome, oni obezbeđuju da između **bilo koja dva** uređaja (rutera, servera ili terminala) postoji **tačno jedan put**.

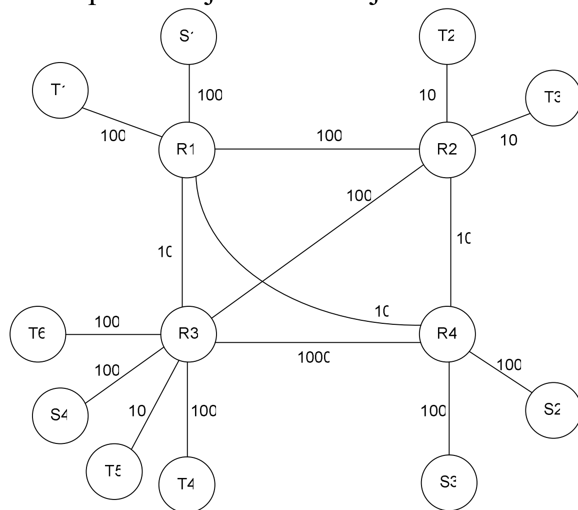
a) Objasniti kako bi se neka računarska mreža modelirala grafom. Nacrtati graf ekvivalentan prikazanoj mreži.

b) Predložiti algoritam za određivanje veza preko kojih ruteri treba da primaju i prosleđuju poruke prema uslovima zadatka. Proces određivanja veza prikazati u koracima.



Rešenje:

Prikazana računarska mreža se može modelirati grafom tako što bi svaki server, terminal ili ruter bio predstavljen čvorom, a njihove međusobne veze granama grafa. Graf topološki ekvivalentan datoj računarskoj mreži prikazan je na sledećoj slici.



Čvorovi koji predstavljaju pojedinačne uređaje su obeleženi slovnim oznakama (R-ruter, S-server, T-terminal) i pratećim rednim brojevima. Međutim, s obzirom na to da je u ovom slučaju nebitno o kojoj vrsti uređaja se radi, oznake su isto tako mogle da budu druge, nepovezane oznake.

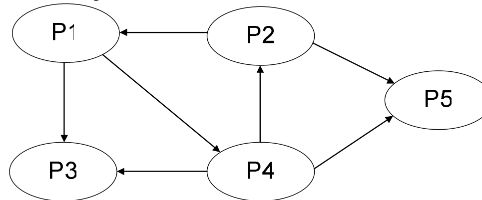
Radi jednostavnije analize problema, težine grana su određene tako što su najpre brzine svih veza prikazane u jedinici Mb/s, a zatim je jedinica izostavljena jer je ona nebitna.

U postavci zadatka se traži da između bilo koja dva čvora grafa postoji tačno jedan put i da se pri tome koriste najbrži raspoloživi putevi. Ovde se može prepoznati zadatak određivanja *maksimalnog* obuhvatnog stabla, jer se u ovom slučaju, za razliku od minimalnog obuhvatnog stabla, teži zadržavanju grana maksimalne težine. Iz tog razloga može da se koristi bilo koji od algoritama za određivanje minimalnog obuhvatnog stabla uz modifikaciju da se traže grane maksimalne a ne minimalne težine. Alternativno, pre dodele težina grana, brzine su mogle biti predstavljene u recipročnim jedinicama (s/b – sekundi po bitu). Tada bi veze najveće brzine bile predstavljene najmanjom vrednošću pa bi se algoritmi za određivanje minimalnog obuhvatnog stabla mogli primeniti bez modifikacije.

Studentima se prepušta da odrede obuhvatno stablo za dati graf primenom modifikacije nekog od poznatih algoritama.

Zadatak 7.5

Programski sistem se sastoji od programskih modula P1, P2, P3, P4, P5. Ovaj sistem je predstavljen datim usmerenim grafom u kome su čvorovi moduli, a grane pozivi između njih, tako da grana (i,j) odgovara pozivu modula Pj, od strane modula Pi. Odrediti koji su moduli rekurzivni.



Rešenje:

Rekurzija može biti direktna ili indirektna:

- direktna: modul poziva sam sebe
- indirektna: modul A poziva modul B koji poziva modul A

Pojava ciklusa u grafu ukazuje na postojanje rekurzije

Matrica puta pokazuje međusobnu povezanost čvorova

- p[i, j] = 1 ako postoji put od čvora i do j
- p[i, j] = 0 u suprotnom

Ponekad se naziva matrica dostižnosti (*reachability*). Polazi se od matrice eksplicitno zadatih putanja (direktne veze između čvorova grafa):

$$p^{(1)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Ako postoji put dužine 1 od čvora i do čvora j, u i-toj vrsti i j-toj koloni matrice $p^{(1)}$ se nalazi vrednost 1, u suprotnom 0. Na isti način se može videti postojanje puteva dužine tačno 2 u matrici $p^{(2)}$ koja se dobija kao logički proizvod $p^{(1)}*p^{(1)}$. Postojanje puteva dužine 2 ili manje se određuje na osnovu matrica $p^{(1)}$ i $p^{(2)}$: put od čvora i do čvora j postoji ako $p^{(1)}_{(i,j)} = 1$ ili $p^{(2)}_{(i,j)} = 1$. Daljim postupkom (logičkim množenjem $p^{(2)}*p^{(1)}$) dobijajaju se putevi dužine 3, itd. Postupak se sprovodi sve dok se ne pronađe matrica puteva dužine *n* (broj čvorova grafa), odnosno 5 u ovom primeru.

$$p^{(2)} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad p^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad p^{(4)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad p^{(5)} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Konačno, matrica dostižnosti se dobija logičkim sabiranjem svih dobijenih matrica:

$$p = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Rekurzivni su oni moduli koji u matrici puteva imaju vrednost 1 na glavnoj dijagonali ($p_{(i,i)} = 1$). To su moduli 1, 2 i 4.

Matrica dostižnosti se može dobiti i primenom *Warshall*-ovog algoritma. Polazi se od matrice eksplicitno zadatih putanja (direktne veze između čvorova grafa) i u svakom koraku *k* matrica dostižnosti se ažurira, odnosno dodaju se putevi *p*[*i,j*] koji se mogu ostvariti preko čvora *k*.

```
WARSHALL
P=A
for k=1 to n do
   for i=1 to n do
      for j=1 to n do
         p[i,j]=p[i,j] or (p[i,k] and p[k,j])
      end_for
   end_for
end_for
```

Treba primetiti da je u najugnježdenijoj petlji *p*[*i,k*]=const, a *p*[*i,j*] se ne menja ako je p[*i,k*]=0. Zato najugnježdeniju petlju treba izvršavati samo ako je *p*[*i,k*]=1. Drugim rečima, najugnježdeniju petlju treba zameniti sledećim kodom:

```
if (p[i,k]=1) then
   for j=1 to n do
      p[i,j]=p[i,j] or p[k,j]
   end_for
end_if
```

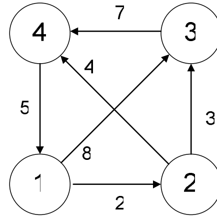
U nastavku je dat izgled matrice dostižnosti posle svakog koraka. Krajnji rezultat je, naravno, isti kao i onaj dobijen prethodnom metodom, ali je složenost postupka $O(n^3)$.

$$p = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad p^{(1)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad p^{(2)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$p^{(3)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad p^{(4)} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad p^{(5)} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Zadatak 7.6

Definisati pojmove ekscentričnosti čvora i središta grafa. Za graf sa slike naći ekscentričnosti svih čvorova i središte grafa



Ekscentričnost čvora v se definiše kao **maksimum najkraćih rastojanja** d[i, v] od svih čvorova grafa i do čvora v, $ecc(v, v \in V) = \max \{d[i, v] : i \in V\}$

Središte grafa se definiše kao čvor koji ima **najmanju ekscentričnost** $(\min \{ecc(v), v \in V\})$

Problem se rešava *Floyd*-ovim algoritmom:

FLOYD(*W*)	Ulaz algoritma: matrica težina W (n x n):

```
FLOYD(W)
D = W
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
      if (d[i, j] > d[i, k] + d[k, j]) then
          t[i, j] = t[k, j]
          d[i, j] = d[i, k] + d[k, j]
      end_if
    end_for
  end_for
end_for
```

Ulaz algoritma: matrica težina W (n x n):

- $w[i, j] =$ 0 ako je $i = j$,
- $w[i, j] =$ $w(i, j)$ ako je $i \neq j$ i $(i, j) \in E$,
- $w[i, j] =$ ∞ ako je $i \neq j$ i $(i, j) \notin E$

Izlaz algoritma:

- matrica D, d[i,j] je najkraće rastojanje i-j
- matrica T, t[i,j] je pretposlednji čvor na putu i-j

Rešenje:

Inicijalno:

$$D = \begin{bmatrix} 0 & 2 & 8 & \infty \\ \infty & 0 & 3 & 4 \\ \infty & \infty & 0 & 7 \\ 5 & \infty & \infty & 0 \end{bmatrix} \qquad T = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 \\ 4 & 0 & 0 & 0 \end{bmatrix}$$

k=1:

$$D = \begin{bmatrix} 0 & 2 & 8 & \infty \\ \infty & 0 & 3 & 4 \\ \infty & \infty & 0 & 7 \\ 5 & \mathbf{7} & \mathbf{13} & 0 \end{bmatrix} \qquad T = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 \\ 4 & \mathbf{1} & \mathbf{1} & 0 \end{bmatrix}$$

k=2:

$$D = \begin{bmatrix} 0 & 2 & \mathbf{5} & \mathbf{6} \\ \infty & 0 & 3 & 4 \\ \infty & \infty & 0 & 7 \\ 5 & 7 & \mathbf{10} & 0 \end{bmatrix} \qquad T = \begin{bmatrix} 0 & 1 & \mathbf{2} & \mathbf{2} \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 \\ 4 & 1 & \mathbf{2} & 0 \end{bmatrix}$$

k=3:

$$D = \begin{bmatrix} 0 & 2 & 5 & 6 \\ \infty & 0 & 3 & 4 \\ \infty & \infty & 0 & 7 \\ 5 & 7 & 10 & 0 \end{bmatrix} \qquad T = \begin{bmatrix} 0 & 1 & 2 & 2 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 \\ 4 & 1 & 2 & 0 \end{bmatrix}$$

k=4:

$$D = \begin{bmatrix} 0 & 2 & 5 & 6 \\ \mathbf{9} & 0 & 3 & 4 \\ \mathbf{12} & \mathbf{14} & 0 & 7 \\ 5 & 7 & 10 & 0 \end{bmatrix} \qquad T = \begin{bmatrix} 0 & 1 & 2 & 2 \\ \mathbf{4} & 0 & 2 & 2 \\ \mathbf{4} & \mathbf{1} & 0 & 3 \\ 4 & 1 & 2 & 0 \end{bmatrix}$$

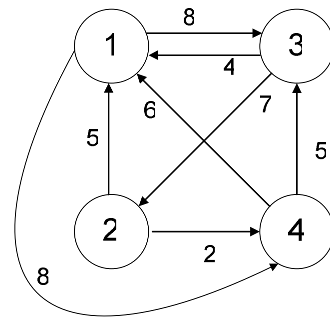
Ekscentričnosti čvorova su (redom): 12, 14, 10, 7 (posmatra se matrica D po kolonama). Središte grafa je čvor 4.

Za rekonstrukciju najkraćeg puta koristi se sledeći algoritam nad matricom T:

```
PATH(i, j)
if (i = j) then
        PRINT(i)
        return
else
        if (t[i, j] = 0) then  PRINT(Nema puta između i i j)
        else
                PATH(i, t[i, j])
                PRINT(j)
        end_if
end_if
```

Zadatak 7.7

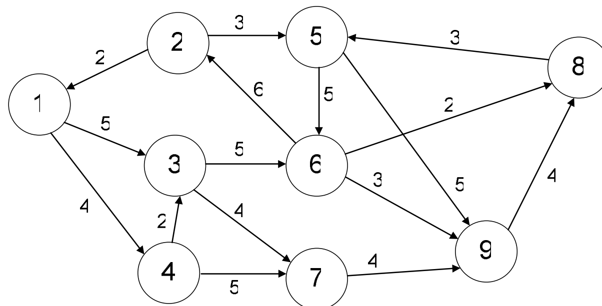
Za graf sa slike naći ekcentričnosti svih čvorova i središte grafa.



Rešenje:
Ekscentričnosti čvorova:
1: **6**
2: 15
3: 8
4: 9

Zadatak 7.8

Kakva je namena Dijkstra algoritma? Za graf sa slike naći najkraće puteve od čvora 6 do ostalih čvorova primenom Dijkstra-inog algoritma. Dati izgled vektora najkraćih rastojanja i vektora prethodnika posle svake iteracije.



```
DIJKSTRA(W)
S = {x}
for i = 1 to n, i≠x do
        d[i] = w[x, i]
        if (w[x, i] ≠ ∞) then t[i] = x
        else    t[i] = 0
        end_if
end_for
for k = 1 to n - 1 do
        find min {d[i] : i ∈ (V – S) }
        if (d[i] = ∞) then return
        S = S + {i}
        for each j ∈ (V – S) do
                if (d[i] + w[i, j] < d[j]) then
                        d[j] = d[i] + w[i, j]
                        t[j] = i
                end_if
        end_for
end_for
```

Ulaz predstavlja matrica težina *W*.
Izlaz čine:
- vektor d (dužina n)
- vektor *t* (dužina n)

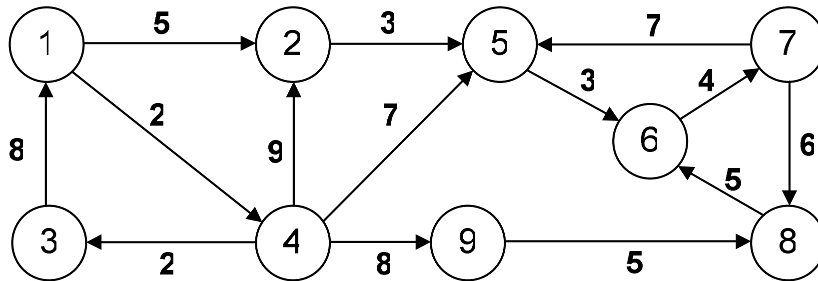
gde je:
- *d*[*i*] najkraće rastojanje od polaznog čvora x do čvora *i*,
- *t*[*i*] je čvor-prethodnik čvora *i* na najkraćem putu od čvora 1

Ukoliko u nekom koraku algoritma ne može da se pronađe put ni do jednog od preostalih čvorova, algoritam prekida rad (nisu svi čvorovi dostižni iz polaznog čvora). Vektori *d* i *t* ostaju ažurni, odnosno pronađeni su najkraći putevi do dostižnih čvorova, a pronađeni su i oni čvorovi koji su nedostižni.

s		d								t							
		1	2	3	4	5	7	8	9	1	2	3	4	5	7	8	9
6	-	∞	6	∞	∞	∞	∞	2	3	0	6	0	0	0	0	6	6
6,8	8	∞	6	∞	∞	5	∞	2	3	0	6	0	0	8	0	6	6
6,8,9	9	∞	6	∞	∞	5	∞	2	3	0	6	0	0	8	0	6	6
6,8,9,5	5	∞	6	∞	∞	5	∞	2	3	0	6	0	0	8	0	6	6
6,8,9,5,2	2	8	6	∞	∞	5	∞	2	3	2	6	0	0	8	0	6	6
6,8,9,5,2,1	1	8	6	13	12	5	∞	2	3	2	6	1	1	8	0	6	6
6,8,9,5,2,1,4	4	8	6	13	12	5	17	2	3	2	6	1	1	8	4	6	6
6,8,9,5,2,1,4,3	3	8	6	13	12	5	17	2	3	2	6	1	1	8	4	6	6
6,8,9,5,2,1,4,3,7	7	8	6	13	12	5	17	2	3	2	6	1	1	8	4	6	6

Zadatak 7.9

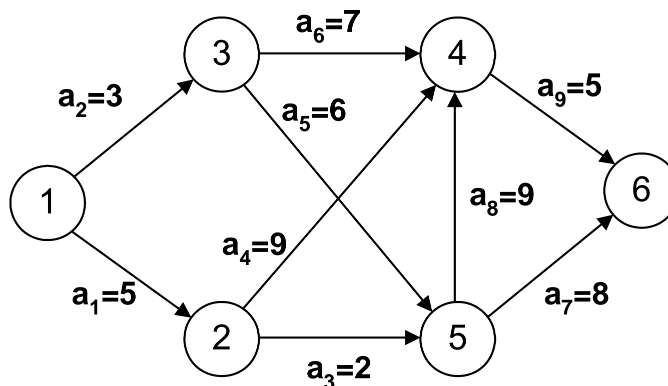
Za graf sa slike naći najkraće puteve od čvora 4 do ostalih čvorova primenom Dijkstra-inog algoritma. Dati izgled vektora najkraćih rastojanja i vektora prethodnika posle svake iteracije.



S		d								t							
		1	2	3	5	6	7	8	9	1	2	3	5	6	7	8	9
4	-	∞	9	2	7	∞	∞	∞	8	0	4	4	4	0	0	0	4
4,3	3	10	9	**2**	7	∞	∞	∞	8	3	4	4	4	0	0	0	4
4,3,5	5	10	9	**2**	**7**	10	∞	∞	8	3	4	4	4	5	0	0	4
4,3,5,9	9	10	9	**2**	**7**	10	∞	13	**8**	3	4	4	4	5	0	9	4
4,3,5,9,2	2	10	**9**	**2**	**7**	10	∞	13	**8**	3	4	4	4	5	0	9	4
4,3,5,9,2,1	1	**10**	**9**	**2**	**7**	10	∞	13	**8**	3	4	4	4	5	0	9	4
4,3,5,9,2,1,6	6	**10**	**9**	**2**	**7**	**10**	14	13	**8**	3	4	4	4	5	6	9	4
4,3,5,9,2,1,6,8	8	**10**	**9**	**2**	**7**	**10**	14	**13**	**8**	3	4	4	4	5	6	9	4
4,3,5,9,2,1,6,8,7	7	**10**	**9**	**2**	**7**	**10**	**14**	**13**	**8**	3	4	4	4	5	6	9	4

Zadatak 7.10

Pronaći kritičan put i dozvoljena kašnjenja na projektu predstavljenim grafom sa slike.



Rešenje:

Grafom je prikazan redosled odvijanja faza projekta. Grana od faze X ka fazi Y označava da faza Y počinje nakon što se neophodan deo faze X završi. Težina grane označava trajanje neophodnog dela faze X.

Kritičan put: Put najveće dužine između početnog i završnog čvora grafa kojim je predstavljen projekat, a koji određuje vreme trajanja projekta

Vremenska složenost određivanja kritičnog puta: O(e+n)

Da bi se odredio kritičan put i dozvoljena kašnjenja, potrebno je odrediti za svaki čvor:

- najranije vreme otpočinjanja (*earlyest start time*) EST
  $$EST[i] = \max\{EST[j] + w(j, i) : j \in P(i)\}$$
  po definiciji: EST[i] = 0

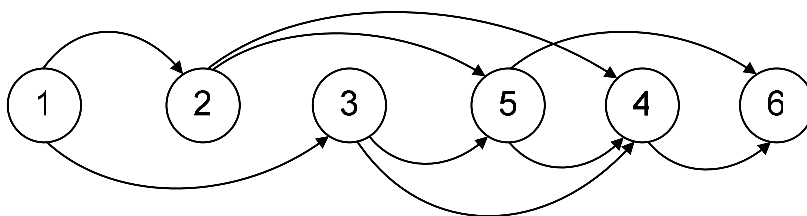
- najkasnije vreme otpočinjanja (latest start time) LST
  $$LST[i] = \min\{LST[j] - w(i, j) : j \in S(i)\}$$
  po definiciji: LST[n] = EST[n]

- Dozvoljeno kašnjenje:
  $$L[i] = LST[i] – EST[i]$$
  Za događaje na kritičnom putu kašnjenje nije dozvoljeno (L=0)

```
CRITICAL-PATH(G)
TOP-SORT(G,T)
EST[T[1]]=0
for u = 2 to n do
        i = T[u]
        for each j ∈ P(i) do
                EST[i] = max{ EST[j] + w(j, i) }
        end_for
end_for
LST[T[n]] = EST[T[n]]
for u = n-1 downto 1 do
        i = T[u]
        for each j ∈ S(i) do
                LST[i] = min{ LST[j] - w(i, j) }
        end_for
end_for
for i = 1 to n do
        L[i] = LST[i] – EST[i]
end_for
```

TOP-SORT: vrši topološko sortiranje čvorova grafa

P(i) : skup prethodnika čvora i
S(i) : skup sledbenika čvora i



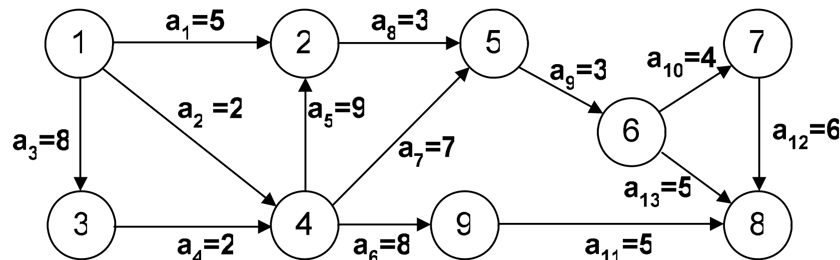
Topološki sortiran graf

i	*EST*	*LST*	*L*
1	**0**	**0**	**0**
2	**5**	**7**	**2**
3	**3**	**3**	**0**
5	**9**	**9**	**0**
4	**18**	**18**	**0**
6	**23**	**23**	**0**

Ovaj algoritam se u računarskoj nauci se naziva ***C****ritical* ***P****ath* ***M****ethod* (CPM). Algoritam prvo vrši topološko sortiranje usmerenog grafa, koje podrazumeva linearizaciju čvorova grafa, odnosno određivanje linearnog poretka čvorova tako da se svaki čvor u tom redosledu javlja pre svog suseda. Ovakav linearni poredak čvorova grafa moguće je ustanoviti ako i samo ako je graf acikličan.

Zadatak 7.11

Pronaći kritičan put i dozvoljena kašnjenja na projektu predstavljenim grafom sa slike. Prikazati topološko sortiranje grafa po koracima.



Rešenje:

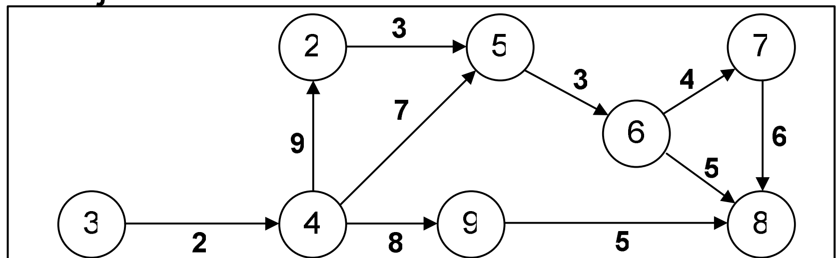
Za svaki aciklični usmereni graf (***D****irected* ***A****cyclic* ***G****raph,* ***DAG***), poput grafa prikazanog na slici, može se naći najmanje jedan čvor koji nema prethodnika, odnosno koji nije susedan bilo kom drugom čvoru, što znači da je njegov ulazni stepen grananja jednak nuli. Taj čvor će biti prvi u linearnom poretku. Izostavljanjem tog čvora i izlaznih grana dobija se novi aciklični graf nad kojim se ponavlja isti postupak. S obzirom na to da u acikličnom grafu postoji **najmanje** jedan čvor bez prethodnika (dakle, može ih biti više) izbor narednog elementa u linearnom poretku u opštem slučaju nije jedinstven, pa zato ni konačni topološki poredak ne mora biti jedinstven.

<table>
<tr>
<td rowspan="2">

<u>TOP-SORT(*G*)</u>
$A = V(G)$
$B = E(G)$
for $i = 1$ **to** n **do**
    find $u \in A : d_{in}(u) = 0$
    $T[i] = u$
    $A = A - \{u\}$
    for all v, $(u, v) \in B$ **do**
        $B = B - \{(u, v)\}$
    end_for
end_for

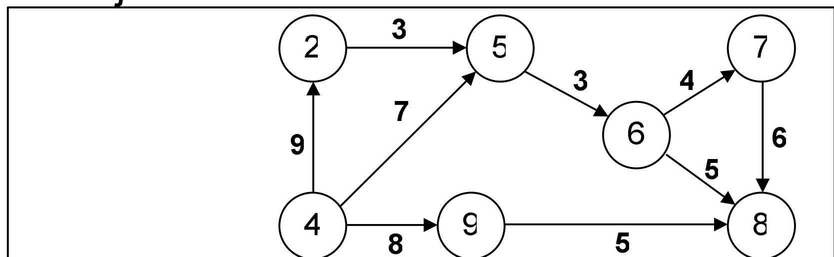
</td>
<td>

Uklanja se čvor 1



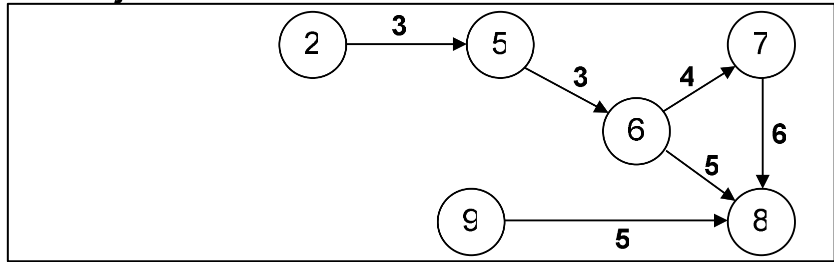
</td>
</tr>
<tr>
<td>

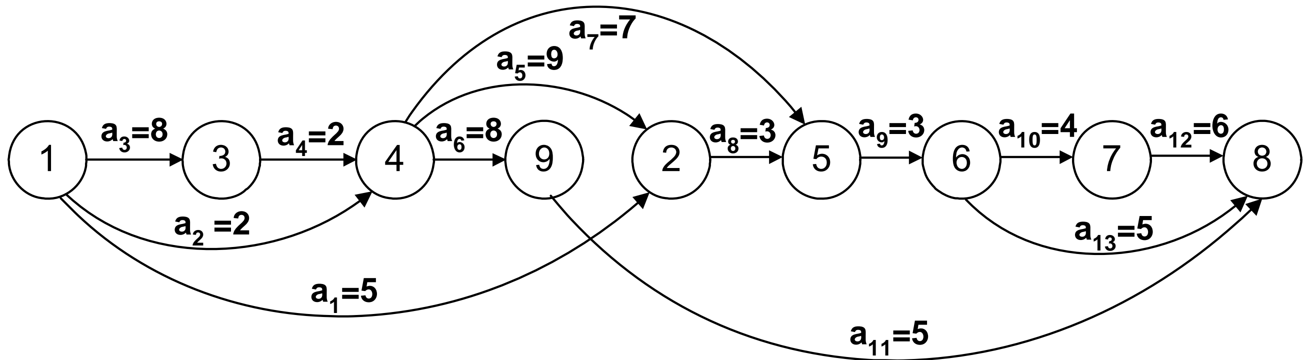
Uklanja se čvor 3



</td>
</tr>
</table>

Uklanja se čvor 4





Identična analiza se može sprovesti ako se graf linearizuje u suprotnom poretku, odnosno izborom čvora koji će biti *poslednji* u linearnom poretku. To će biti čvor koji nema susede (u konkretnom primeru, to je čvor 8), odnosno kome je *izlazni stepen* jednak nuli. Ukljanjanjem tog čvora (proglašavajući ga *poslednjim* u linearnom poretku) i uklanjanjem svih njegovih *ulaznih* grana, dobija se novi aciklični graf nad kojim se može sprovesti identičan postupak. Studentima se preporučuje da konstruišu topološki poredak datog grafa i na ovaj način.

i	EST	LST	L
1	0	0	0
3	8	8	0
4	10	10	0
9	18	30	12
2	19	19	0
5	22	22	0
6	25	25	0
7	29	29	0
8	35	35	0

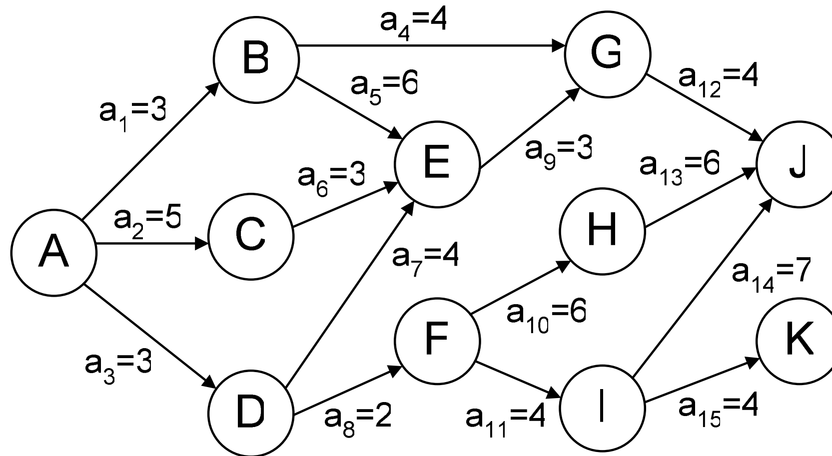
aktivnost	*l*
1	**14**
2	**8**
3	**0**
4	**0**
5	**0**
6	**12**
7	**5**
8	**0**
9	**0**
10	**0**
11	**12**
12	**0**
13	**5**

Desna tabela određuje za svaku aktivnost, predstavljenu granom a_i, dozvoljeno kašnjenje te aktivnosti. Za granu koja povezuje čvorove i i j ovo kašnjenje će iznositi $LST[j] – EST[i] - a_{ij}$.

Kritične aktivnosti su one koje ne smeju kasniti, tj. dozvoljeno kašnjenje je jednako nuli. To su aktivnosti 3, 4, 5, 8, 9, 10 i 12. Kašnjenje ovih aktivnosti bi sigurno produžilo ceo projekat. Sa druge strane, ne postoji implikacija da bi ubrzanje ove aktivnosti ubrzalo projekat.

Zadatak 7.12

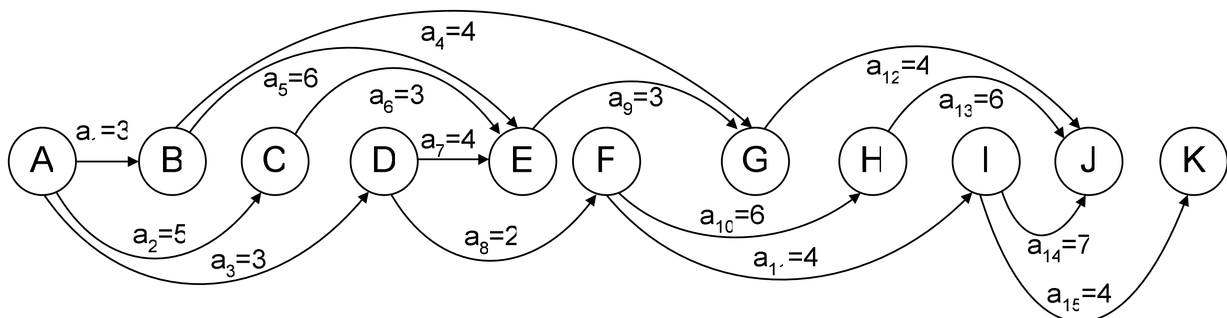
Šta je PERT graf? Šta je kritičan put i kolika je ukupna vremenska složenost određivanja kritičnog puta? Naći najranije i najkasnije trenutke započinjanja aktivnosti za projekat čiji je PERT graf dat na slici, kao i dozvoljeno kašnjenje za svaku od njih. Navesti kritične aktivnosti.



PERT – Project Evaluation & Review Technique

- 1958: US DoD – US Navy Special Projects Office
- Razvijen kao deo projekta "Polaris"
- PERT graf se još naziva PERT mreža (network)
- Napredniji od CPM
    - procene (optimistična, najverovatnija, pesimistična)
    - procena vremena (o + 4*n + p)/6

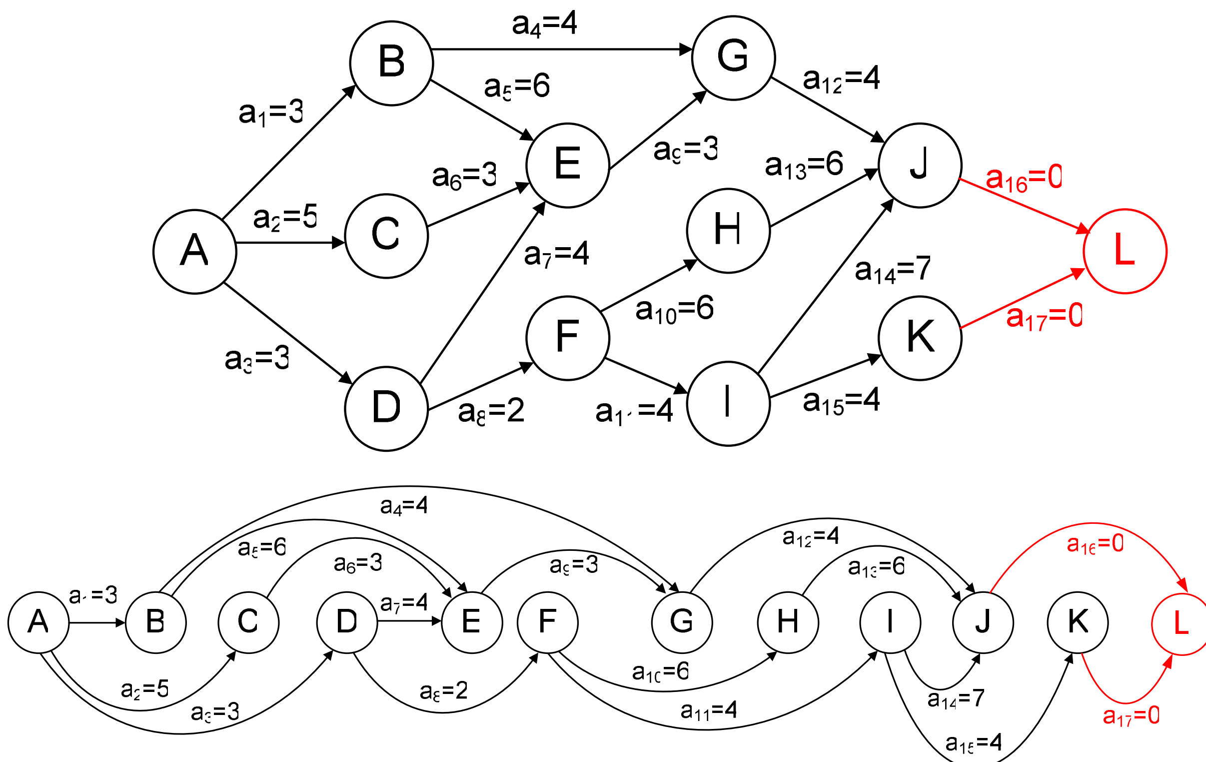
Kod metode kritičnog puta je dozvoljeno da postoji samo jedan terminalni čvor. Kod PERT-a se može pojaviti više terminalnih čvorova (koji nemaju sledbenike). Primenjuje se isti postupak, s tim što se kod određivanja LST za terminalne čvorove uzima najveća vrednost od LST terminalnih čvorova (ili EST, pošto LST za terminalne čvorove inicijalno dobija vrednost odgovarajućeg EST).



i	*EST*	*LST*	*L*
A	0	0	<u>0</u>
B	3	4	1
C	5	7	2
D	3	3	<u>0</u>
E	9	10	1
F	5	5	<u>0</u>
G	12	13	1
H	11	11	<u>0</u>
I	9	10	1
J	17	17	<u>0</u>
K	13	**<u>17</u>**	4

a	*l*
1	1
2	2
3	0
4	6
5	1
6	2
7	3
8	0
9	1
10	0
11	1
12	1
13	0
14	1
15	4

Alternativni pristup određivanju kritičnih aktivnosti PERT grafa podrazumeva svođenje na CPM. Svođenje se obavlja na sledeći način: uvede se dodatni čvor (označen sa L na sledećoj slici). Za svaki terminalni čvor polaznog PERT grafa doda se po jedna grana (aktivnost) do novog čvora. Dodate aktivnosti imaju trajanje jednako nuli. Na ovaj način je dobijen graf sa jednim terminalnim čvorom. Potom se sprovede ista procedura topološkog sortiranja i određivanja kritičnog puta i kritičnih aktivnosti.



i	*EST*	*LST*	*L*
A	0	0	0
B	3	4	1
C	5	7	2
D	3	3	0
E	9	10	1
F	5	5	0
G	12	13	1
H	11	11	0
I	9	10	1
J	17	17	0
K	13	17	4
L	17	**17**	0

a	*l*
1	1
2	2
3	0
4	6
5	1
6	2
7	3
8	0
9	1
10	0
11	1
12	1
13	0
14	1
15	4
16	0
17	4

Aktivnosti koje se vode do novog čvora treba na kraju izostaviti iz liste kritičnih aktivnosti, ukoliko se neke od njih ispostave kao kritične (a najmanje jedna hoće. Zašto?).

8. OSNOVNI METODI PRETRAŽIVANJA

Zadatak 8.1

Data je jednostruko ulančana lista bez zaglavlja. Skicirati algoritam sekvencijalnog pretraživanja liste za ključ k, ako se koristi graničnik. Koja je mana pristupa? Predložiti način(e) poboljšanja.

Rešenje:

Graničnik je element liste ekvivalentan svim ostalim, koji sadrži ključ koji se traži (k) i dodaje se na kraj liste. Lista se sekvencijalno pretražuje ali se vrši samo jedno testiranje – testiranje na vrednost ključa jer on sigurno postoji u listi (nema potrebe testirati da li se došlo do kraja liste). Ako se traženi ključ nađe pre graničnika, to znači da je postojao u listi. U suprotnom nije postojao.

```
LIST-SEQ-SEARCH-SENT(L, key)
addNode(L, key)
tekuci = L.first
while (tekuci.key ≠ key) do
        tekuci = tekuci.next
end_while
if (tekuci = L.last) then
        tekuci = NIL
end_if
removeLastNode(L)
return tekuci
```

Mana postupka: uklanjanje graničnog elementa liste nakon završene pretrage – mora da se obavi još jedan sekvencijalni prolaz kroz listu.

Moguća poboljšanja:

- koristiti dvostruko ulančanu listu
- modifikovati kod tako da se zapamti adresa poslednjeg elementa liste pre dodavanja graničnika, da bi se uklanjanje graničnika moglo obaviti bez novog sekvencijalnog prolaza kroz listu.

Zadatak 8.2

U niz se redom dodaju sledeći ključevi: 1, 2, 8, 12, 4, 3, 5. Zatim se redom dohvataju sledeći ključevi: 12, 2, 5, 12, 8, 5, 2, 12. Odrediti prosečnu dužinu uspešnog traženja ključa ako se koriste metode **prebacivanja na početak** i **transpozicije**.

Rešenje:

Kada verovatnoća pretrage nije ista za sve ključeve efikasnije je pomeriti ključeve sa većom verovatnoćom bliže početku niza.

Kada je verovatnoća uniformno raspodeljena, ali je vremenska lokalnost obraćanja ključevima takva da je velika verovatnoća da će se tokom neke skorije pretrage opet tražiti trenutni ključ, moguće je primeniti tehnike ubrzanja pretrage:

- **prebacivanje na početak**: vrši premeštanje elementa na čijoj poziciji je pronađen odgovarajući ključ na početak niza. Elementi koji mu prethode se pomeraju za jedno mesto u desno.
- **transpozicija**: element na čijoj poziciji je pronađen ključ i njegov prethodni zamenjuju mesto

Prebacivanje na početak

Ključ:	1	2	8	12	4	3	5	Dužina pretrage
12	**12**	**1**	**2**	**8**	4	3	5	4
2	**2**	**12**	1	8	4	3	5	3
5	**5**	**2**	**12**	**1**	**8**	**4**	**3**	7
12	**12**	**5**	**2**	1	8	4	3	3
8	**8**	**12**	**5**	**2**	**1**	4	3	5
5	**5**	**8**	**12**	2	1	4	3	3
2	**2**	**5**	**8**	**12**	1	4	3	4
12	**12**	**2**	**5**	**8**	1	4	3	4
							Prosečna dužina pretrage:	4.125

Transpozicija:

Ključ:	1	2	8	12	4	3	5	Dužina pretrage:
12	1	2	**12**	**8**	4	3	5	4
2	**2**	**1**	12	8	4	3	5	2
5	2	1	12	8	4	**5**	**3**	7
12	2	**12**	**1**	8	4	5	3	3
8	2	12	**8**	**1**	4	5	3	4
5	2	12	8	1	**5**	**4**	3	6
2	2	12	8	1	5	4	3	1
12	**12**	**2**	8	1	5	4	3	2
							Prosečna dužina pretrage:	3.625

Diskusija
Šta je bolje: prebacivanje na početak ili transpozicija?

Transpozicija je *inertnija* i zbog toga manje podložna gubitku na performansama zbog sporadičnih pretraga na ključeve koji se inače retko traže.

Simulacije su pokazale:
- u startu bolje prebacivanje na početak
- dugoročno gledano, bolja transpozicija

Zato se ponekad koristi kombinacija ove dve metode, najpre prebacivanje na početak, kasnije transpozicija. Prebacivanje na početak je pogodnije za ulančanu listu, a manje pogodno za nizove, zbog linearne složenosti operacije pomeranja elemenata.

U nastavku je dat program na programskom jeziku C koji realizuje pretragu niza uz korišćenje metoda transpozicije i prebacivanja na početak. Izabrana metoda se dostavlja kao parametar (u vidu pokazivača na funkciju) funkciji koja vrši pretragu niza. Studentima se preporučuje da isti program za vežbu napišu na nekom objektno-orijentisanom jeziku primenjujući konstrukcije primerene izabranom jeziku.

```
/* prebacuje kljuc sa pozicije k na pocetak niza */
void prebaci_na_pocetak(int niz[], int k)
{
int i, t = niz[k];

      for(i = k; i > 0; i--)
            niz[i] = niz[i-1];

      niz[0] = t;
}

/* vrsi transpoziciju kljuca na poziciji k sa prethodnikom */
void transpozicija(int niz[], int k)
{
      if( k > 0 )
      {
      int t = niz[k];
            niz[k] = niz[k-1];
            niz[k-1] = t;
      }
}


/* pronalazi kljuc u nizu; vraca broj obavljenih pristupa nizu
   ili 0 ako kljuc ne postoji u nizu */
int nadji(int niz[], int n, int kljuc )
{
int i;

      for(i = 0; i < n; i++)
            if( niz[i] == kljuc )
                  return i+1;

      return 0;
}


#include <stdio.h>
#include <stdlib.h>


/* ispisuje sadrzaj niza */
void ispis_niza(int niz[], int n)
{
int i;
      for(i = 0; i < n; i++)
            printf("%4d ", niz[i]);
      printf("\n");
}

typedef void obrada(int [], int);
```

```
/* vrsi pretragu nad zadatim nizom na neke kljuceve primenom zadate metode */
double pretraga(int niz[], int n, int kljucevi[], int br_kljuceva, obrada *o)
{
int i, uk_br_pristupa, br_nadjenih;

      for(i = br_nadjenih = uk_br_pristupa = 0; i < br_kljuceva; i++)
      {
      int t = nadji(niz, n, kljucevi[i]);
            printf("%d\t", kljucevi[i]);
            if( t )
            {
                  o(niz, t-1);
                  ispis_niza(niz, n);
                  uk_br_pristupa += t, br_nadjenih++;
            }
            else
                  printf("Kljuc %d ne postoji u nizu.\n", kljucevi[i]);
      }
      if( br_nadjenih ) return (double)uk_br_pristupa / br_nadjenih;
      else              return 0;
}

void main()
{
int *niz1, *niz2, *kljucevi, duzina_niza, broj_kljuceva, i;
double rezultat;

    printf("Broj elemenata niza? :"); scanf("%d", &duzina_niza);
    printf("Broj kljuceva za pretragu? :"); scanf("%d", &broj_kljuceva);

    niz1 = malloc( duzina_niza * sizeof(int) );
    niz2 = malloc( duzina_niza * sizeof(int) );
    kljucevi = malloc( broj_kljuceva * sizeof(int) );
    if( ! niz1 || ! niz2 || ! kljucevi )
    {
          printf("Greska u alokaciji memorije.\nProgram se prekida.\n");
          exit(0);
    }

    printf("Elementi niza? :");
    for(i = 0; i < duzina_niza; i++) scanf("%d", &niz1[i]), niz2[i] = niz1[i];
    printf("Kljucevi? :");
    for(i = 0; i < broj_kljuceva; i++) scanf("%d", &kljucevi[i]);

    printf("Transpozicija:\n\t");
    ispis_niza(niz1, duzina_niza);
    rezultat = pretraga(niz1, duzina_niza, kljucevi, broj_kljuceva, transpozicija);
    printf("Prosecan broj pristupa: %.3lf\n\n\n", rezultat );

    printf("Prebacivanje na pocetak:\n\t");
    ispis_niza(niz2, duzina_niza);
    rezultat = pretraga(niz2, duzina_niza, kljucevi, broj_kljuceva, prebaci_na_pocetak);
    printf("Prosecan broj pristupa: %.3lf\n\n\n", rezultat );

    free(kljucevi);
    free(niz1);
    free(niz2);
}
```

Zadatak 8.3

Skicirati algoritam binarne pretrage. Koji su preduslovi za primenu ovog algoritma? Odrediti njegovu vremensku složenost.

Rešenje:

```
BIN-SEARCH(K, key)
low =1
high = n
while (low ≤ high) do
        mid = (low + high)/2
        if (key = K[mid]) then
                return mid
        else if (key < K[mid]) then
                        high = mid - 1
                else
                        low = mid + 1
                end_if
        end_if
end_while
return 0
```

Preduslovi:

1. vrši se nad linearno uređenom strukturom kod koje je moguć **direktan pristup proizvoljnom elementu** (niz)
2. elementi moraju biti **uređeni** (neopadajuće/nerastuće)

Vremenska složenost: O(log n)

Primer: pretraga na ključ 67

2	4	5	7	21	24	29	32	51	62	67	81	83	88	92
low							mid							high

2	4	5	7	21	24	29	32	51	62	67	81	83	88	92
								low			mid			high

2	4	5	7	21	24	29	32	51	62	67	81	83	88	92
								low	mid	high				

2	4	5	7	21	24	29	32	51	62	67	81	83	88	92
										low high mid				

Zadatak 8.4

Data je **uređena tabela** koja redom sadrži ključeve 3, 5, 9, 11, 14, 21, 27, 35, 42, 47, 57, 61, 73, 88, 99. Predložiti strukturu podataka za **pretragu na više ključeva nad uređenom tabelom** i primerom ilustrovati njenu potrebu.

Porediti efikasnost upotrebe predložene strukture i višestruke binarne pretrage nad datom tabelom, ako se traže ključevi 11, 27, 42, 48

Rešenje:

Predložena struktura podataka: vektor dužine n (n = broj ključeva za koje se vrši pretraga). Potprogramu se kao argument prosleđuje **vektor A** sa vrednostima ključeva na koje se vrši pretraga. Kao rezultat, potprogram vraća **vektor** indeksa **B** dužine n. Ako u tabeli postoji ključ A[i], onda se u B[i] nalazi redni broj ulaza u tabeli u kojem je pronađen dati ključ. U suprotnom, B[i] ima vrednost 0.

```
SEQ-SEARCH-MUL(K, S)
for i = 1 to m do
        P[i] = 0
end_for
i = j = 1
while (i ≤ n) and (j ≤ m) do
        while (i < n) and (S[j] > K[i]) do
                i = i + 1
        end_while
        if (S[j] = K[i]) then
                P[j] = i
                i = i+1
        end_if
        j = j + 1
end_while
```

Pretpostavka: ključevi koji se traže su uređeni (tj. vektor A je uređen). U suprotnom pretraga na više ključeva ne daje bolje rezultate nego višestruka pojedinačna pretraga.

Nakon nalaženja jednog ključa, od date pozicije se produžava sa traženjem narednog. Zbog uređenosti tabele i vektora A, sigurno je da se ključ A[i+1] ne može pojaviti pre ključa A[i].

Ideja: izbeći višestruku pretragu niza koristeći informaciju od prethodne pretrage. Drugim rečima prilikom pretrage na ključ A[i+1] koristi se informacija dobijena od pretrage na ključ A[i].

3	5	9	**11**	14	21	**27**	35	**42**	47	57	61	73	88	99

Primenom predložene strukture se vrši 15 poređenja na vrednost ključa. Kao rezultat vraća se vektor (4,7,9,0).

Višestrukom primenom metode birnarnog traženja, broj poređenja na vrednost ključa je sledeći:

Ključ:	11	27	42	48	Ukupno:
Poređenja:	3	7	7	8	25

Za vežbu:

Kako bi se izvršilo poboljšanje metode binarne pretrage na više ključeva ?

Kako izvesti pretragu na više ključeva nad neuređenom tabelom?

Zadatak 8.5 (Septembar 2009)

Date su dve duži definisane svojim temenima u Dekartovoj (2D) ravni. Poznato je da se duži seku. Koristeći ideju algoritma binarne pretrage kao osnovu, napisati funkciju na jeziku C (ili C++) koja iterativnim postupkom pronalazi tačku preseka zadatih duži. Broj koraka iteracije se zadaje kao parametar funkcije. Koristiti sledeće deklaracije tipova:

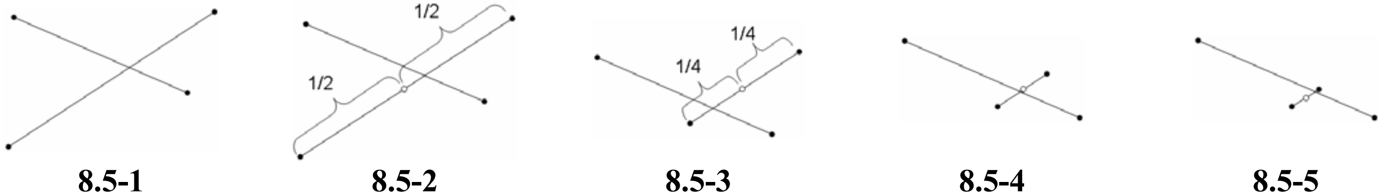
```
typedef struct { double x, y; } Tacka;
typedef struct { Tacka t1, t2; } Duz;
```

Na raspolaganju je sledeća pomoćna funkcija koju ne treba pisati, a koja određuje da li duž definisana tačkama P i Q seče duž D. Funkcija vraća vrednost 0 ako postoji tačka preseka, a 1 u suprotnom.

```
istaStrana( Tacka P, Tacka Q, Duz D);
```

Rešenje:

Primer koji ilustruje postupak je prikazan na sledećim slikama (8.5-1 do 8.5-5).



8.5-1 **8.5-2** **8.5-3** **8.5-4** **8.5-5**

Slika 8.5-1 prikazuje početno stanje. Jedna od dve duži (u primeru duž koja se prostire od donjeg levog do gornjeg desnog ugla na slikama) je odabrana za sužavanje ka tački preseka. Pridružimo temena te duži oznake *low* i *hi*. Primetiti da nije bitno koje teme nosi koju oznaku (oznake namerno nisu prikazane na slici). Postupak je iterativan: u svakom koraku se preostala dužina duži polovi tako što se uoči tačka u oznaci *mid* na sredini duži (nepopunjena tačka na slikama 8.5-2 do 8.5-5) a zatim utvrđuje da li su tačke *low* i *mid* sa iste strane druge duži, sa kojom se traži presek. U slučaju da jesu, presek se nalazi između *mid* i *hi*, a u suprotnom između *low* i *mid*. U zavisnosti od ishoda prethodnog utvrđivanja strana, *low* ili *hi* će se premestiti u tačku *mid*, nakon čega se završava tekuća iteracija i počinje nova.

```
typedef struct { double x, y; } Tacka;
typedef struct { Tacka t1, t2; } Duz;
int istaStrana( Tacka P, Tacka Q, Duz D);

Tacka sredina(Tacka t1, Tacka t2) {
Tacka rezultat;
      rezultat.x = (t1.x + t2.x)/2;
      rezultat.y = (t1.y + t2.y)/2;
      return rezultat;
}
```

```
Tacka presek(Duz d1, Duz d2, int n) {
int i;
Tacka low = d1.t1, hi = d1.t2, mid;
  for(i = 0; i < n; i++) {
    mid = sredina(low, hi);
    if( istaStrana(mid, low, d2) )   low = mid;
    else hi = t;
  }
  return t;
}
```

Primetiti razlike u odnosu na osnovni algoritam binarne pretrage niza:

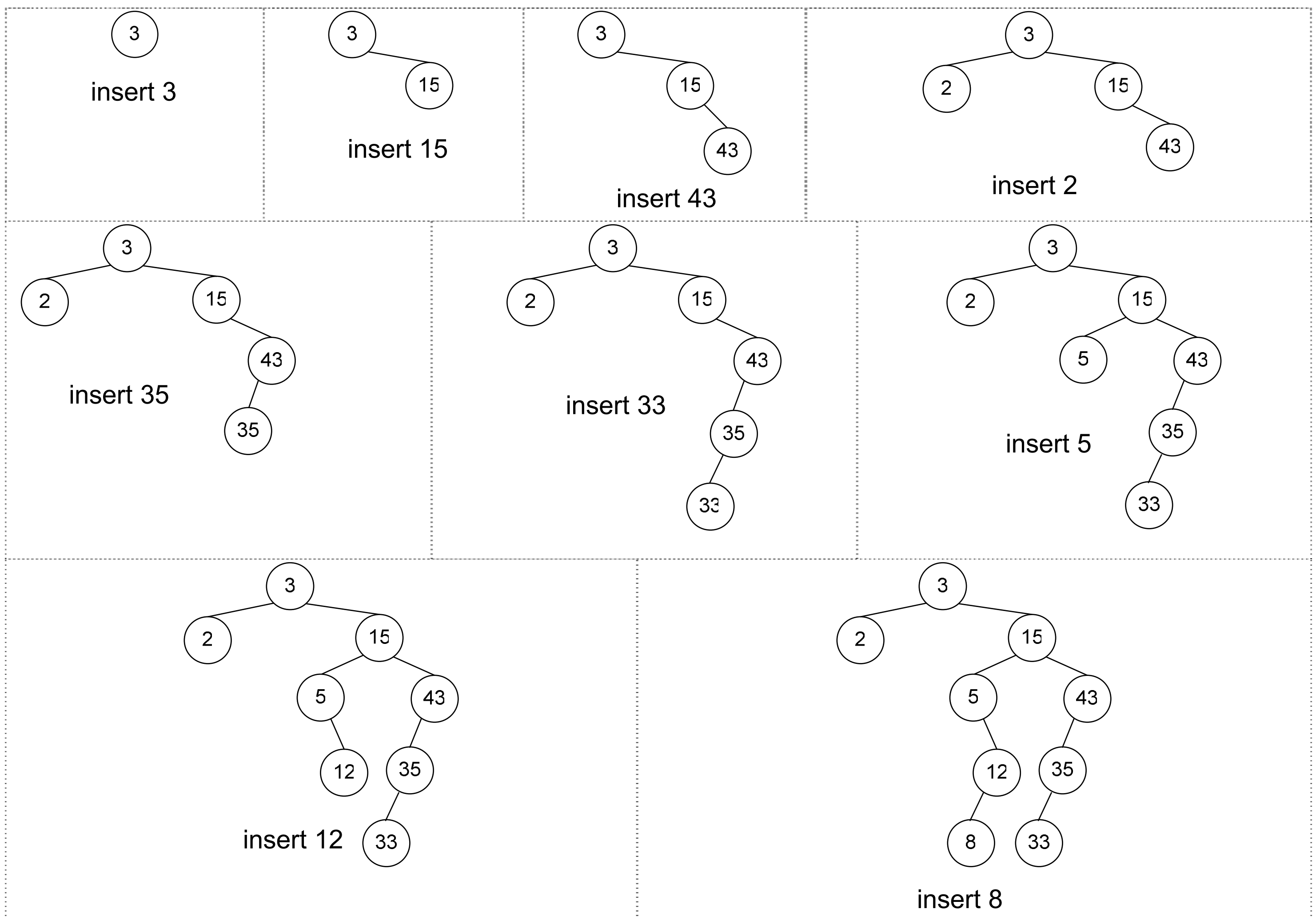
1. obrada se prekida nakon zadatog broja iteracija, jer ne postoji način da se utvrdi da li tačka *mid* odgovara traženom preseku (preciznije – način postoji, na primer analitički, ali bi tada tačka preseka bila poznata pa je čitav postupak suvišan). Ako se očekuje određivanje tačke preseka sa greškom manjom od zadate vrednosti, moguće je postupiti na sledeći način: s obzirom na to da se dužina izabrane duži polovi u svakoj iteraciji, na osnovu njene početne dužine može se odrediti koliko je iteracija potrebno da bi greška aproksimacije bila u zadatim granicama. Alternativno, moguće je određivati razdaljinu za uzastopna određivanja pozicije tačke mid i obustaviti postupak kada se ustanovi da je razdaljina manja od zadate vrednosti.
2. tačke *low* i *hi* se postavljaju na *mid*, a ne na *mid*+1 odnosno *mid*-1, respektivno, nakon utvrđivanja u kojoj oblasti se nalazi tačka preseka. Dodavanje 1 ili -1 bi bilo logički pogrešno: u ovom slučaju radi se o tačkama u ravni – šta predstavlja dodavanje skalarne vrednosti dvodimenzionom vektoru?
3. tačka preseka se uvek može naći (u granicama u kojima to preciznost realnih brojeva dozvoljava)

9. STABLA BINARNOG PRETRAŽIVANJA

Zadatak 9.1

U binarno stablo pretraživanja umetnuti redom čvorove sa vrednostima ključeva 3, 15, 43, 2, 35, 33, 5, 12, 8. Prikazati izgled stabla nakon svakog umetanja. Kako bi izgledao redosled ispisivanja vrednosti ključeva, ako se stablo obiđe u inorder poretku? Navesti prednosti stabla binarnog pretraživanja nad metodama pretraživanja vektora (linearnim i nelinearnim).

Rešenje:



Inorder obilaskom stabla binarnog pretraživanja bi se uvek dobili ključevi uređeni po neopadajućem poretku.

Prednosti stabla:

- jednostavno umetanje i brisanje ključeva
- relativno mala vremenska složenost: O(h) – u najboljem slučaju O(log n), u najgorem O(n)

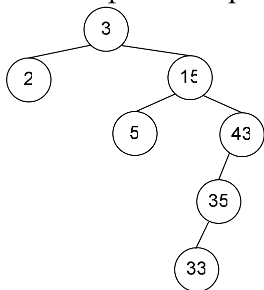
Zadatak 9.2

Navesti i ilustrovati sve slučajeve brisanja čvorova iz binarnog stabla pretraživanja.
Iz binarnog stablo pretraživanja iz prethodnog zadatka, nakon umetanja svih čvorova, redom se brišu čvorovi 8, 15, 5. Nacrtati izgled stabla nakon svakog brisanja.

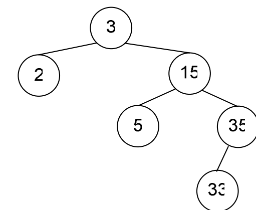
Rešenje:
Mogući slučajevi brisanja čvorova iz stabla binarnog pretraživanja:

- čvor koji se briše nema potomke (list):
  ⇨odgovarajući pokazivač roditeljskog čvora postaje NIL

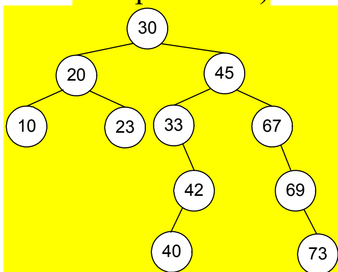
- čvor koji se briše ima samo jednog direktnog potomka:
  ⇨direktni potomak preuzima mesto čvora koji se briše



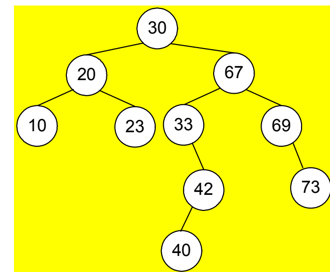
DELETE 43



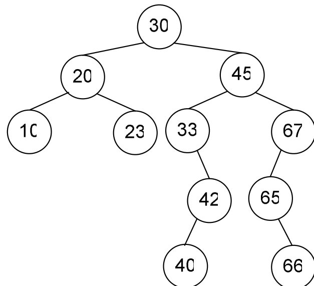
- čvor koji se briše ima oba sina:
  ⇨ pronalazi se sledbenik S čvora koji se briše
  - čvor koji se briše je roditelj čvora S: čvor S zauzima mesto svog roditelja (preuzima njegovo levo podstablo)



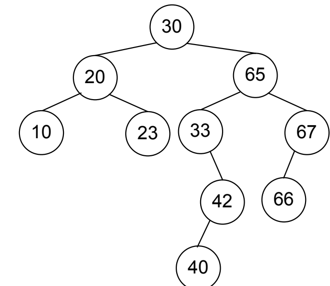
DELETE 45
(sledbenik je 67)

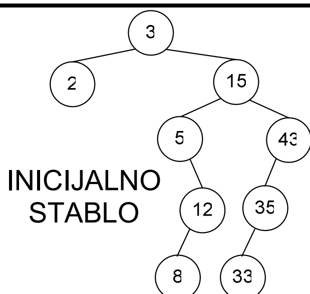


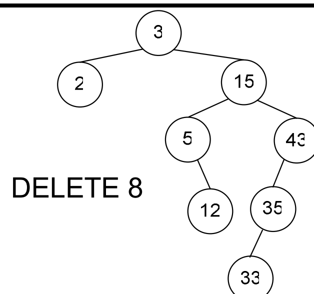
  - čvor koji se briše nije roditelj čvora S: eventualni desni potomak čvora S postaje levi potomak čvora roditelja čvora S, a S zauzima mesto čvora koji se briše.
    Zašto čvor S ne može imati levog potomka?

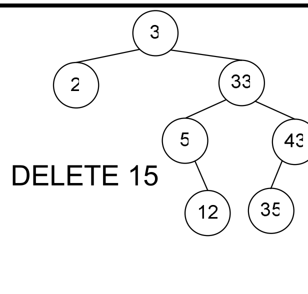


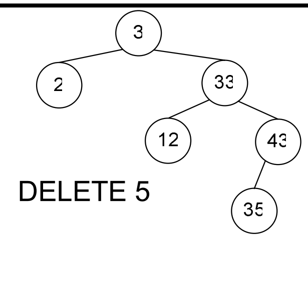
DELETE 45
(sledbenik je 65)











Zadatak 9.3

U binarno stablo pretraživanja se najpre redom umeću ključevi 15, 25, 18, 31, 16, 2, 12, 14, 8, 22, 23, 28, 27. Zatim se redom brišu ključevi 2, 25, 15, 16, 18, 22, 23. Prikazati izgled stabla nakon svakog umetanja i brisanja.

Rešenje:

NAKON UMETANJA

DELETE 2

DELETE 25

DELETE 15

DELETE 16

DELETE 18

DELETE 22

DELETE 23

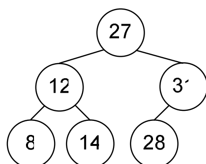
Zadatak 9.4

Binarno stablo pretraživanja B se dobija tako što se u njega redom umeću ključevi dobijeni inorder obilaskom binarnog stabla pretraživanja A. Komentarisati osobine stabla B (kompletnost, balansiranost). Proceniti vremensku složenost pretrage proizvoljnog ključa u stablu B. Predložiti redosled umetanja ključeva u stablo B da bi se poboljšale performanse pretrage.

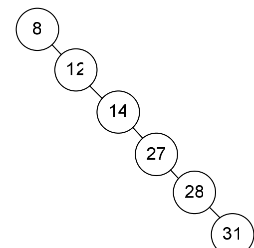
Rešenje:

Inorder obilaskom stabla se pristupa ključevima u neopadajućem redosledu.

Umetanje ključeva datim redosledom bi napravilo nebalansirano stablo kod kojeg svaki čvor (izuzev lista) ima samo desnog potomka – svodi se na ulančanu listu.

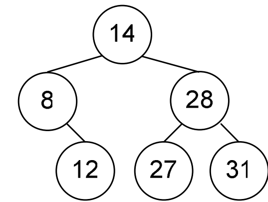


8 - 12 - 14 - 27 - 28 - 31



Binarno stablo pretraživanja ima najbolje performanse kada je balansirano. Balansiranost bi se postigla ako bi se najpre izvršilo umetanje ključa na sredini niza ključeva dobijenog *inorder* obilaskom, a zatim ključeva na polovini intervala između početka i sredine niza, odnosno sredine i kraja niza, itd.

Ključ	8	12	14	27	28	31
Redni broj	2	4	1	5	3	6



Zadatak 9.4

Za dati programski segment prikazati izgled steka i stanje tabele simbola u trenutku *1* i *2*

```
main()
{
int a,i,j;
      for(i = 0; i < 10; i++)
      {
      char odgovor = 'n';

            while(odgovor != 'd' && odgovor != 'n' )
                  scanf("%c", &odgovor);
            *1*
            for(j = i; j >= 0; j--)
            {
            double b, a = i-j * 10, c;
                  if( odgovor == 'd' )    b = a-5;
                  else b = a;
            *2*
            }
      }
}
```

Rešenje:

Prilikom svakog započinjanja novog bloka instrukcija, na stek se smešta redni broj ulaza u tabeli simbola odakle počinje smeštanje lokalnih simbola (definisanih u tom bloku). Tako se na vrhu steka uvek nalazi indeks ulaza tabele simbola lokalnih za tekući blok.

Novi simbol se umeće u tabelu na prvoj slobodnoj lokaciji. Međutim, radi efikasne pretrage tabele simbola, vrši se logičko povezivanje ulaza tabele u vidu binarnog stabla. Zbog toga, svaki ulaz tabele sadrži dva celobrojna podatka (indeksa) koji služe za pristupanje onim ulazima tabele simbola gde se nalaze simboli definisani u tekućem bloku a koji leksikografski prethode odnosno slede datom simbolu. Po uzoru na binarno stablo pretraživanja, levim indeksom se pristupa simbolima koji leksikografski prethode a desnim onima koji slede datom simbolu. Vrednost 0 označava nevalidan indeks (nema podstabla).

IZGLED U TAČKI *1*

STEK
4
1

TABELA SIMBOLA

1	0	a	2
2	0	i	3
3	0	j	0
4	0	odgovor	0
5	0	?	0
6	0	?	0
7	0	?	0

IZGLED U TAČKI *2*

STEK
5
4
1

TABELA SIMBOLA

1	0	a	2
2	0	i	3
3	0	j	0
4	0	odgovor	0
5	6	b	7
6	0	a	0
7	0	c	0

AVL Stabla

Zadatak 9.5

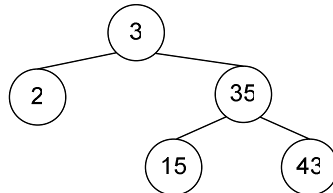
Navesti nedostatke nebalansiranih stabala pretraživanja. Kakva su to AVL stabla? Navesti kriterijum balansiranosti AVL stabala. Po čemu se taj kriterijum razlikuje od opšteg kriterijuma balansiranosti stabala? Prikazati osnovne transformacije stabla radi održavanja balansiranosti prilikom umetanja elemenata.

Rešenje:
Glavni nedostatak opštih binarnih stabala: umetanja i brisanja čvorova mogu dovesti do debalansiranosti stabla, time se smanjuje efikasnost osnovnih operacija nad stablom (pretraga, umetanje, brisanje).

AVL stablo je uvedeno 1962. godine (autori: Georgy Maximovich **A**delson-**V**elsky (8.1.1922.–) i Yevgeniy Mikhailovich **L**andis (6.10.1921–12.12.1997)). AVL stablo predstavlja nadgradnju jednostavnog binarnog stabla pretraživanja: nakon svake operacije koja pravi izmene u stablu (umetanje, brisanje) proverava se da li je došlo do debalansa u čvorovima u stablu nakon čega se po potrebi vrši rebalansiranje stabla.

Kriterijum balansiranosti AVL stabla: razlika u visinama levog i desnog podstabla svakog čvora sme biti najviše 1 (tzv. visinsko balansiranje stabla).
Opšti kriterijum balansiranosti binarnog stabla: razlika broja čvorova u levom i desnom podstablu svakog čvora može biti najviše 1.

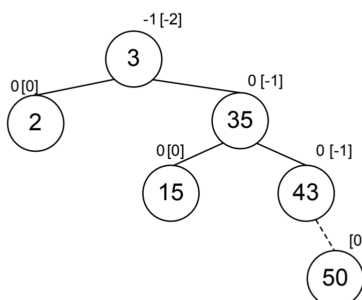
Primer stabla koje je nebalansirano prema opštem kriterijumu, a balansirano prema AVL kriterijumu:



Potreba za rebalansiranjem stabla se javlja kod umetanja čvora, kada je neki njegov predak bio nagnut pre umetanja (**kritičan čvor**) i čije nagnuće nakon umetanja prelazi dozvoljenu granicu. Umetnuti čvor je, kao i kod stabla binarnog pretraživanja, list.
Razlikuju se sledeća dva slučaja ažuriranja stabla nakon umetanja:

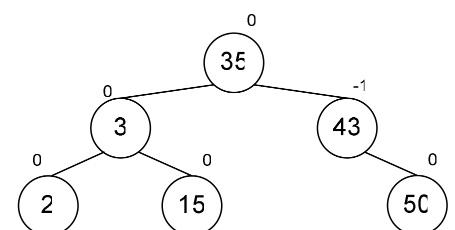
1. Sin kritičnog čvora naginje na istu stranu kao i kritični čvor (potrebna rotacija ulevo ili udesno)
2. Nakon umetanja lista, sin kritičnog čvora naginje na suprotnu stranu od kritičnog čvora (potrebna dvostruka rotacija, u različitim smerovima)

Primer za prvi slučaj: u stablo se dodaje čvor 50. Balans svakog čvora prikazan je iznad simbola čvora. U uglastim zagradama je balans nakon umetanja. Nakon umetanja, čvor 3 postaje kritičan (balans -2). On naginje udesno, kao i njegov desni potomak, pa je dovoljna rotacija ulevo oko čvora 3 da bi se stablo rebalansiralo.



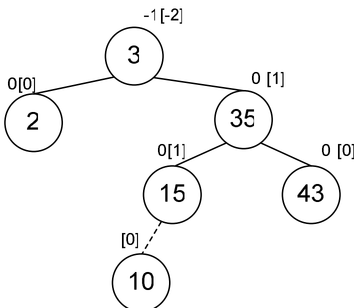
Balans čvorova nakon umetanja čvora 50, ali pre rebalansiranja.

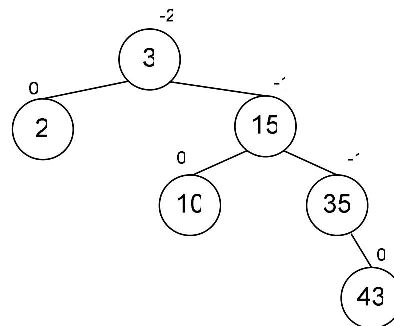
<u>LEFT-ROTATION(*x*)</u>
y = right(*x*)
temp = *left*(*y*)
left(*y*) = *x*
right(*x*) = *temp*

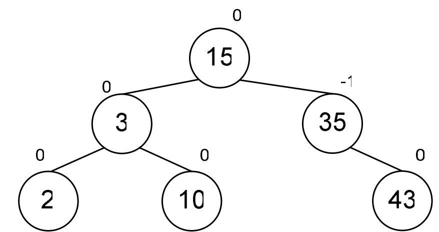


Izgled stabla nakon rebalansiranja

Primer za drugi slučaj: u stablo se dodaje čvor 10. Nakon umetanja, čvor 3 postaje kritičan (balans -2). On naginje udesno, ali njegov desni potomak, čvor 35, naginje ulevo, pa je potrebna dvostruka rotacija, najpre udesno oko čvora 35, da bi i on naginjao udesno kao njegov roditelj. Nakon ove rotacije, stablo je dovedeno u istu situaciju kao u slučaju 1, pa se onda vrši rotacija ulevo oko čvora 3 da bi se stablo konačno rebalansiralo.


Balans čvorova nakon umetanja čvora 10, ali pre rebalansiranja.


Izgled stabla nakon rotacije udesno oko čvora 35.

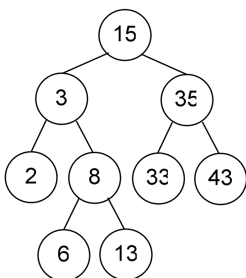

Izgled stabla nakon rebalansiranja

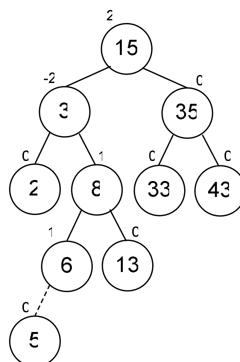
Zadatak 9.6

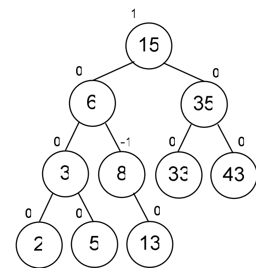
Prikazati izgled visinski balansiranog (AVL) stabla pretraživanja u koje se redom umeću vrednosti 3, 15, 43, 2, 35, 33, 8, 6, 13, 5, 7, 14 nakon svakog umetanja.

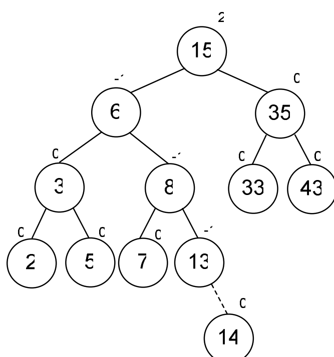
Rešenje:

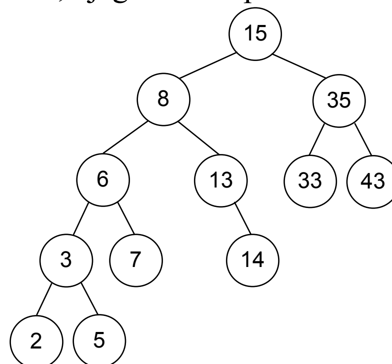
U slučaju da ne dođe do debalansa prilikom umetanja novog čvora, proces umetanja se vrši kao kod običnog stabla binarnog pretraživanja. To je slučaj sa umetanjem svih ključeva do ključa 5. Izgled stabla u tom trenutku je prikazan na prvoj slici. Umetanje ključa 5 zahteva rebalansiranje stabla.

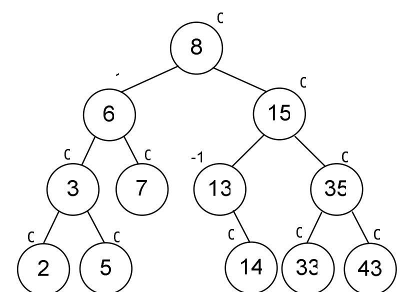

Izgled stabla nakon umetanja ključeva 3-13


Umetanjem ključa 5 dolazi do debalansa stabla: čvor 3 naginje udesno, njegov desni potomak ulevo.


Nakon rebalansiranja stabla ne postoji nijedan čvor sa nedozvoljenim balansom.


Izgled stabla nakon umetanja čvora 7. Umetanje čvora 14 će izazvati debalans stabla.

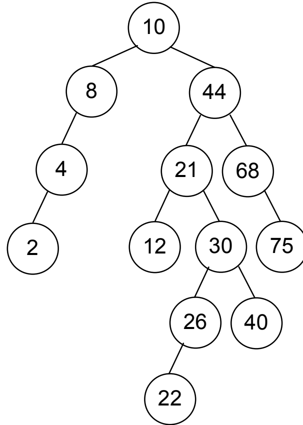

Izgled stabla nakon rotacije ulevo oko čvora 6. Sada čvor 15 (kritičan čvor) i njegov levi potomak naginju ulevo.


Izgled stabla nakon završene dvostruke rotacije, najpre oko čvora 6, zatim oko čvora 15.

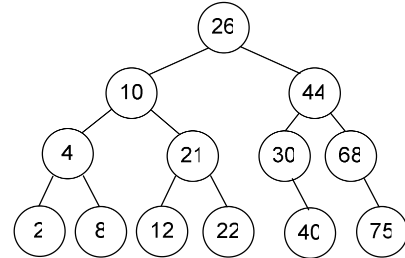
Zadatak 9.7 – Zadatak za samostalnu vežbu

U stablo binarnog pretraživanja se redom umeću ključevi: 10, 8, 44, 4, 21, 68, 2, 12, 30, 75, 26, 40, 22. Izvesti visinsko balansiranje datog binarnog stabla primenom metoda za balansiranje AVL stabla. Da li je tako dobijeno stablo AVL stablo?

Rešenje:



Izgled stabla binarnog pretraživanja nakon umetanja svih ključeva.



Izgled rezultujućeg stabla nakon primene svih transformacija rebalansiranja stabla.

10. STABLA OPŠTEG PRETRAŽIVANJA

10.1 B stabla

Zadatak 10.1.1

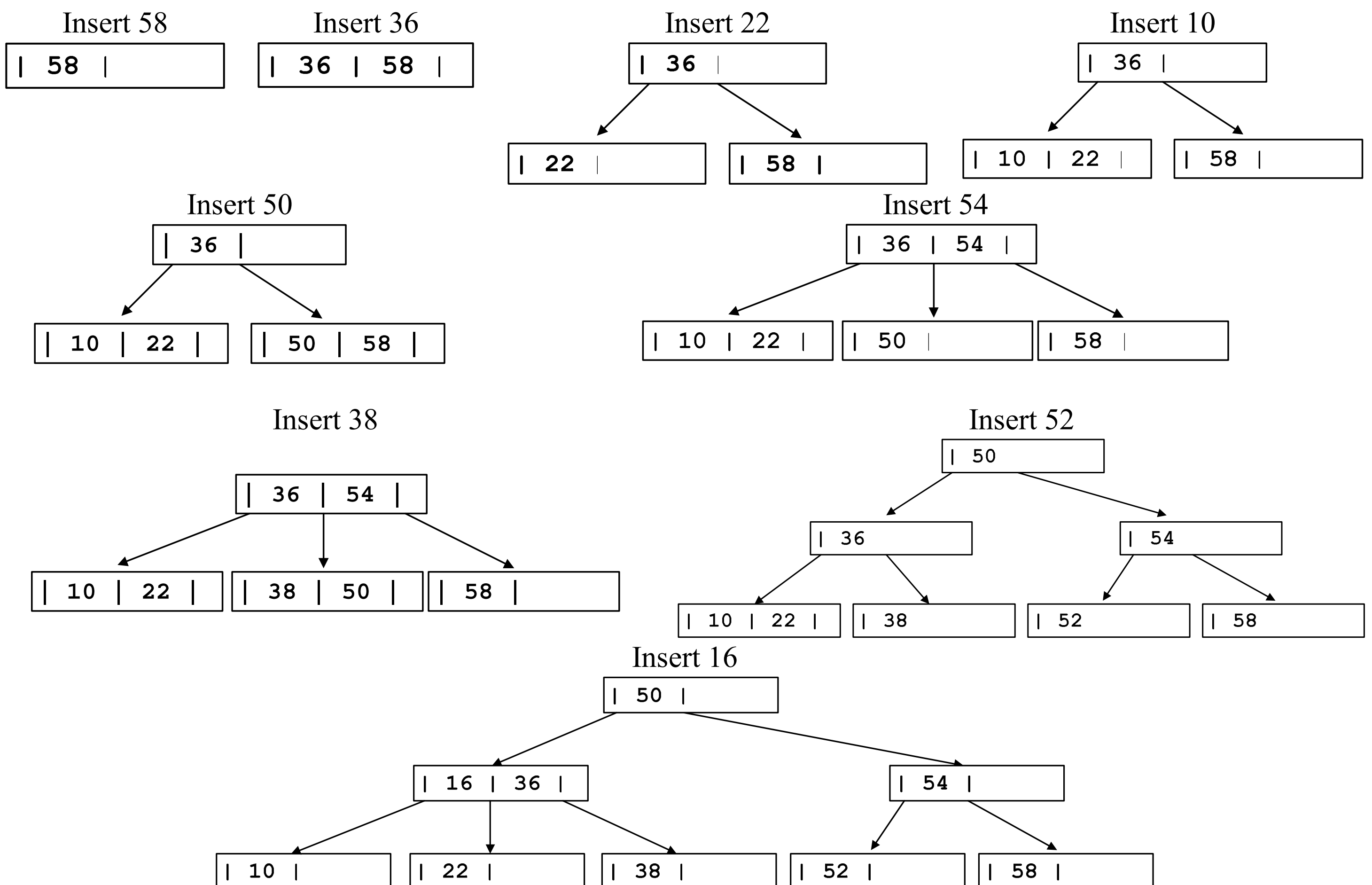
U inicijalno prazno B-stablo reda 3 umeću se redom ključevi 58, 36, 22, 10, 50, 54, 38, 52, 16, 40, 42, 12, 6, 27, 21 a zatim se redom brišu ključevi 50, 42, 38, 40, 52.

a) Nacrtati izgled stabla nakon svake od navedenih izmena.
b) Koliki je srednji broj pristupa prilikom uspešnog i neuspešnog traženja, kao i popunjenost B stabla, posle svih umetanja ključeva i u završnom stanju?

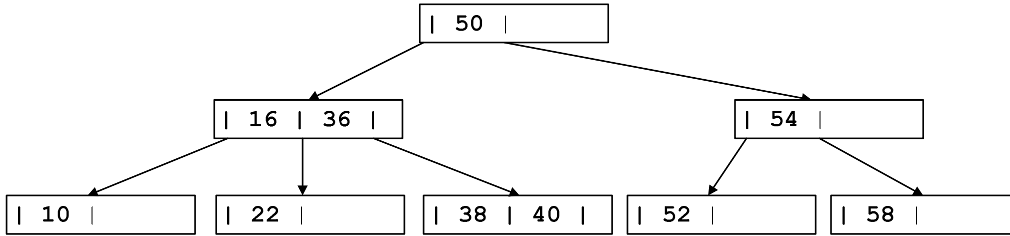
Rešenje:

m=3

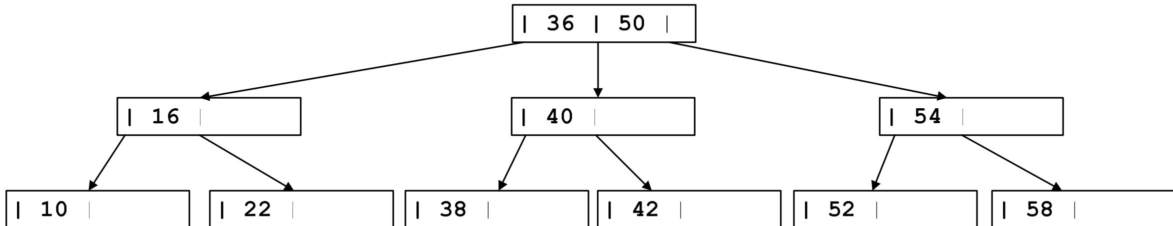
- svi čvorovi osim korena i listova imaju najmanje $\lceil \frac{3}{2} \rceil = 2$ podstabla
- listovi imaju najmanje $\lceil \frac{3}{2} \rceil - 1 = 1$ ključeva.



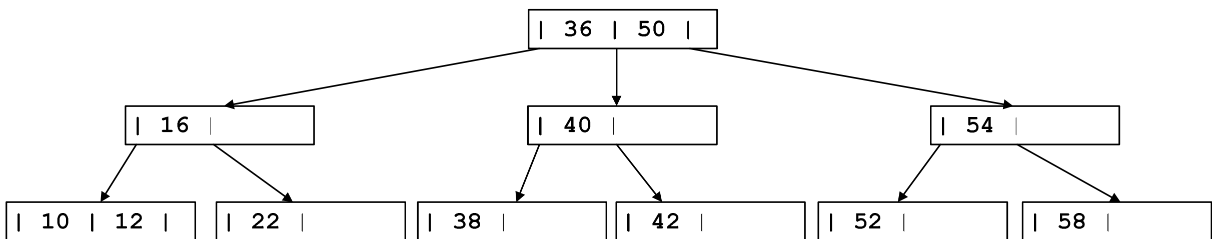
Insert 40



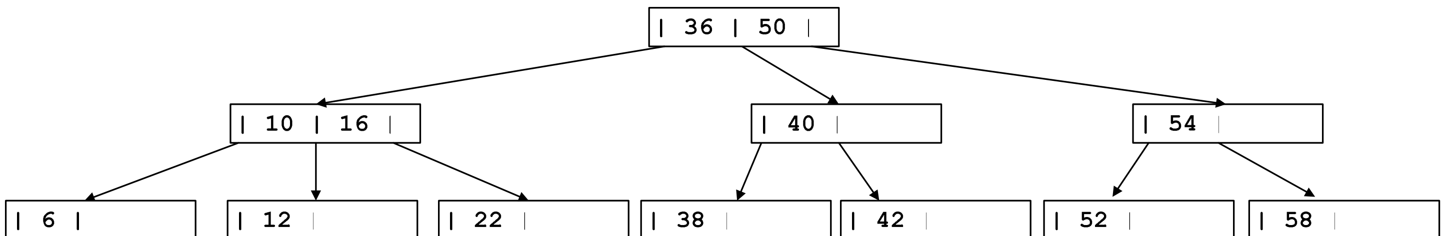
Insert 42



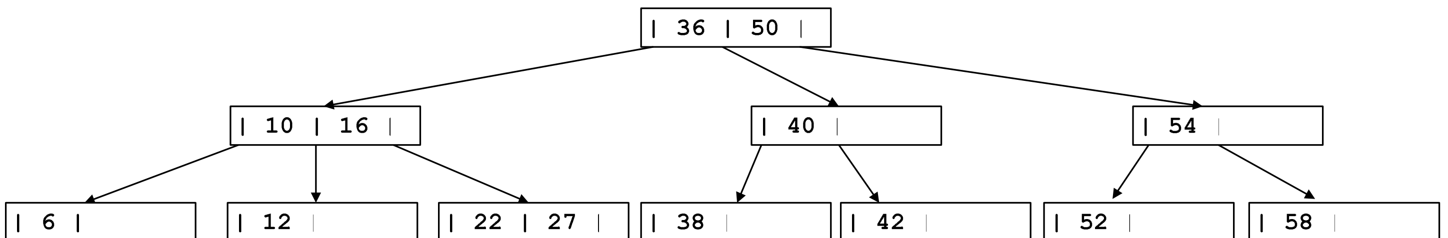
Insert 12



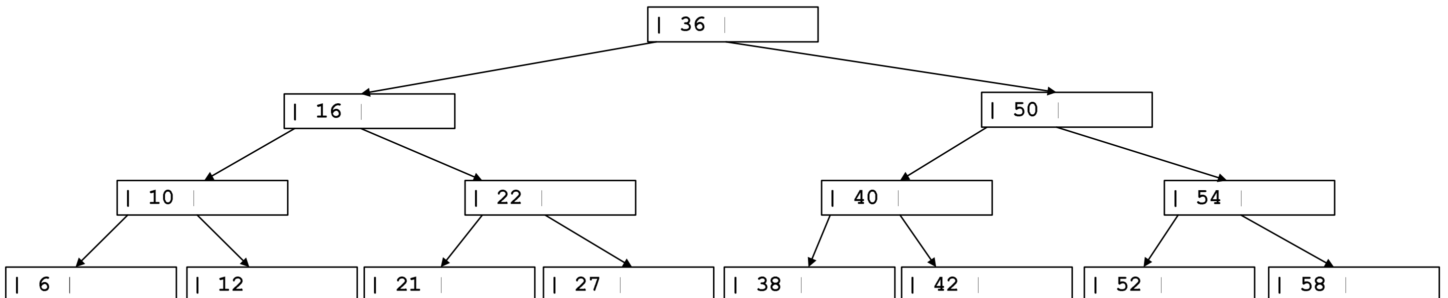
Insert 6



Insert 27

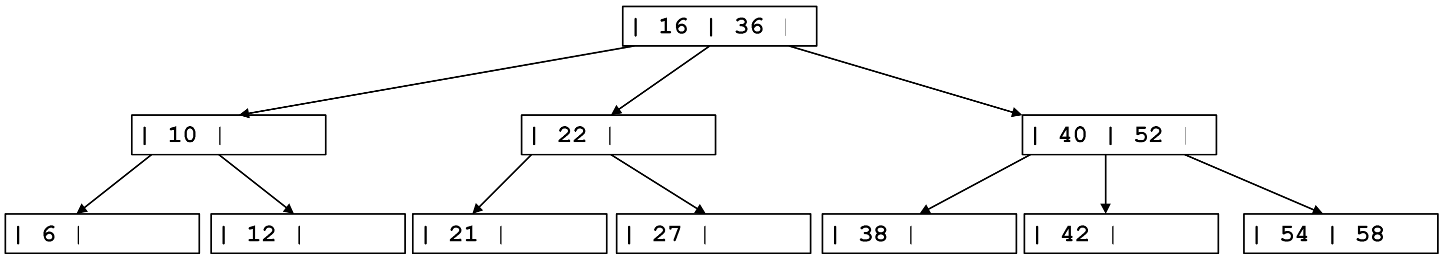


Insert 21

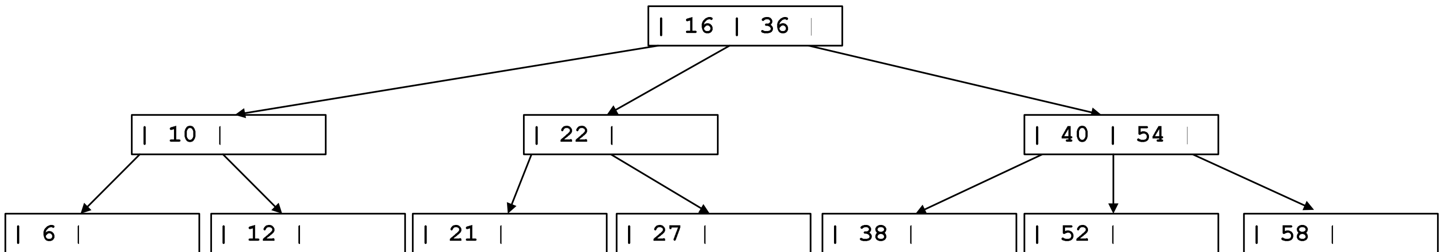


Popunjenost stabla: 0.5 (50%)	Broj pristupa: - Neuspešno traženje: 4 - Uspešno traženje: (1*1 + 2*2 + 4*3 + 8*4)/15=3.267

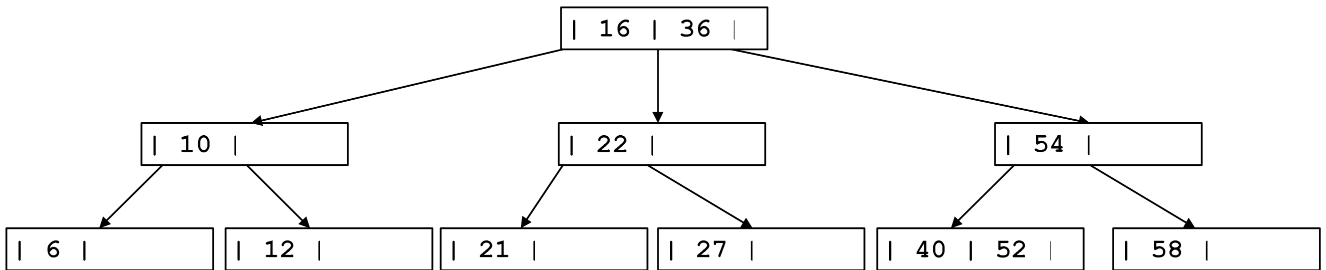
Delete 50



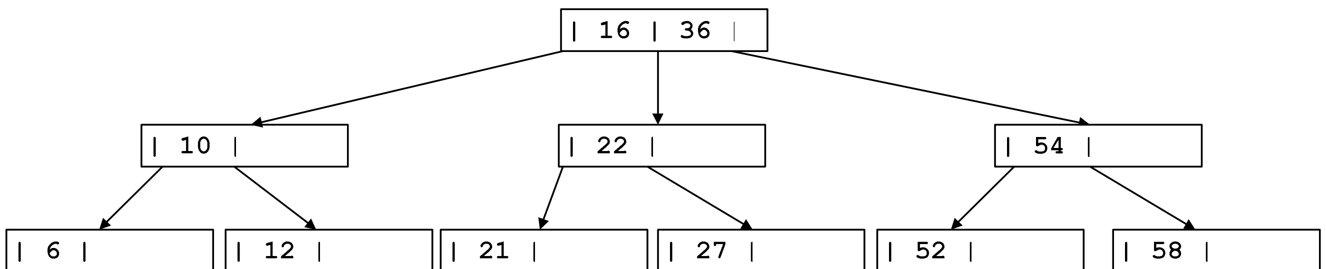
Delete 42



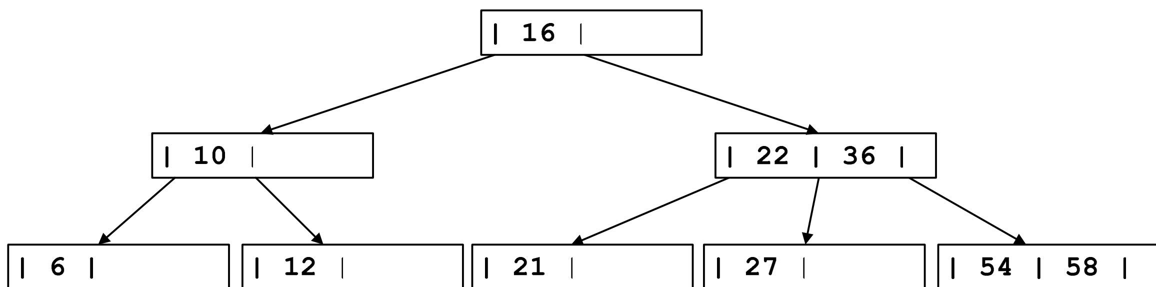
Delete 38



Delete 40



Delete 52



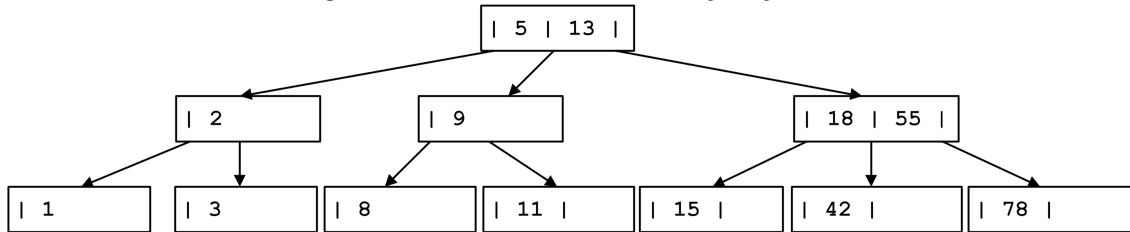
Popunjenost stabla: 0.625 (62.5%)	Broj pristupa: - Neuspešno traženje: 3 - Uspešno traženje: (1*1 + 3*2 + 6*3)/10=2.5

Zadatak 10.1.2 – Zadatak za samostalnu vežbu

U inicijalno prazno B-stablo reda 3 umeću se redom ključevi 8, 18, 13, 3, 5, 42, 15, 11, 9, 55, 78, 1, 2 a zatim se redom brišu ključevi 9, 5, 1, 78. Nacrtati izgled stabla nakon svake od navedenih izmena. Koliki je srednji broj pristupa prilikom uspešnog i neuspešnog traženja, kao i popunjenost B stabla, posle svih umetanja ključeva i u završnom stanju?

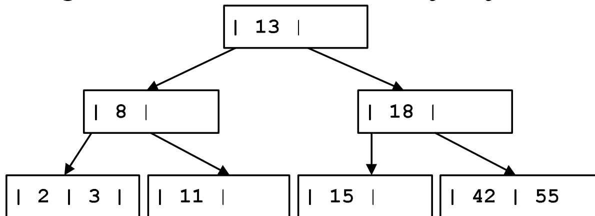
Rešenje:

Izgled stabla nakon svih umetanja ključeva



Popunjenost stabla: 13/22=0.59 (59%)	Broj pristupa: - Neuspešno traženje: 3 - Uspešno traženje: (2*1 + 4*2 + 7*3)/13=2.38

Izgled stabla nakon svih brisanja ključeva



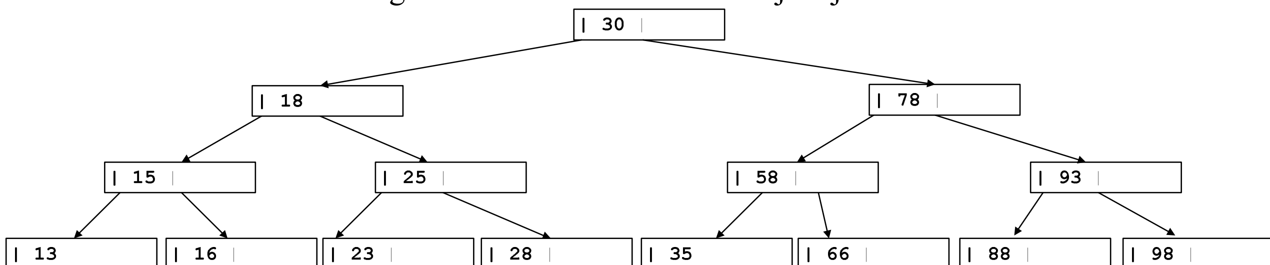
Popunjenost stabla: 9/14=0.64 (64%)
Broj pristupa:
- Neuspešno traženje: 3
- Uspešno traženje: (1*1 + 2*2 + 6*3)/9= 23/9 =2.55

Zadatak 10.1.3 – Zadatak za samostalnu vežbu

U inicijalno prazno B-stablo reda 3 umeću se redom ključevi 13, 23, 18, 16, 35, 30, 28, 58, 78, 88, 66, 25, 98, 93, 15, a zatim se redom brišu ključevi 18, 30, 88, 16. Nacrtati izgled stabla nakon svake od navedenih izmena. Koliki je srednji broj pristupa prilikom uspešnog i neuspešnog traženja, kao i popunjenost B stabla, posle svih umetanja ključeva i u završnom stanju?

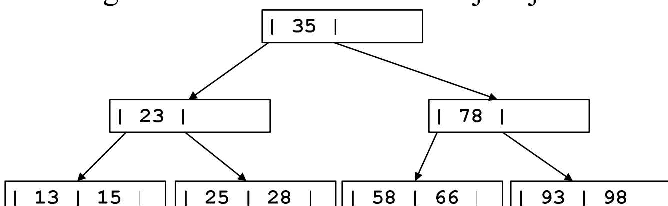
Rešenje:

Izgled stabla nakon svih umetanja ključeva



Popunjenost stabla: 0.5 (50%)	Broj pristupa: - Neuspešno traženje: 4 - Uspešno traženje: (1*1 + 2*2 + 4*3 + 8*4)/15=3.267

Izgled stabla nakon svih brisanja ključeva



Popunjenost stabla: 0.79 (79%)
Broj pristupa:
- Neuspešno traženje: 3
- Uspešno traženje: (1*1 + 2*2 + 8*3)/11=2.64

10.2 B* Stabla

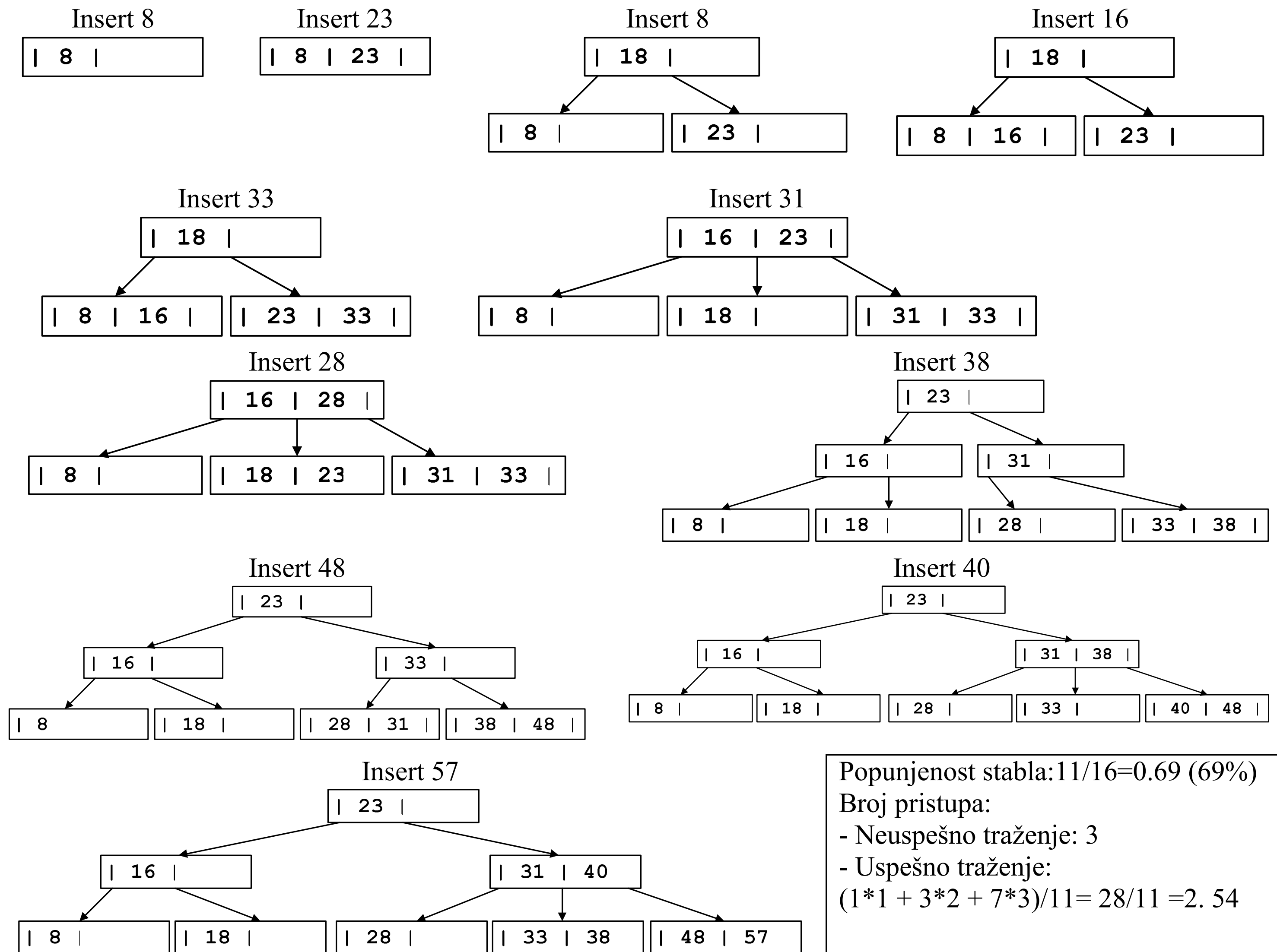
Zadatak 10.2.1

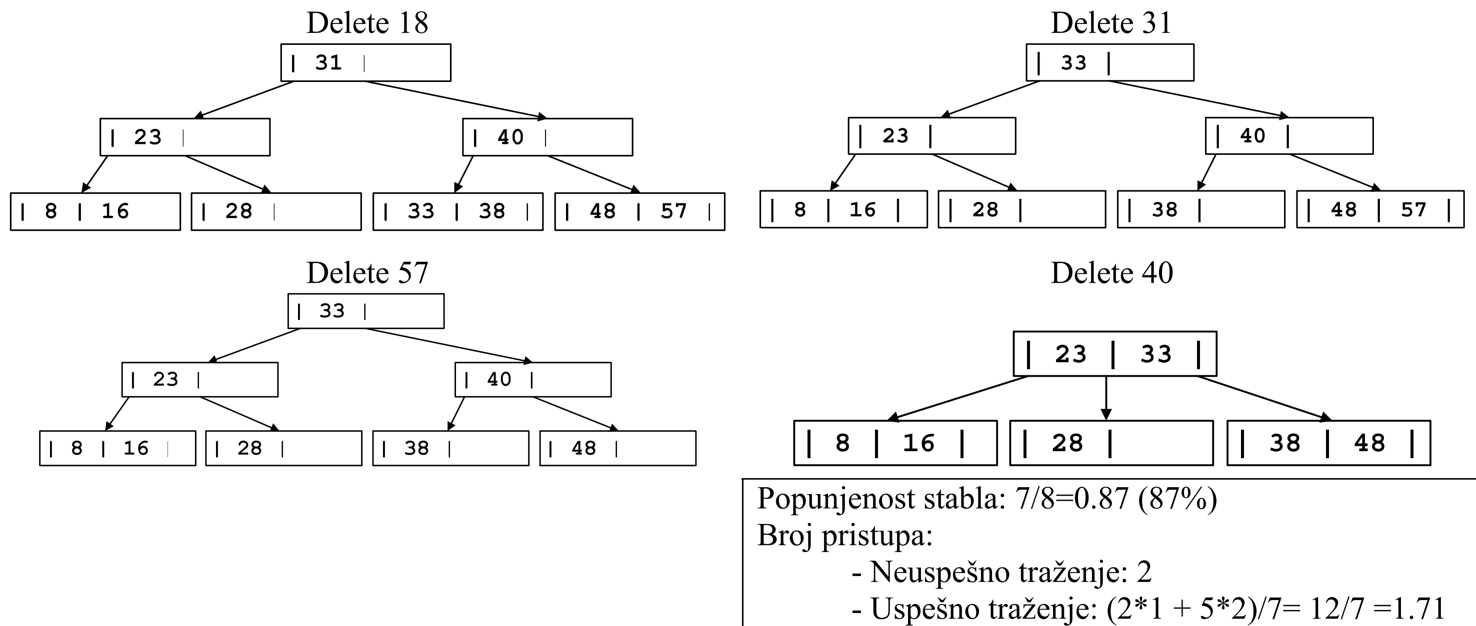
U prazno B*-stablo reda 3 umeću se redom ključevi 8, 23, 18, 16, 33, 31, 28, 38, 48, 40, 57 a zatim se redom brišu ključevi 18, 31, 57, 40. Nacrtati izgled stabla nakon svake od navedenih izmena. Koliki je srednji broj pristupa prilikom uspešnog i neuspešnog traženja, kao i popunjenost stabla, posle svih umetanja ključeva i u završnom stanju?

Rešenje:

m=3

- svaki čvor osim listova i korena ima najmanje $\left\lceil \frac{2m-1}{3} \right\rceil = \left\lceil \frac{6-1}{3} \right\rceil = 2$ podstabla
- koren ima najmanje 2, a najviše $2\left\lfloor \frac{2m-2}{3} \right\rfloor + 1 = 2\left\lfloor \frac{6-2}{3} \right\rfloor + 1 = 3$ podstabla
- nakon podele nekog čvora, ključevi se raspoređuju u odnosu 1-1-2





10.3 B+ Stabla

Zadatak 10.3.1

U prazno B+-stablo reda 3 umeću se redom ključevi 40, 20, 30, 25, 35, 22, 45, 37, 10, 18, 24 a zatim se redom brišu ključevi 35, 22, 30, 18. Nacrtati izgled stabla nakon svake od navedenih izmena. Koliki je srednji broj pristupa prilikom uspešnog i neuspešnog traženja, kao i popunjenost stabla, posle svih umetanja ključeva i u završnom stanju?

Rešenje:

Za razliku od B i B* stabala, unutrašnji čvorovi i listovi kod B+ stabala se razlikuju po sadržaju. Informacioni sadržaj postoji samo u okviru listova. Unutrašnji čvorovi stabla sadrže replike ključeva iz listova, a jedan ključ može ima više replika. Replicira se ključ najveće vrednosti u svakom podstablu i prenosi se u roditeljski čvor.

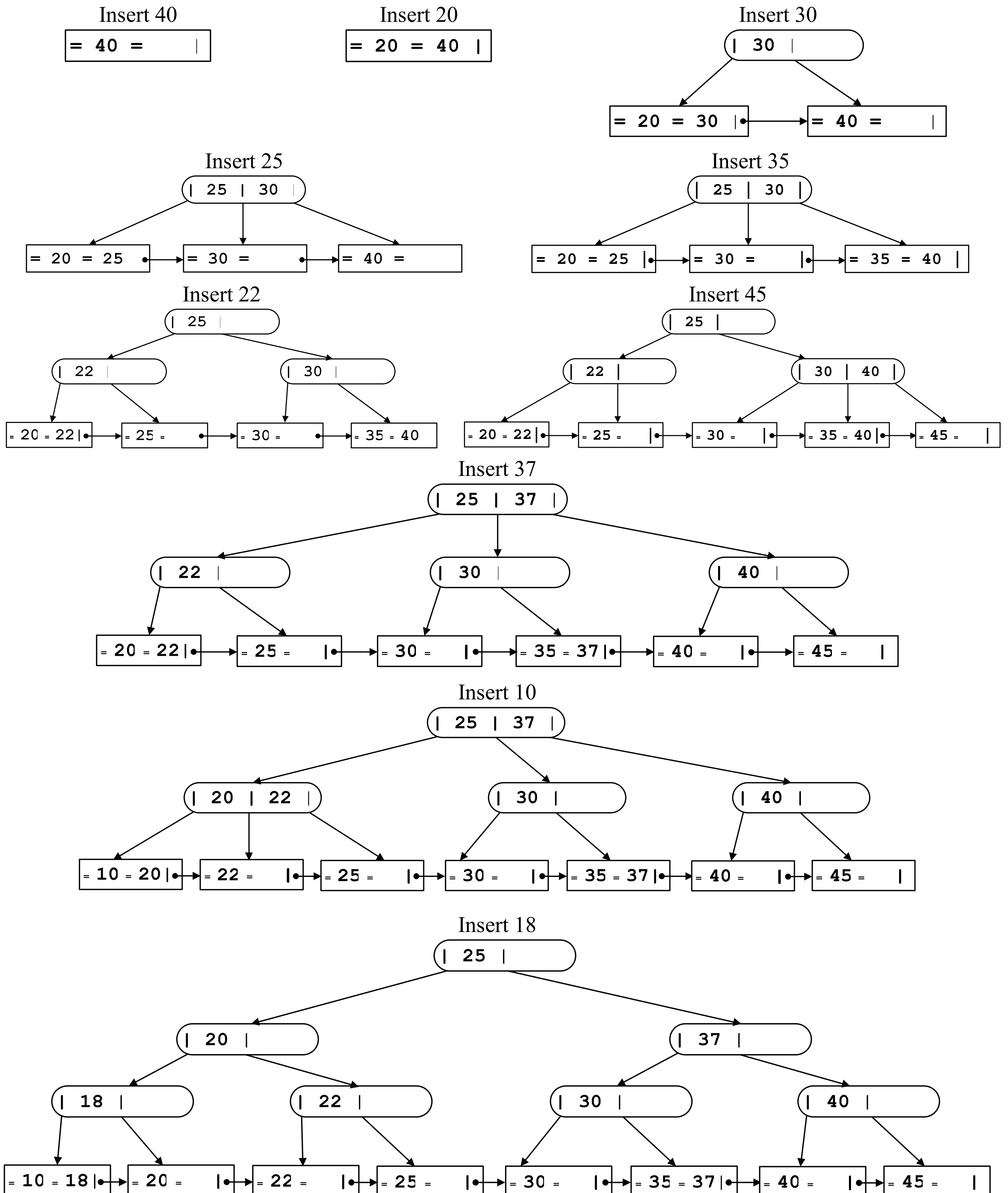
Slično B stablu, svaki unutrašnji čvor stabla ima najviše m podstabala. Koren ima najmanje 2 podstabla, a unutrašnji čvorovi $\lceil \frac{m}{2} \rceil$, dok listovi imaju najmanje $\lfloor \frac{m}{2} \rfloor$ ključeva.

Prilikom umetanja ili brisanja, unutrašnji čvorovi stabla se ponašaju kao kod B–stabla, a razlika postoji kod ažuriranja listova. Novi ključ se uvek umeće u list. Ako dođe do prepunjenja lista X, list X se prelama i novi list Y se dodaje u stablo. $\lceil \frac{m}{2} \rceil$ ključeva koji su se nalazili u listu X prilikom umetanja novog ključa (uključujući i njega), ostaje u čvoru X, a ostatak ide u čvor Y. Najveći ključ iz čvora X se propagira u roditeljski čvor. Ključ se uvek briše iz lista. Pri tome se vodi računa o tome da li je obrisani ključ bio repliciran, jer tada unutrašnji čvorovi stabla moraju da se ažuriraju i obrisani ključ mora da se zameni najvećim preostalim ključem iz datog podstabla.

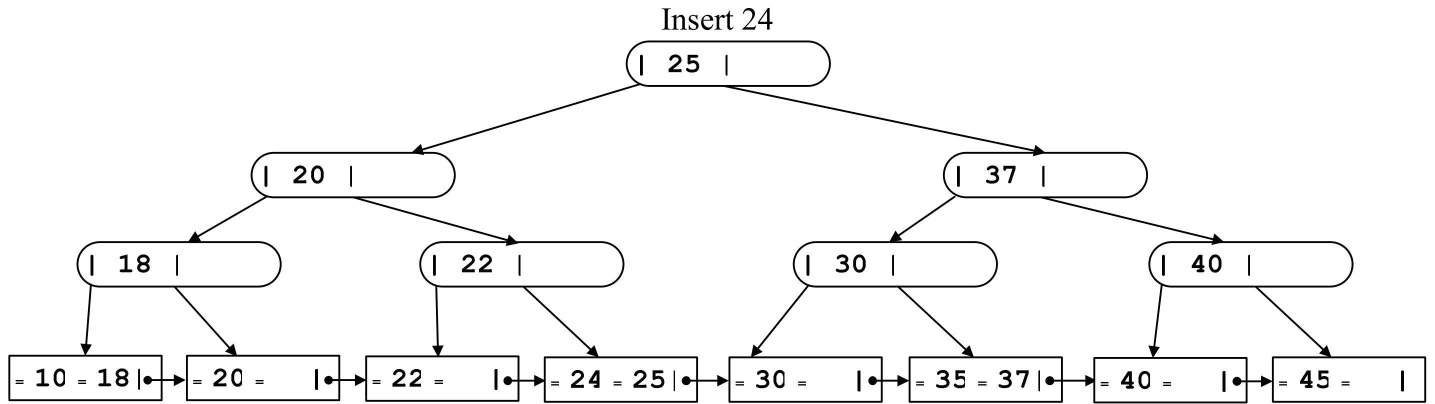
m=3

- svaki unutrašnji čvor i koren imaju najmanje 2, a najviše 3 podstabla
- listovi imaju najmanje 1 ključ

Napomena: simbolom = je prikazan separator između ključeva u listovima stabla. Simbolom | su prikazani pokazivači na podstabla odnosno na sledeći čvor u ulančanoj listi. Listovi stabla su predstavljeni pravougaonicima, unutrašnji čvorovi su predstavljeni zaobljenim pravougaonicima.

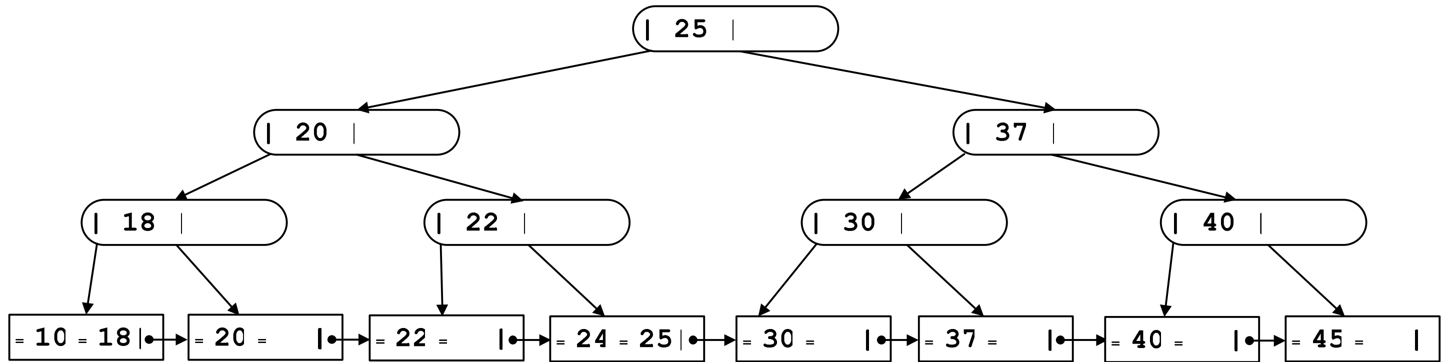


Insert 24

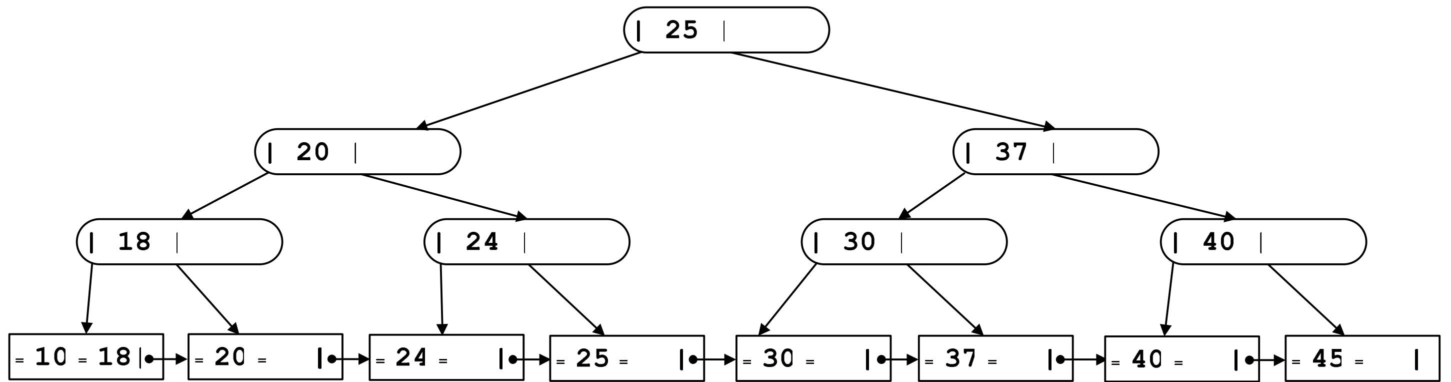


Popunjenost stabla: 18/30=0.6 (60%)	Broj pristupa: - Neuspešno traženje: 4 - Uspešno traženje: 4

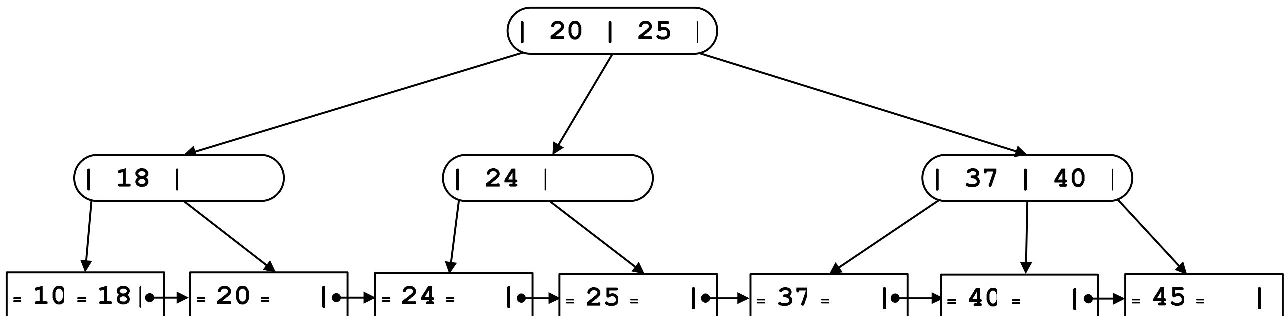
Delete 35

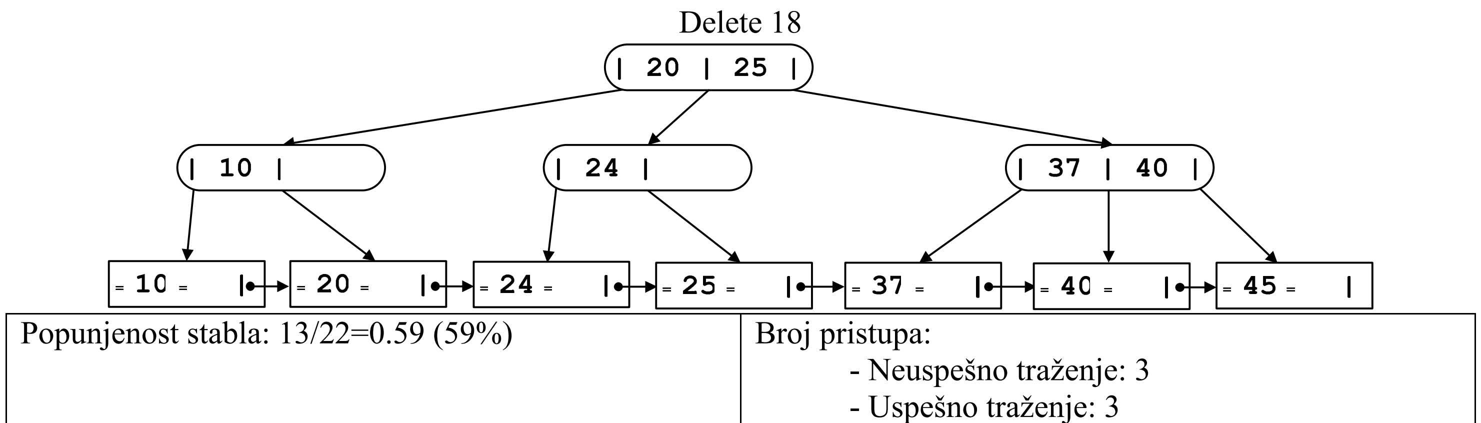


Delete 22



Delete 30



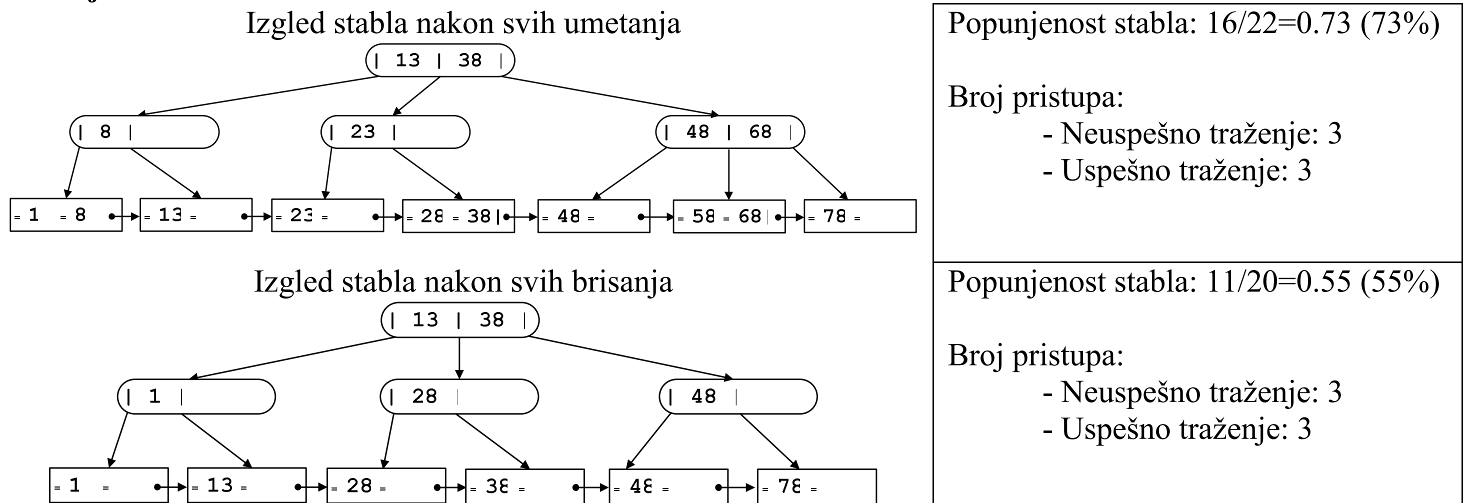


Popunjenost stabla: 13/22=0.59 (59%)	Broj pristupa: - Neuspešno traženje: 3 - Uspešno traženje: 3

Zadatak 10.3.2 – Zadatak za samostalnu vežbu

U prazno B+-stablo reda 3 umeću se redom ključevi 78, 13, 48, 38, 58, 23, 68, 1, 8, 28 a zatim se redom brišu ključevi 58, 23, 68, 8. Nacrtati izgled stabla nakon svake od navedenih izmena. Koliki je srednji broj pristupa prilikom uspešnog i neuspešnog traženja, kao i popunjenost stabla, posle svih umetanja ključeva i u završnom stanju?

Rešenje:



Popunjenost stabla: 16/22=0.73 (73%) Broj pristupa: - Neuspešno traženje: 3 - Uspešno traženje: 3
Popunjenost stabla: 11/20=0.55 (55%) Broj pristupa: - Neuspešno traženje: 3 - Uspešno traženje: 3

10.4 Trie i Digitalna Stabla

Zadatak 10.4.1

Objasniti šta je *trie* a šta *digitalno* stablo. Koje su njihove prednosti i mane u odnosu na druga stabla opšteg pretraživanja? Porediti performanse i memorijske zahteve. Prikazati osnovne operacije (umetanje i brisanje ključa) nad ovim stablima.

Rešenje:

Trie: stablo za skladištenje prefiksnih ključeva (znakovnih nizova) kod koga ključevi istog prefiksa dele jedan ili više čvorova. Ključevi su smešteni u zasebne listove. Naziv potiče od re**TRIE**val, čita se isto kao i "*tree*". Neki autori namerno čitaju kao "*try*" da bi pravili razliku.

<u>Digitalno stablo</u>: trie kod koga su delovi ključa smešteni u unutrašnjim čvorovima. Kod njih nema potrebe za smeštanjem ključeva u zasebne listove. Potreban specijalan simbol koji označava kraj ključa (EOK)

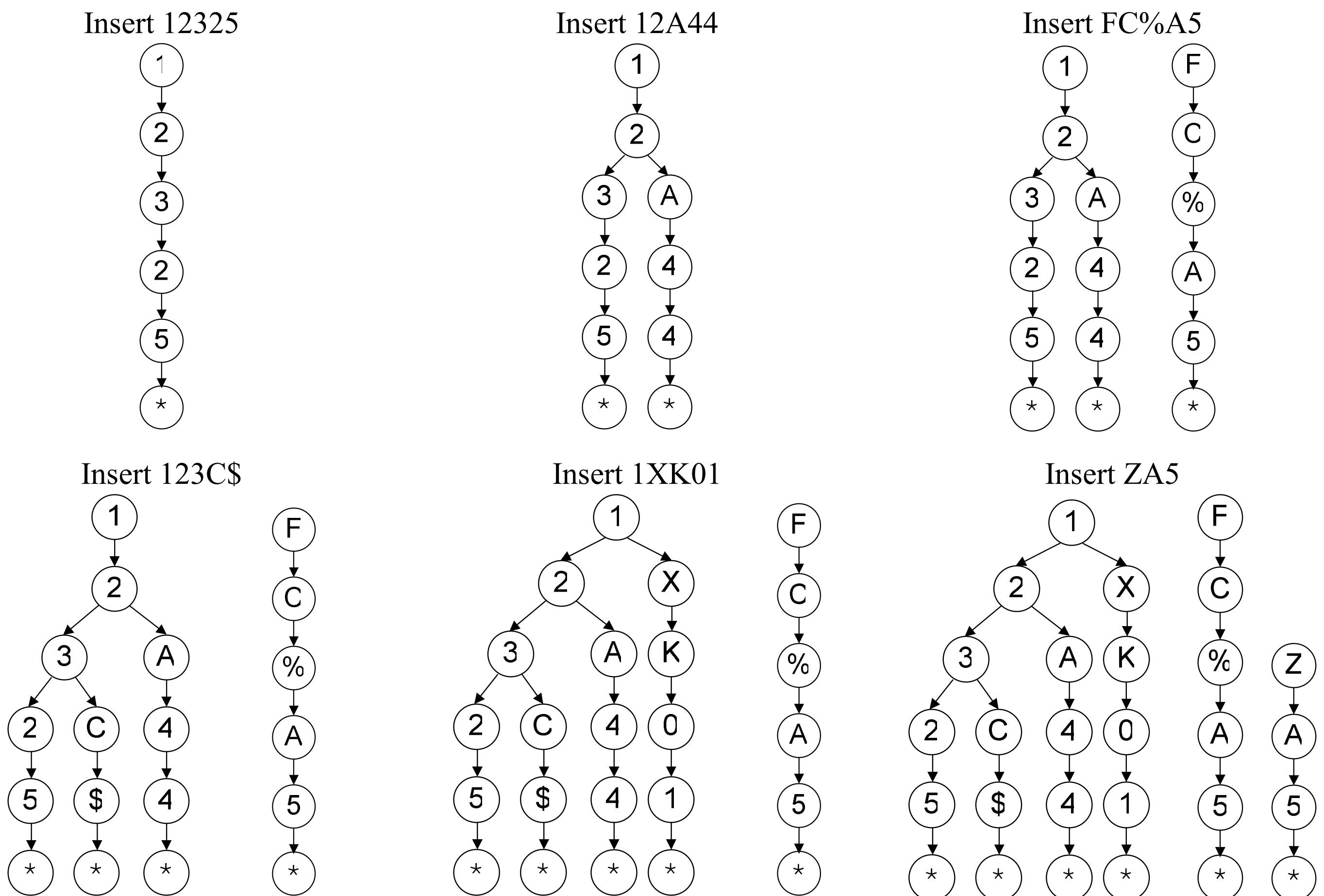
Prednosti:

- vreme pretrage proporcionalno dužini ključa O(m) (najgori slučaj) uz manju vremensku konstantu (poredi se samo deo ključa), dok kod ostalih stabala zavisi od broja ključeva
- bolje iskorišćenje prostora za veći broj kraćih ključeva (naročito kod digitalnog stabla)
- nije potrebno balansiranje radi efikasne pretrage

Mane:

- loše iskorišćenje prostora za dugačke ključeve bez zajedničkog prefiksa (postoje implementacije koje prevazilaze ovaj nedostatak)
- složeniji algoritmi nego kod običnog m-arnog stabla
- postoje podaci koji se ne mogu efikasno predstaviti, poput realnh brojeva

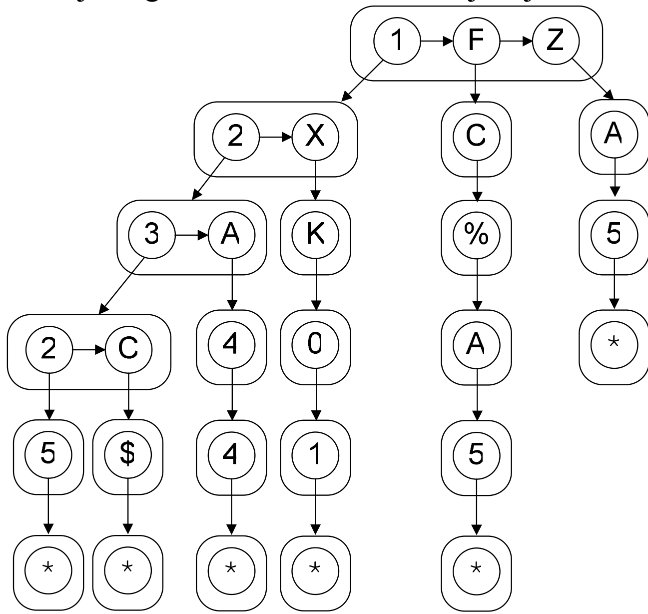
Digitalno stablo: efikasna primena kada su ključevi retki.
Primer: umetanje ključeva 12325, 12A44, FC%A5, 123C$, 1XK01, ZA5, ZX81 (simbol * označava EOK)



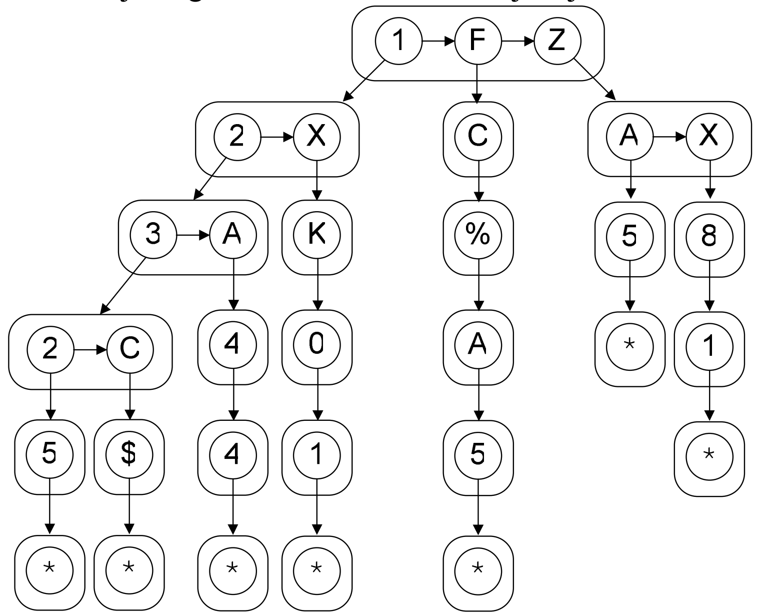
U datom primeru, ključevi su znakovni nizovi proizvoljne dužine. Svako podstablo sadrži ključeve koji imaju zajednički prefiks. Na primer, ključevi 12325 i 123C$ imaju zajednički prefiks 123, pa se zato oba ključa nalaze u istom podstablu.

Način na koji je u gornjem primeru prikazano digitalno stablo ne odgovara njegovoj implementaciji. Sledeća slika detaljnije prikazuje izgled stabla nakon umetanja ključa ZA5.

Detaljni izgled stabla nakon umetanja ključa ZA5

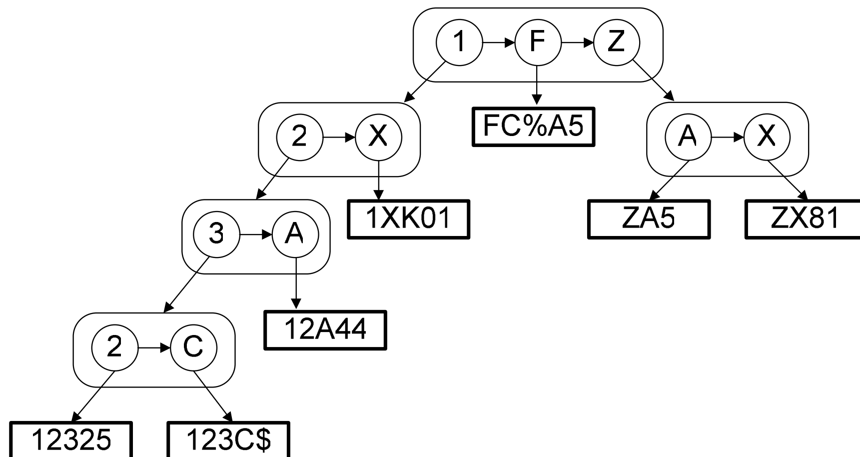


Detaljni izgled stabla nakon umetanja ključa ZX81

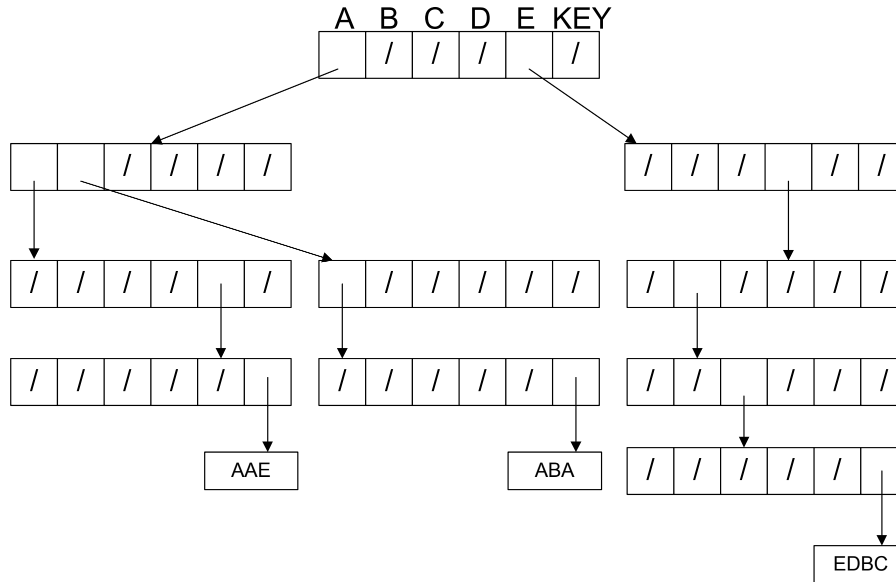


U okviru čvora digitalnog stabla se nalazi lista pokazivača na čvorove-potomke (podstabla). Drugim rečima, svaki čvor sem listova sadrži listu kod koje je informacioni deo jednog elementa deo ključa i pokazivač na podstablo koje odgovara tom ključu. Na primer, svi ključevi koji započinju simbolom "1" nalaze se u podstablu na koje ukazuje element liste "1" korena. Čvorovi stabla su prikazani zaobljenim pravougaonicima, a elementi lista krugom. Ovakav pristup je memorijski efikasnije nego da se statički alocira niz koji pokriva ceo alfabet, ali je sporiji zbog potrebe da se liste pretražuju.

U slučaju dugih, retkih ključeva, nešto drugačijom implementacijom stabla se može postići dodatna ušteda u memorijskom prostoru, uz dugotrajniju režiju prilikom umetanja ili brisanja ključeva. Ideja je sledeća: ako neko podstablo sadrži samo jedan ključ, onda se taj ključ ne razdvaja na pojedinačne simbole, već se taj ključ odmah smešta u list. Na sledećoj slici je prikazano stablo iz prethodnog primera uz primenu ove modifikacije:



Trie stablo: efikasna primena za veću gustinu ključeva.
U okviru čvora se nalazi **niz pokazivača** na potomke i **poseban pokazivač** na specijalan list koji sadrži ključ.
Potomaka ima koliko i simbola od kojih se grade ključevi.
Primer: skup simbola je A..E. Prikazati izgled stabla nakon umetanja ključeva AAE,ABA, EDBC.
Neiskorišćene pokazivače predstaviti simbolom "/".

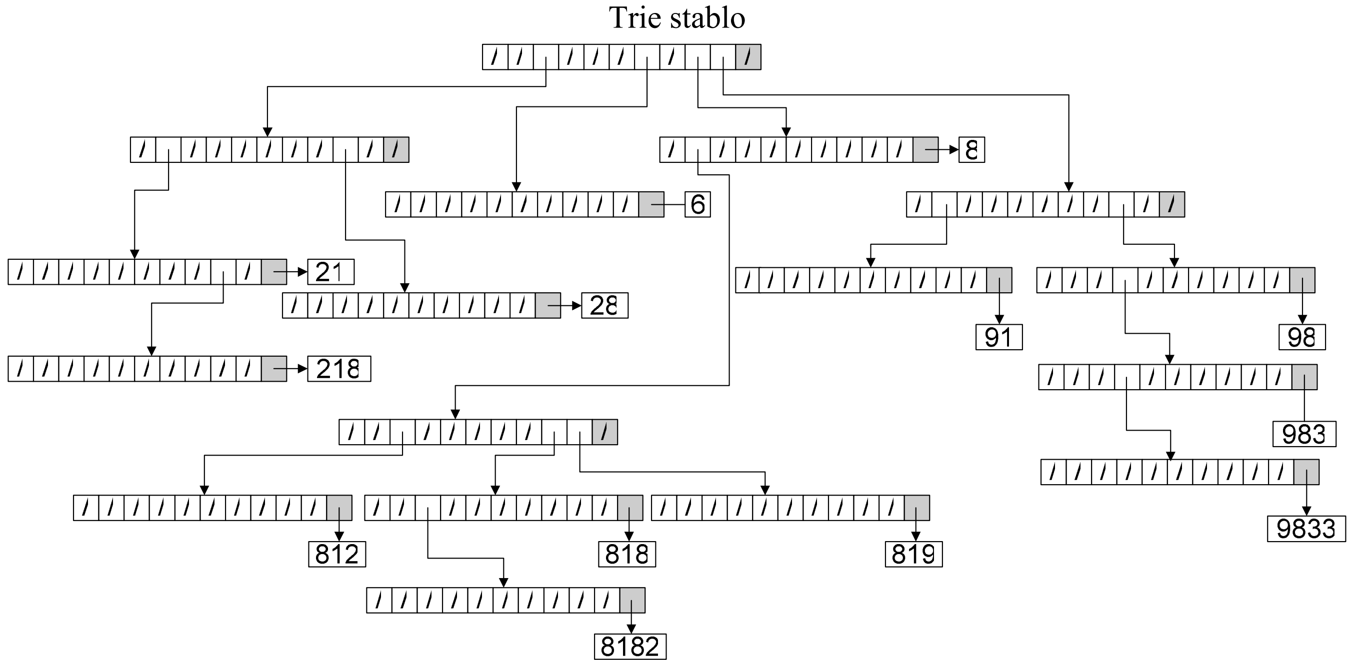


Za razliku od digitalnog stabla, svaki čvor trie stabla fiksno alocira niz pokazivača one dužine koja odgovara broju simbola u alfabetu od kojeg se sastavljaju ključevi. Dok je kod digitalnog stabla za pristup podstablu potreban prolazak kroz listu, ovde se podstablu direktno pristupa koristeći odgovarajući simbol ključa kao indeks. Postignut je brz pristup ključu uz veliku cenu u neiskorišćenom memorijskom prostoru naročito u slučaju retkih ključeva čije se dužine značajno razlikuju.

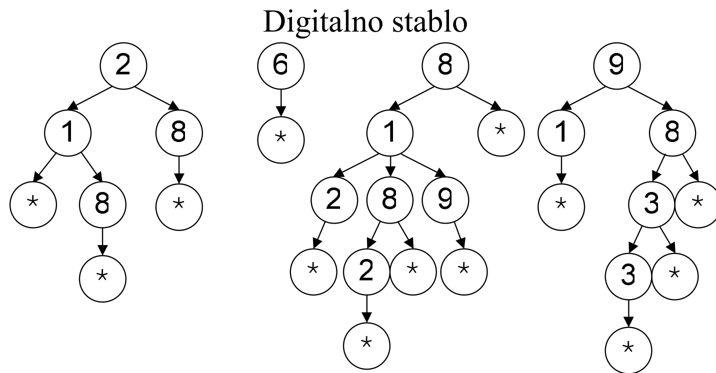
Zadatak 10.4.2

Nacrtati izgled trie stabla za sledeći skup ključeva: 28, 812, 8, 819, 218, 818, 21, 9833, 8182, 91, 6, 98, 983. Izračunati iskorišćenje memorijskog prostora koje zauzima stablo Predložiti rešenje koje bi poboljšalo iskorišćenje. Za isti skup ključeva, nacrtati izgled digitalnog stabla.

Rešenje:



Ukupan broj čvorova: 17
Ukupan broj pokazivača: 187
Broj upotrebljenih pokazivača: 29
Iskorišćenje: 29/187=0.155 (15.5%)



11. HEŠIRANJE

Hash (eng.)**:** potiče od francuskog glagola *hacher* (seckati):
- (neformalno) nered
- (kulinarstvo) mešavina fino seckanih namirnica

Heširanje: tehnika kojom se vrši preslikavanje skupa ključeva na tabelu značajno manjih dimenzija

Cilj heširanja: ostvariti efikasan pristup (pretraga, umetanje, brisanje) ključevima smeštenih u tabelu, uz malu cenu režije.

Preslikavanje se vrši primenom **heš funkcije** nad ključevima. Heš funkcija kao rezultat vraća redni broj ulaza (**matičnu adresu**) u tabeli gde treba smestiti ključ.
Problem: više ključeva ima istu matičnu adresu – **kolizija**. Takvi ključevi se nazivaju **sinonimima**
Rešenje:
- primeniti **savršenu heš funkciju** koja garantuje da nema kolizije
  (nije jednostavna za određivanje, svi ključevi moraju biti unapred poznati)
- primeniti neku od tehnika za razrešavanje kolizije
  1. otvoreno adresiranje
  2. ulančavanje

Zadatak 11.1

Podaci se smeštaju u heš tabelu sa 9 ulaza primenom metode otvorenog adresiranja sa dvostrukim heširanjem. Primarna heš funkcija je $h_p(K)=K \bmod 9$, a sekundarna heš funkcija je $h_s(K)=4 + (K \bmod 2)$.
a) Prikazati popunjavanje tabele ako redom dolaze ključevi 38, 28, 33, 20, 23, 39.
b) Izračunati srednji broj pristupa prilikom uspešnog i neuspešnog traženja u tabeli, kao i verovatnoću popunjavanja preostalih praznih lokacija pri prvom sledećem umetanju, ako su svi ključevi jednako verovatni.

Rešenje:
Metoda otvorenog adresiranja: dozvoliti da se ključ smesti u lokaciju koja nije njegova matična adresa. Kada se konstatuje kolizija na matičnoj adresi, generiše se niz adresa (**ispitni niz**), od kojih se prva prazna koristi za smeštanje ključa.

Metode generisanja niza adresa:
- linearno pretraživanje
- kvadratno pretraživanje
- dvostruko heširanje

Uopšteno: $h_{i+1}(K) = (h_i(K) + g(K)) \bmod n$

0	
1	28
2	38
3	39
4	
5	20
6	33
7	23
8	

Akcije prilikom umetanja ključeva (hronološki redosled)

(38 mod 9) = 2 – ulaz je slobodan
(28 mod 9) = 1 – ulaz je slobodan
(33 mod 9) = 6 – ulaz je slobodan
(20 mod 9) = 2 – ulaz nije slobodan
    (2 + 4 + 20 mod 2) mod 9 = 6 – ulaz nije slobodan
    (6 + 4 + 20 mod 2) mod 9 = 1 – ulaz nije slobodan
    (1 + 4 + 20 mod 2) mod 9 = 5 – ulaz je slobodan
(23 mod 9) = 5 – ulaz nije slobodan
    (5 + 4 + 23 mod 2) mod 9 = 1 – ulaz nije slobodan
    (1 + 4 + 23 mod 2) mod 9 = 6 – ulaz nije slobodan
    (6 + 4 + 23 mod 2) mod 9 = 2 – ulaz nije slobodan
    (2 + 4 + 23 mod 2) mod 9 = 7 – ulaz je slobodan
(39 mod 9) = 3 – ulaz je slobodan

Srednji broj pristupa:

- uspešno: P_u = 13/6 ~ 2.16
- neuspešno:P_{nu}= 210/84 = 2.5 ;   približno P_{nu}: α=2/3, 1/(1- α) = 3

Pretraga može biti uspešna samo ako se traži ključ koji se nalazi u tabeli. Tom prilikom, ispitni niz koji se generiše za dati ključ je isti kao i prilikom umetanja tog ključa u tabelu. U ovom slučaju, srednji broj pristupa je, drugim rečima, srednja dužina ispitnog niza. Iz tog razloga, dovoljno je prebrojati pokušaje prilikom umetanja ključeva (ima ih 13) i podeliti sa brojem ključeva (ima ih 6) da bi se dobio traženi broj pristupa.

Za izračunavanje srednjeg broja pristupa pri neuspešnom pretraživanju, biće primenjena sledeća analiza, ali na osnovu sledećih pretpostavki:

- primenjena heš funkcija ne ponavlja indekse ulaza tabele sve dok ne generiše indekse za sve ulaze (što je tačno za kombinaciju primarne i sekundarne heš funkcije iz ovog primera)
- ključevi na koje se vrši (neuspešna) pretraga su jednako verovatni
- heš funkcija uniformno raspoređuje ključeve

Pretraga u heš tabeli započinje sa onim ulazom koji vrati heš funkcija za dati ključ. Ukoliko je ulaz u heš tabeli prilikom pretrage prazan, pretraživanje se proglašava neuspešnim i bio je potreban svega jedan pristup heš tabeli. Ukoliko je popunjen nekim drugim ključem, koristeći sekundarnu heš funkciju traži se sledeći ulaz u ispitnom nizu. Ukoliko je on prazan, pretraživanje se proglašava neuspešnim i bilo je potrebno dva pristupa heš tabeli da bi se to zaključilo, a ukoliko se u njemu nalazi neki drugi ključ, ponovo se poziva sekundarna heš funkcija i tako redom, sve dok se ne dođe do praznog ulaza ili dok se ne obiđu svi ulazi tabele (heš tabela je puna) što je indikacija da je pretraživanje neuspešno. Ukoliko se bilo gde u ispitnom nizu pojavi dati ključ, pretraživanje je uspešno i taj slučaj nije od interesa – posmatra se isključivo neuspešna pretraga. Srednji broj pristupa pri neuspešnoj pretrazi se određuje na sledeći način:
$P_{nu} = 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + 4 \cdot p_4 + 5 \cdot p_5 + 6 \cdot p_6 + 7 \cdot p_7$, gde je p_i verovatnoća da se u i-tom pristupu heš tabeli zaključilo da je pretraživanje neuspešno, tj. da je u prvih i-1 pristupa pronađen neki drugi ključ, a u i-tom pristupu je pronađen prazan ulaz. U konkretnom primeru, za donošenje zaključka da je pretraživanje neuspešno, potrebno je najviše 7 pristupa (u najgorem slučaju pristupa se 6 puta svim ostalim popunjenim ulazima). U opštem slučaju, potrebno je najviše m+1 pristup heš tabeli, gde je m broj popunjenih ulaza, za slučaj da heš tabela nije 100% popunjena, ili m pristupa za slučaj kada je heš tabela 100% popunjena.
Za određivanje P_{nu}, potrebno je odrediti sve verovatnoće koje figurišu u izrazu.

$p_1 = \frac{3}{9} = \frac{1}{3}$ - verovatnoća da je heš funkcija vratila ulaz koji je slobodan, i pretraživanje već posle prvog pristupa se proglašava neuspešnim (od devet mogućih ulaza, tri su prazna).

$p_2 = \frac{6}{9} \cdot \frac{3}{8} = \frac{1}{4}$ - verovatnoća da je heš funkcija vratila ulaz koji je zauzet drugim ključem (6/9), ali je u narednom pristupu pronađen prazan ulaz, i pretraživanje se posle drugog pristupa proglašava neuspešnim (od preostalih osam mogućih ulaza za drugi pokušaj, tri su prazna). Ovde se koristi uvedena pretpostavka da kombinacija primarne i sekundarne heš funkcija neće vratiti ranije vraćeni broj ulaza dok se ne obiđu svi ostali (zbog toga je 3/8, a ne 3/9).

$p_3 = \frac{6}{9} \cdot \frac{5}{8} \cdot \frac{3}{7} = \frac{5}{28}$ - verovatnoća da je heš funkcija vratila ulaz koji je zauzet drugim ključem (6/9), ali i da je u narednom pristupu pronađen ulaz zauzet nekim drugim ključem (od preostalih osam mogućih ulaza, pet su zauzeti), a u trećem pristupu je pronađen prazan ulaz (od preostalih sedam mogućih ulaza za treći pokušaj, tri odgovaraju), i pretraživanje se posle trećeg pristupa proglašava neuspešnim.

$$p_4 = \frac{6}{9} \cdot \frac{5}{8} \cdot \frac{4}{7} \cdot \frac{3}{6} = \frac{5}{42}$$

$$p_5 = \frac{6}{9} \cdot \frac{5}{8} \cdot \frac{4}{7} \cdot \frac{3}{6} \cdot \frac{3}{5} = \frac{1}{14}$$

$$p_6 = \frac{6}{9} \cdot \frac{5}{8} \cdot \frac{4}{7} \cdot \frac{3}{6} \cdot \frac{2}{5} \cdot \frac{3}{4} = \frac{1}{28}$$

$$p_7 = \frac{6}{9} \cdot \frac{5}{8} \cdot \frac{4}{7} \cdot \frac{3}{6} \cdot \frac{2}{5} \cdot \frac{1}{4} \cdot \frac{3}{3} = \frac{1}{84}$$

$$P_{nu} = 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + 4 \cdot p_4 + 5 \cdot p_5 + 6 \cdot p_6 + 7 \cdot p_7 =$$

$$= \frac{1}{3} + \frac{1}{2} + \frac{15}{28} + \frac{10}{21} + \frac{5}{14} + \frac{3}{14} + \frac{1}{12} = \frac{210}{84} = 2.5$$

Za potrebe procene gornje granice očekivanog broja pristupa prilikom neuspešne pretrage, može se primeniti sledeća, manje precizna analiza. Neka je α popunjenost tabele, data kao količnik broja ključeva smeštenih u tabelu m i veličine tabele n. Tada $1-\alpha$ predstavlja nepopunjeni deo tabele, odnosno deo koji čine prazni ulazi. Ako je $\alpha = 0.95$, $1-\alpha = 0.05 = \frac{1}{20}$, tada je u proseku potrebno najviše iz 20 pristupa da bi se konstatovalo da je pretraživanje bilo neuspešno. Može se i formalno dokazati da je prosečan broj potrebnih pristupa obrnuto srazmeran veličini praznog dela heš tabele. Stoga se za ovu procenu koristi izraz $P_{nu} \le \frac{1}{1-\alpha}$. U našem slučaju, $\frac{1}{1-\alpha} = 3$, što je veće od izračunatih 2.5, ali daje bržu procenu.

Da bi se odredila verovatnoća popunjavanja praznih lokacija u tabeli, potrebno je utvrditi koji sve ispitni nizovi postoje i koja je verovatnoća pojavljivanja određene prazne lokacije u njemu. S obzirom na to da sekundarna heš funkcija sadrži izračunavanje K mod 2, postojaće dva ciklička ispitna niza, jedan za K parno, a drugi za K neparno. Najjednostavnije je pretpostaviti da je prva adresa u ispitnom nizu 0 i od nje produžiti ispitni niz. Određivanje ispitnog niza se završava kada se ponovo dođe do adrese 0.

<table>
<tr><th>K parno</th><th>K neparno</th><td rowspan="13">U ovom zadatku, prazne su lokacije 0, 4 i 8. Posmatranjem ova dva ispitna niza, može se uočiti sledeće:
za parno K, najveća je verovatnoća da se ispitni niz završi na adresi 0, tačnije – ako se u prvom koraku ne pristupi lokaciji 4 ili 8, sigurno će se završiti na adresi 0
za neparno K važi to isto, ali ne za adresu 0 već za adresu 8 – ako se u prvom koraku ne pristupi adresi 0 ili 4, sigurno će se ispitni niz završiti na adresi 8
Iz toga se može zaključiti da je verovatnoća popunjavanja adresa 0 i 8 jednaka, a da je verovatnoća popunjavanja adrese 4 jednaka verovatnoći da se odmah pristupi adresi 4 i iznosi 1/9. Iz toga se jednostavno određuje da verovatnoća popunjavanja adresa 0 i 8 iznosi 4/9.</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>4</td><td>5</td></tr>
<tr><td>8</td><td>1</td></tr>
<tr><td>3</td><td>6</td></tr>
<tr><td>7</td><td>2</td></tr>
<tr><td>2</td><td>7</td></tr>
<tr><td>6</td><td>3</td></tr>
<tr><td>1</td><td>8</td></tr>
<tr><td>5</td><td>4</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>...</td><td>...</td></tr>
</table>

Zadatak 11.2

Podaci se smeštaju u heš tabelu sa 7 ulaza primenom metode otvorenog adresiranja sa dvostrukim heširanjem. Primarna heš funkcija je $h_p(K)=K \bmod 7$, a sekundarna heš funkcija je $h_s(K)=2 + (K \bmod 3)$.

a) Prikazati popunjavanje tabele ako redom dolaze ključevi 45, 35, 17, 25, 18.

b) Izračunati srednji broj pristupa prilikom uspešnog i neuspešnog traženja u tabeli, kao i verovatnoću popunjavanja preostalih praznih lokacija pri prvom sledećem umetanju, ako su svi ključevi jednako verovatni.

Rešenje:

0	35
1	18
2	
3	45
4	17
5	
6	25

Srednji broj pristupa:

- uspešno: $P_u = 12/5 = 2.4$
- neuspešno: $P_{nu} = 2.67$

Verovatnoća popunjavanja:

- $2 : \frac{1}{7} + \frac{5}{7} \cdot \frac{1}{3} (\frac{1}{5} + 1 + 0) = \frac{3}{7}$
- $5 : \frac{1}{7} + \frac{5}{7} \cdot \frac{1}{3} (\frac{4}{5} + 0 + 1) = \frac{4}{7}$

Ispitni nizovi

K mod 3=0	K mod 3=1	K mod 3=2
0	0	0
2	3	4
4	6	1
6	2	5
1	5	2
3	1	6
5	4	3
0	0	0

Objašnjenje za određivanje verovatnoće popunjavanja lokacije 2 prilikom narednog umetanja ključa: lokacija 2 može biti popunjena ako je matična adresa novog ključa 2 ili ako njegova matična adresa nije ni 2 ni 5, ali se u ispitnom nizu adresa 2 pojavljuje pre adrese 5.

Verovatnoća da će adresa 2 biti matična adresa ključa: 1/7

Verovatnoća da matična adresa ključa nije ni 2 ni 5 : 5/7

Verovatnoća da je K mod 3=0 : 1/3 ; verovatnoća da je K mod 3=1 : 1/3 ; verovatnoća da je K mod 3=2 : 1/3

Verovatnoća da će se u ispitnom nizu pojaviti adresa 2 pre adrese 5:

- za K mod 3 = 0 : 1/5 (jedina adresa u tom ispitnom nizu je adresa 0)
- za K mod 3 = 1 : 1 (ispitni niz prolazi kroz sve adrese sem 2 i 5 pre nego što se pojavi adresa 2)
- za K mod 3 = 2 : 0 (ispitni niz najpre prolazi kroz adresu 5, pa do adrese 2 ne dolazi)

Na sličan način se određuje verovatnoća popunjavanja adrese 5 prilikom narednog umetanja ključa.

Zadatak 11.3

Ključevi se smeštaju u heš tabelu sa 10 ulaza primenom metode objedinjenog ulančavanja. Heš funkcija je $h_1(K) = K \bmod 10$. Prikazati popunjavanje tabele ako redom dolaze ključevi 42, 9, 25, 62, 88, 50, 19 i 78.

Rešenje:

Razrešenje kolizije se u ovom zadatku rešava ulančavanjem ključeva. Osnovna ideja razrešenja kolizije ulančavanjem je smeštanje ključeva za koje je detektovana kolizija u ulančanu listu tako da se ne generiše ispitni niz već vrši pretraga na ključ prolazom kroz listu. Primenjuju se dva pristupa ulančavanja:

- Odvojeno ulančavanje
  - ulaz tabele sadrži pokazivač na ulančanu listu umetnutih sinonima
  - koristi se posebno alocirana memorija (veliko rasipanje za malu popunjenost tabele)
- Objedinjeno ulančavanje
  - svaki ulaz tabele sadrži indeks sledećeg ključa u sekvenci (ne moraju svi ključevi u sekvenci biti sinonimi)
  - koristi memoriju alociranu za tabelu
  - efikasnije iskorišćenje memorije za malu popunjenost tabele
  - složenija implementacija

Obe metode ima smisla primenjivati samo za relativno kratke liste. U suprotnom dolazi do neželjenog gubitka performansi.

Insert 42: 42 mod 10 = 2

0	1	2	3	4	5	6	7	8	9
empty	empty	**42**	empty	empty	empty	empty	empty	empty	empty
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
									free

Insert 9: 9 mod 10 = 9

0	1	2	3	4	5	6	7	8	9
empty	empty	42	empty	empty	empty	empty	empty	empty	**9**
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
									free

Insert 25: 25 mod 10 = 5

0	1	2	3	4	5	6	7	8	9
empty	empty	42	empty	empty	**25**	empty	empty	empty	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
									free

Insert 62: 62 mod 10 = 2

0	1	2	3	4	5	6	7	8	9
empty	empty	42	empty	empty	25	empty	empty	**62**	9
-1	-1	**8**	-1	-1	-1	-1	-1	-1	-1
								free	

Matična adresa ključa 62 je 2. Ovaj ulaz u heš tabelu je već zauzet, pa se ključ nadovezuje na kraj ulančane liste počevši od adrese 2. Kako indeks sledećeg ulaza na adresi 2 iznosi -1, to znači da je ključ koji se nalazi na adresi 2 poslednji element liste iza kojeg treba nadovezati ključ 62. Zbog toga se pronalazi prva slobodna lokacija, počevši od lokacije koju indeksira **free**, idući prema nižim adresama u tabeli. Prva slobodna adresa je adresa 8, pa se zato na adresi 2 u polju sledećeg elementa liste smešta vrednost 8, a na adresi 8 se smešta novi ključ.

Insert 88: 88 mod 10 = 8

0	1	2	3	4	5	6	7	8	9
empty	empty	42	empty	empty	25	empty	**88**	62	9
-1	-1	8	-1	-1	-1	-1	-1	**7**	-1
							free		

Insert 50: 50 mod 10 = 0

0	1	2	3	4	5	6	7	8	9
50	empty	42	empty	empty	25	empty	88	62	9
-1	-1	8	-1	-1	-1	-1	-1	7	-1
							free		

Insert 19: 19 mod 10 = 9

0	1	2	3	4	5	6	7	8	9
50	empty	42	empty	empty	25	**19**	88	62	9
-1	-1	8	-1	-1	-1	-1	-1	7	**6**
						free			

Insert 78: 78 mod 10 = 8

Objašnjenje za umetanje ključa 78: matična adresa ovog ključa je 8 i zauzeta je. Ključ 78 treba umetnuti na kraj ulančane liste u okviru koje se nalazi adresa 8. Sledeći element ove liste je na adresi 7. Na toj adresi se nalazi ključ 88 (lokacija nije prazna) ali to je poslednji element liste (adresa sledećeg elementa je -1). Sada treba pronaći slobodnu lokaciju na koju će se umetnuti ključ 78. Lokacija 6 koja se indeksira pomoću free nije prazna, pa se free dekrementira. Lokacija 5 takođe nije prazna, pa se free još jednom dekrementira. Lokacija 4 jeste prazna i na nju će se smestiti ključ 78. Takođe treba ažurirati lokaciju 7, tako da se naznači da se na lokaciji 4 nalazi sledeći element liste. Sledeća slika prikazuje izgled tabele nakon ovih izmena.

0	1	2	3	4	5	6	7	8	9
50	empty	42	empty	**78**	25	19	88	62	9
-1	-1	8	-1	-1	-1	-1	**4**	7	6
				free					

Zadatak 11.4

Podaci se smeštaju u heš tabelu sa 10 ulaza. Primarna heš funkcija je $h_1(K) = K \bmod 10$. Prikazati punjenje tabele ako redom dolaze ključevi 8, 29, 52, 13, 89, 23, 50 i 44 u slučajevima primene sledećih metoda:

- linearno pretraživanje
- dvostruko heširanje sa sekundarnom heš funkcijom $h_2(K) = 2 + (K \bmod 2)$
- odvojeno ulančavanje
- objedinjeno ulančavanje.

Rešenje:

	Linearno pretraživanje	Dvostruko heširanje	Odvojeno ulančavanje	Objedinjeno ulančavanje		
0	89	50	50	50	-1	
1	50			empty	-1	
2	52	52	52	52	-1	
3	13	13	13→23	13	6	
4	23	44	44	44	-1	
5	44	89		empty	-1	
6		23		23	-1	free
7				89	-1	
8	8	8	8	8	-1	
9	29	29	29→89	29	7	

Zadatak 11.4 (IR2ASP - Jun 2007)

Napisati potprograme na programskom jeziku C ili C++ za rad sa heš tabelom, primenom metode objedinjenog ulančavanja. Ključevi su nenegativni celi brojevi. Obezbediti sledeće funkcionalnosti: stvaranje tabele sa zadatim brojem ulaza, umetanje ključa, dohvatanje ključa (vraća negativnu vrednost ako ključ ne postoji u tabeli). Napisati glavni program koji demonstrira korišćenje navedenih potprograma.

Rešenje

Ovde je dat obimniji program nego što se tražilo u postavci zadatka. Rešenje zadatka je prikazano zasivljeno.

```
// HashTable.h
#include <iostream>
using namespace std;

class HashTable
{
protected:
  int velicina;
  int slobodna;

  struct Element
  {
     int kljuc;
     int sledeci;
     Element():kljuc(-1), sledeci(-1) { }
  };

  Element *tabela;
```

```
  int pronadji_ulaz(int kljuc) const;
  int h(int kljuc) const
  { return kljuc % velicina; }

public:
     HashTable(int vel);
     virtual ~HashTable() { if( tabela ) delete []tabela; }

  int umetni(int kljuc);
  int dohvati(int kljuc);

  friend ostream & operator<<(ostream &os,
                              const HashTable &table);
};

// HashTable.cpp
#include "HashTable.h"
HashTable::HashTable(int vel) {
      velicina = vel;
      slobodna = vel-1;
      tabela = new Element[vel];
}

unsigned int HashTable::pronadji_ulaz(unsigned int kljuc) const {
      unsigned int i = h(kljuc);
      while( tabela[i].kljuc != kljuc && tabela[i].sledeci != -1)
            i = tabela[i].sledeci;
      return i;
}

int HashTable::umetni(int kljuc) {
      int ind = pronadji_ulaz(kljuc);
      if( tabela[ind].kljuc == kljuc ) return 0;
      int j;
      if( tabela[ind].kljuc == -1 ) j = ind;
      else  {
            while( tabela[slobodna].kljuc != -1 )
               if( --slobodna < 0 ) throw "Tabela je prepunjena";
            j = slobodna; tabela[ind].sledeci = slobodna;
      }
      tabela[j].kljuc = kljuc;
      return 1;
}

int HashTable::dohvati(unsigned int kljuc) {
      unsigned int i = pronadji_ulaz(kljuc);
      if( tabela[i].kljuc == kljuc ) return i;
      return -1;
}
```

```
ostream & operator<<(ostream &os, const HashTable &tabela) {
      for(int i = 0; i < tabela.velicina; i++)       {
         os << i << '\t' << tabela.tabela[i].kljuc << '\t' <<
                           tabela.tabela[i].sledeci;
         if( tabela.slobodna == i ) os << '\t' << "slob";
         os << endl;
      }
      return os;
}

// Glavni program
#include<iostream>
using namespace std;
#include "HashTable.h"
void main() {
HashTable tabela(20);
    try {
       for(int i = 5; i < 120; i+= 6)  tabela.umetni(i);
       cout << tabela;
       if( tabela.dohvati(10) == -1 ) {
         cout << "Kljuc 10 ne postoji" << endl;
         tabela.umetni(10);
       }
       if( tabela.umetni(10)==0) cout<<"Kljuc 10 vec postoji" <<endl;
         tabela.umetni(11); tabela.umetni(12);
      }
      catch(char *s) { cout << s << endl;  }
      cout << tabela;
}
```

12. UNUTRAŠNJE SORTIRANJE

Cilj sortiranja (uređivanja) podataka: rasporediti podatke u određenom poretku, odnosno tako da njihov redosled odražava određenu relaciju koja važi među njima.

Unutrašnje sortiranje se primenjuje nad podacima koji su su smešteni u operativnu memoriju računara.

Prilikom evaluacije kvaliteta algoritma za sortiranje, treba obratiti pažnju na sledeće osobine:
Performanse (od suštinskog značaja za praktičnu primenu algoritma)

- vremenska složenost poređenja elemenata
- vremenska složenost premeštanja ključeva
- upotreba dodatne memorije

Ostale osobine

- rekurzija
- stabilnost

Osobina stabilnosti ponekad može biti značajna prilikom izbora algoritma. Kaže se da je algoritam za sortiranje stabilan ako održava inicijalni redosled podataka koji su ekvivalentni po kriterijumu sortiranja. Na primer, niz kompleksnih brojeva se uređuje po rastućoj vrednosti njihovih modula. Neka dva kompleksna broja A i B imaju isti moduo, ali se u inicijalnom (nesortiranom) nizu kompleksan broj A nalazio pre (tj. na nižoj lokaciji) kompleksnog broja B. Algoritam za sortiranje je stabilan ako se nakon sortiranja broj A nalazi pre broja B.

Generalno, algoritmi za sortiranje se dele na dve grupe: one kod kojih se sortiranje vrši poređenjem vrednosti elemenata i ostale. Ako se sortiranje vrši poređenjem, potrebno je da postoji **tranzitivna** relacija, poput relacije *manje od* (<), nad podacima koji se sortiraju. Kod ostalih metoda se može pojaviti potreba za poređenjem elemenata, ali ona nije dominantna osobina, pa se zato te metode ne svrstavaju u metode poređenja.

Pregled nekih algoritama za sortiranje poređenjem elemenata (izvor: Wikipedia)

Name	Best	Average	Worst	Memory	Stable	Method
Bubble sort	$O(n)$	—	$O(n^2)$	$O(1)$	Yes	Exchanging
Cocktail sort	$O(n)$	—	$O(n^2)$	$O(1)$	Yes	Exchanging
Comb sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	No	Exchanging
Gnome sort	$O(n)$	—	$O(n^2)$	$O(1)$	Yes	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection
Insertion sort	$O(n)$	$O(n + d)$	$O(n^2)$	$O(1)$	Yes	Insertion
Shell sort	—	—	$O(n^{1.5})$	$O(1)$	No	Insertion
Binary tree sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)$	Yes	Insertion
Library sort	$O(n)$	$O(n\log n)$	$O(n^2)$	$O(n)$	Yes	Insertion
Merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	Yes	Merging
In-place merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	Yes	Merging
Heapsort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	No	Selection
Smoothsort	$O(n)$	—	$O(n\log n)$	$O(1)$	No	Selection
Quicksort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	No	Partitioning
Introsort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(\log n)$	No	Hybrid
Patience sorting	$O(n)$	—	$O(n\log n)$	$O(n)$	No	Insertion

Pregled nekih algoritama za sortiranje koji ne vrše sortiranje poređenjem elemenata (izvor: Wikipedia)

Name	Best	Average	Worst	Memory	Stable
Pigeonhole sort	$O(n+2^k)$	$O(n+2^k)$	$O(n+2^k)$	$O(2^k)$	Yes
Bucket sort	$O(n)$	$O(n)$	$O(n)$	$O(2^k)$	Yes
Counting sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	Yes
LSD Radix sort	$O(n \cdot k/s)$	$O(n \cdot k/s)$	$O(n \cdot k/s)$	$O(n)$	Yes
MSD Radix sort	$O(n \cdot k/s)$	$O(n \cdot k/s)$	$O(n \cdot (k/s) \cdot 2^s)$	$O((k/s) \cdot 2^s)$	No
Spreadsort	$O(n)$	$O(n \cdot k/\log(n))$	$O(n \cdot (k - \log(n))^{.5})$	$O(n)$	No

Pregled nekih algoritama za sortiranje koji nisu primenljivi u praksi ili je njihova softverska implementacija neefikasna (izvor: Wikipedia)

Name	Best	Average	Worst	Mem	Stable	Cmp	Other notes
Bogosort	$O(n)$	$O(n \times n!)$	unbounded	$O(1)$	No	Yes	Average time using Knuth shuffle
Bozo sort	$O(n)$	$O(n \times n!)$	unbounded	$O(1)$	No	Yes	Average time is asympotically half that of bogosort
Stooge sort	$O(n^{2.71})$	$O(n^{2.71})$	$O(n^{2.71})$	$O(1)$	No	Yes	
Bead sort	$O(n)$	$O(n)$	$O(n)$	—	N/A	No	Requires specialized hardware
Pancake sorting	$O(n)$	$O(n)$	$O(n)$	—	No	Yes	Requires specialized hardware
Sorting networks	$O(\log n)$	$O(\log n)$	$O(\log n)$	—	Yes	Yes	Requires a custom circuit of size $O(n\log n)$

Zadatak 12.1

Opisati sortiranje primenom metoda umetanja sa smanjenjem inkrementa (*shell sort*). Objasniti na čemu se zasniva efikasnost ovog metoda, kako se bira sekvenca inkremenata i kolika je složenost metoda. Ilustrovati rad algoritma pri sortiranju niza 19, 61, 42, 31, 7, 95, 77 i 25 u tri iteracije sa efikasnim izborom inkrementa.

Rešenje:

```
SHELL-SORT(a, h)
for i = 1 to t do
  inc = h[i]
  for j = inc + 1 to n do
    y = a[j]
    k = j – inc
    while (k ≥ 1) and (y < a[k]) do
      a[k + inc] = a[k]
      k = k – inc
    end_while
  a[k + inc] = y
  end_for
end_for
```

Shell sort pokušava da popravi nedostatke algoritma *insertion sort*:

1. insertion sort najbolje radi kada je niz (gotovo) uređen
2. loše performanse duguje višestrukom pomeranju podataka za jedno mesto (korak)
3. **ideja**: napraviti više prolaza, najpre sa većom vrednošću koraka a zatim sve manjom dok se niz ne svede na situaciju koja je pogodna za *insertion sort*

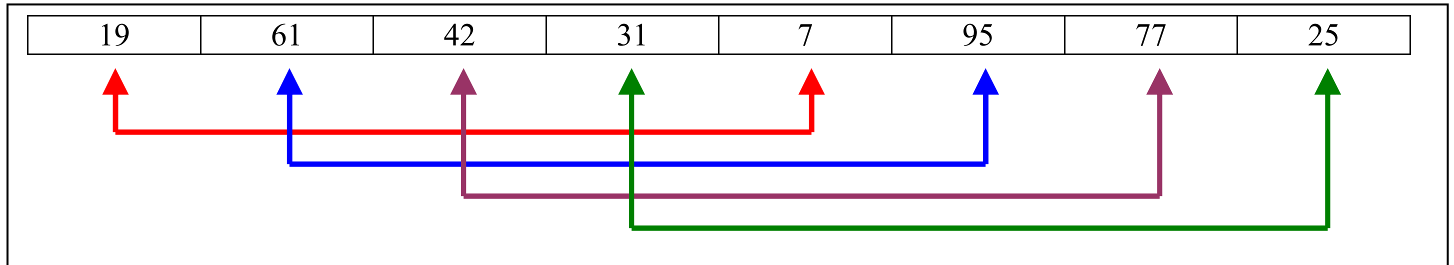
Performanse ovog algoritma **zavise od sekvence inkremenata**. Generalno: sekvenca $h_1,h_2,h_3,...,h_t$ može biti proizvoljna, ali mora da ispuni sledeća dva uslova: (1) $h_{i+1} < h_i$ i (2) $h_t = 1$

Predložena optimalna sekvenca, empirijski utvrđena: 1, 4, 10, 23, 57, 132, 301, 701 (Marcin Ciura, Best Increments for the Average Case of Shellsort, 13th International Symposium on Fundamentals of Computation Theory, 2001)

Performanse: najgore $O(n^2)$, prosečno oko $O(n^{1.3})$ (eksperimentalno utvrđeno). Još uvek nije dokazano da li algoritam može da postigne vremensku složenost O(n log n).

Odabrana sekvenca inkremenata : 4, 2, 1

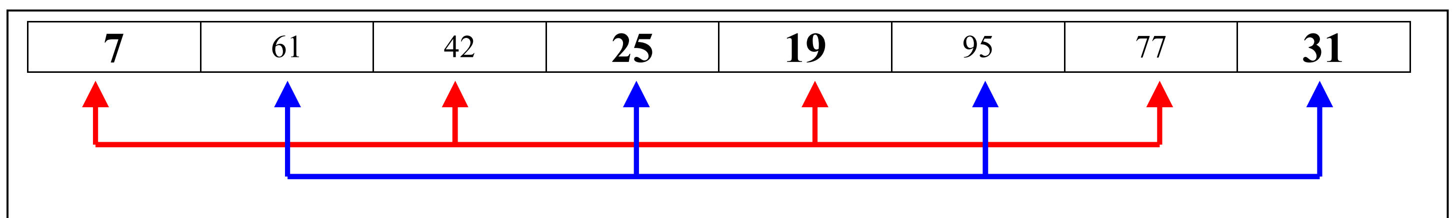
$h_1 = 4$



Pristup algoritma je kao i kod algoritma *insertion sort*, s tim da se elementi niza nad kojim se vrši sortiranje najpre grupišu u podnizove nad kojima se vrši inicialno sortiranje, sa ciljem da se pre sortiranja celog niza podaci urede što je moguće više pre završne faze algoritma koja je zapravo insertion sort. Cilj svake faze pre završne je da sve elemente niza postavi blizu njihove stvarne lokacije nakon sortiranja, izbegavajući time situacije kada je *insertion sort* neefikasan.

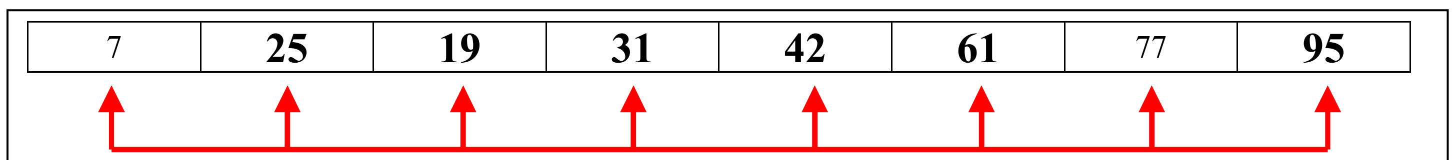
Za odabranu sekvencu inkremenata, mogu se uočiti četiri podniza: 1-5, 2-6, 3-7 i 4-8. U ovoj fazi se vrši sortiranje u okviru podnizova. Drugim rečima, zasebno će se sortirati elementi na lokacijama 1 i 5, zasebno elementi na lokacijama 2 i 6, itd. Na primer, na lokaciji 1 se nalazi broj 19 a na lokaciji 5 broj 7, pa će zbog toga brojevi 7 i 19 zameniti mesto (7 će biti smešten na lokaciju 1, 19 na lokaciju 5).

$h_2 = 2$

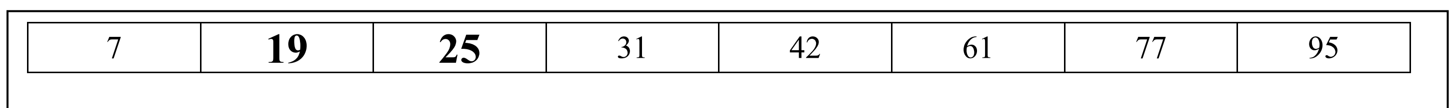


U ovoj fazi postoje dva podniza: 1-3-5-7 i 2-4-6-8 u okviru kojih se vrši sortiranje.

$h_3 = 1$ (završna faza)



Nakon sortiranja

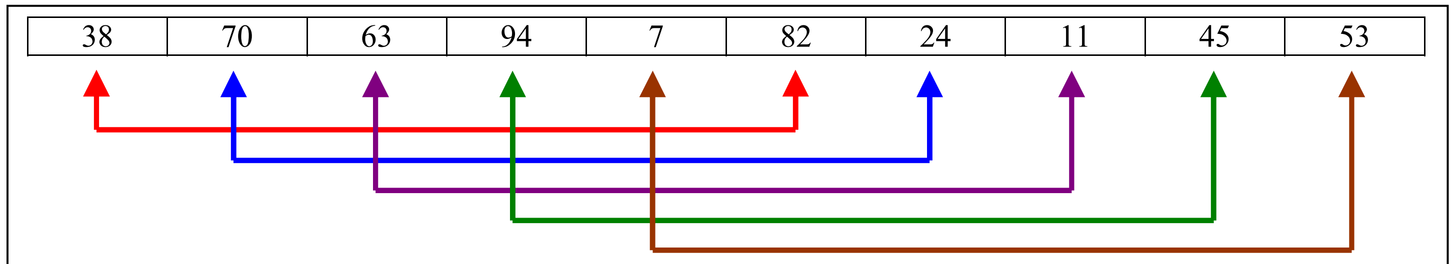


Zadatak 12.2

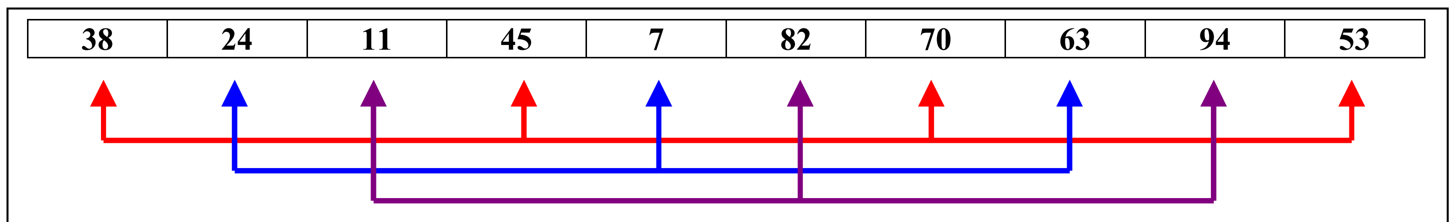
Opisati sortiranje primenom metoda umetanja sa smanjenjem inkrementa (*shell sort*). Objasniti na čemu se zasniva efikasnost ovog metoda, kako se bira sekvenca inkremenata i kolika je složenost metoda. Ilustrovati rad algoritma pri sortiranju niza 38, 70, 63, 94, 7, 82, 24, 11, 45 i 53 u tri iteracije za sledeće vrednosti inkremenata: h_1=5, h_2=3, h_3=1

Rešenje:

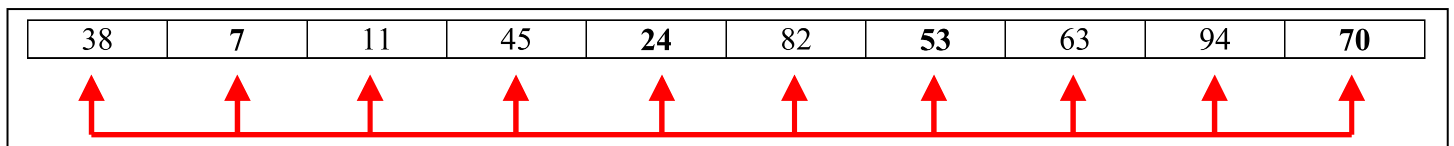
$h_1 = 5$



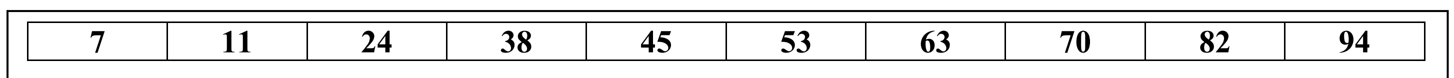
$h_2 = 3$



$h_3 = 1$



Nakon sortiranja:



Za razliku od zadatka 12.1, gde je u poslednjoj fazi izvršena samo jedna zamena, u ovom zadatku je u poslenjoj fazi premešten svaki element niza. Ipak, prve dve faze su generalno premestile elemente manje vrednosti bliže početku niza a elemente veće vrednosti bliže kraju niza, čime je obezbeđeno da u poslednjoj fazi dužina pomerenih nizova bude relativno mala.

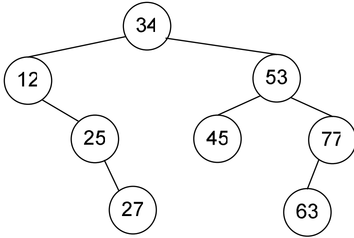
Zadatak 12.3

Ilustrovati postupak sortiranja korišćenjem stabla binarnog pretraživanja na primeru sortiranja niza ključeva 34, 53, 45, 77, 63, 12, 25 i 27. Kako se rešava problem istih ključeva? Koja je složenost ovog postupka?

Rešenje:
Ključevi se umeću u stablo redom u kojem su dati u ulaznom nizu. *INORDER* obilazak stabla daje ključeve u neopadajućem poretku.

Problem istih ključeva se rešava:

- umetanjem ponovljenog ključa u desno podstablo (zašto?)
- ulančavanjem ključeva u datom čvoru



Složenost: najgora $O(n^2)$, najbolja O(n log n), prosečna O(n log n)
Loše osobine:

1. potrebna dodatna memorija
2. loš za pretežno monotone nizove (zašto?)

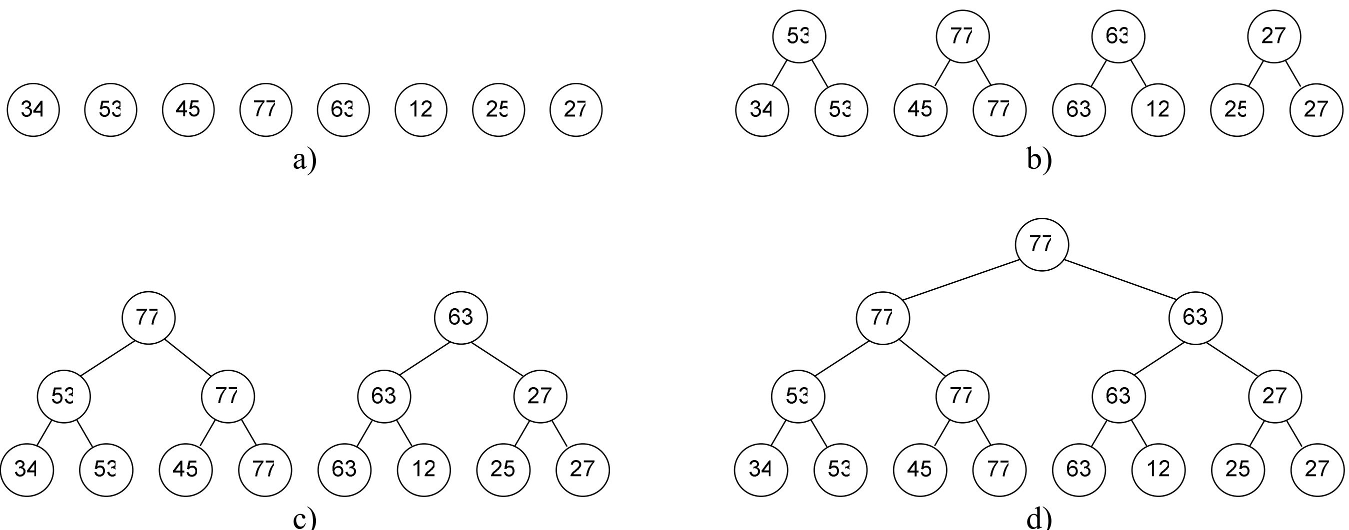
Iako je prosečna vremenska složenost O(n log n), jedna od mana algoritma je ta što ničim ne garantuje da neće doći do najgoreg slučaja, kada je složenost neprihvatljiva - $O(n^2)$.

Zadatak 12.4

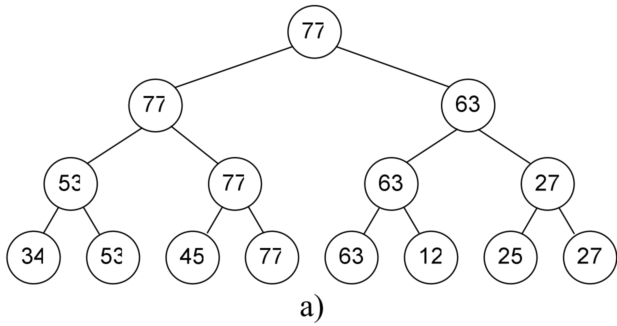
Ilustrovati postupak sortiranja korišćenjem stabla selekcije na primeru sortiranja niza ključeva 34, 53, 45, 77, 63, 12, 25 i 27

Rešenje:
Stablo selekcije je vrsta binarnog stabla koja generalizuje princip kvadratne selekcije, koja se može iskoristiti za poboljšanje performansi metode direktne selekcije (*selection sort*). Generalno, metoda direktne selekcije u nesortiranom delu niza pronalazi element koji se smešta na kraj sortiranog dela niza. Kvadratna vremenska složenost ovog pristupa potiče od činjenice da se linearno pretražuje ceo nesortirani niz da bi se pronašao odgovarajući element koji treba smestiti na kraj sortiranog niza. Metoda se može ubrzati tako što bi se nesortirani deo niza podelio na nekoliko podnizova, redukujući time prostor pretrage: element koji treba dodati na kraj sortiranog niza se traži na nivou podnizova. Ova činjenica predstavlja osnovnu ideju za stablo selekcije. Za razliku od binarnog stabla pretraživanja, ova metoda **nije osetljiva na inicijalnu uređenost niza**.

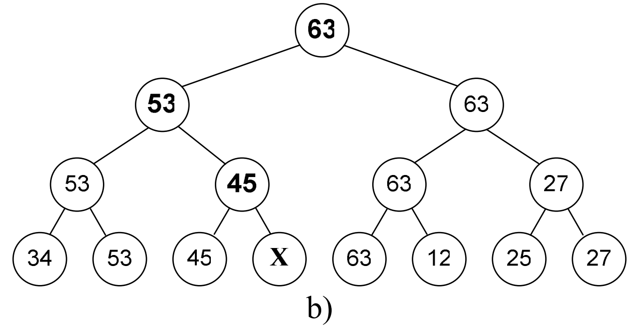
Generisanje stabla polazi od ulaznog niza, tako što ključevi postaju listovi. Listovi se grupišu u parove i iz svakog para listova se ka korenu propagira onaj ključ koji ima najveću vrednost. Postupak formiranja stabla je prikazan na sledećoj slici:



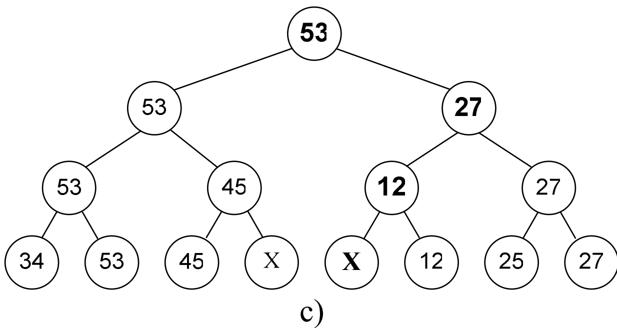
Nakon formiranja stabla, sortiranje se vrši uzimanjem ključa iz korena: u korenu se nalazi najveći ključ. U zavisnosti od načina sortiranja niza (rastuće, opadajuće), uzeti ključ treba smestiti na kraj ili na početak rezultujućeg (sortiranog) niza. Nakon uzimanja ključa iz korena, on se uklanja iz stabla. Zbog načina formiranja stabla, ključ koji se nalazi u korenu nalazi se i na svakom nivou stabla, pa je potrebno izvršiti ažuriranje stabla, počevši od lista u kojem se takođe nalazio uklonjeni ključ. Umesto uklonjenog ključa, u list treba smestiti neku karakterističnu vrednost koja označava da je taj čvor obrađen i da nije od interesa u daljem procesu sortiranja. U ovom zadatku je ta karakteristična vrednost obeležena simbolom X. Prilikom implementacije ovog algoritma, može se upotrebiti vrednost -∞, jer je ona najmanja i garantovano neće biti propagirana do vrha stabla.
Na sledećoj slici je prikazan izgled stabla nakon svakog uklanjanja ključa.



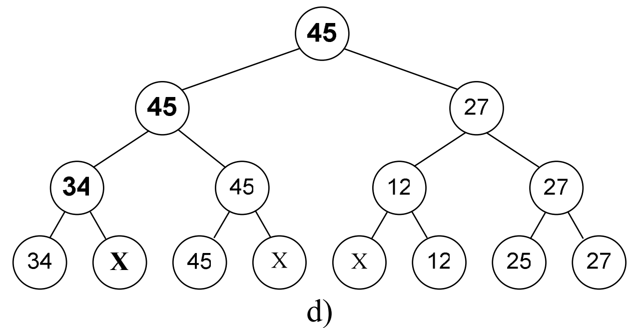
a)



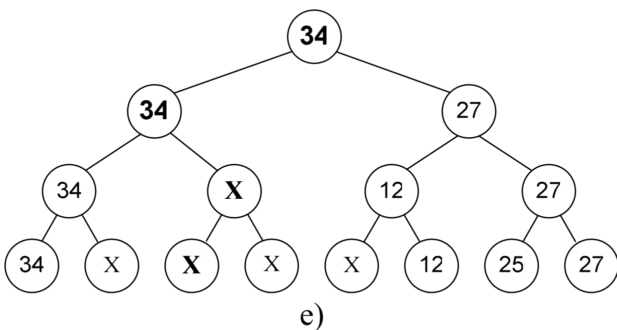
b)



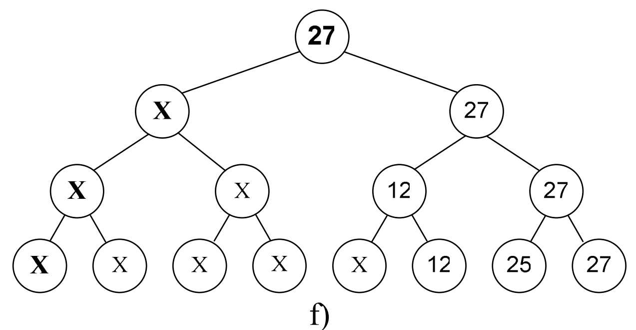
c)



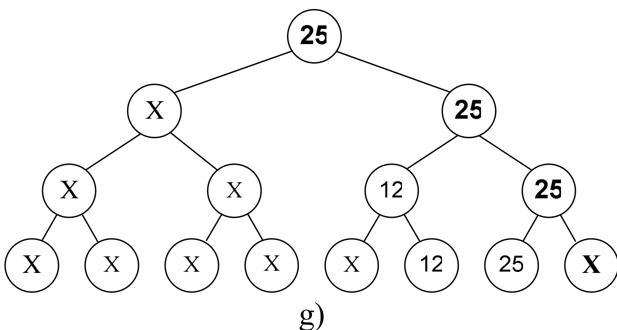
d)



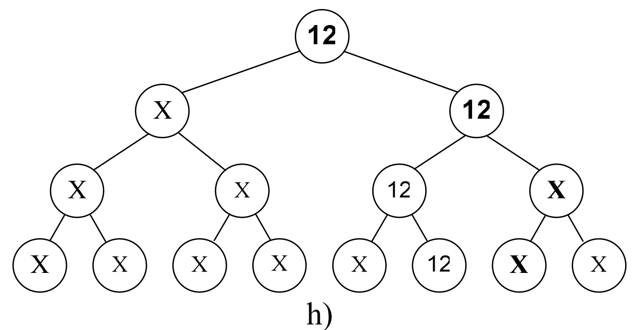
e)



f)



g)



h)

Da li je ovaj algoritam za sortiranje stabilan? Kako bi trebalo postupiti da bi se garantovala njegova stabilnost?

Zadatak 12.5

Precizno objasniti postupak rada algoritma sortiranja *heapsort*. Usvojiti pogodnu memorijsku reprezentaciju i navesti njene prednosti. Za usvojenu strukturu demonstrirati rad algoritma po koracima pri sortiranju niza: 57, 42, 69, 11, 35, 28, 7 i 19 u neopadajućem poretku. Grubo izvesti perfomanse u najgorem i prosečnom slučaju.

Rešenje:

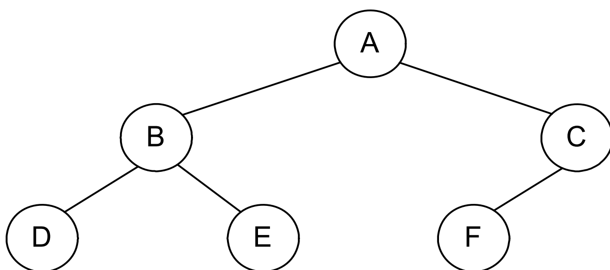
Heapsort je efikasan algoritam za sortiranje, zasnovan na strukturi podataka zvanoj *heap* (eng. gomila).

Heap je vrsta binarnog stabla kod kojeg je vrednost u roditeljskom čvoru veća od vrednosti smeštenim u njegove sinove. Iz ovakve definicije neposredno sledi da je najveća vrednost smeštena u koren stabla. Hip je skoro kompletno stablo. Kod skoro kompletnog stabla su kompletno popunjeni svi nivoi osim poslednjeg. Poslednji nivo može biti delimično popunjen, ali je popunjavanje obavezno izvršeno s leva udesno. Drugim rečima, ako neki unutrašnji čvor stabla ima desnog sina, onda sigurno ima i levog. Takođe ako neki unutrašnji čvor stabla ima barem jednog potomka, to znači da njegov levi sused (ako postoji) ima oba potomka.

Značajna osobina skoro kompletnih stabala, koja se koristi u algoritmu *heapsort*, zasniva se na činjenici da se ovakvo stablo može efikasno smestiti u niz, na osnovu sledećih relacija:

- indeks roditeljskog čvora od čvora indeksa A je A div 2
- indeks levog potomka od čvora indeksa A je 2*A
- indeks desnog potomka od čvora indeksa A je 2*A+1
- koren stabla ima najmanji indeks (prvi element niza).

Za nizove koji se indeksiraju počevši od 0, ove relacije su delimično izmenjene. Sledeća slika prikazuje binarno stablo sa 6 ključeva i način njegovog smeštanja u niz.



1	*2*	*3*	*4*	*5*	*6*
A	B	C	D	E	F

Opis algoritma:

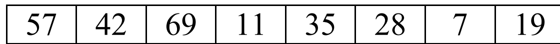
1) Neuređen niz se najpre preuredi u hip. Sve izmene se vrše nad nizom koji se uređuje, pa se ne koristi dodatni prostor. Preuređivanje u hip se vrši zamenom mesta elemenata niza, počevši od prvog elementa niza (odnosno korena stabla) tako da budu ispoštovani uslovi odnosa vrednosti roditelj-potomak koji važe u hipu.
2) Sve dok se ne obradi svih n elemenata ulaznog niza raditi sledeće
   - vrednost u korenu menja mesto sa poslednjim elementom nesortiranog dela niza, odnosno onim delom niza koji je još uvek u hipu
   - nova vrednost u korenu se propagira niz stablo sve dok ne uspostavi odnos roditelj-potomak koji važi u hipu

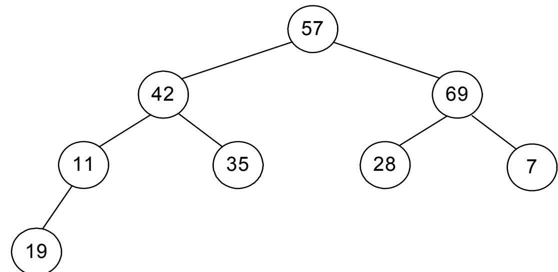
Vremenska složenost je generalno O(n log n), a najbolji, najgori i prosečan slučaj se razlikuju za multiplikativnu konstantu.

Algoritam **nije stabilan.**

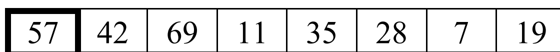
Formiranje hipa:

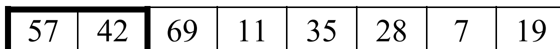
Na osnovu ranijeg primera, stablo koje se dobija od zadatog niza je prikazano na sledećoj slici. Ono, međutim ne zadovoljava uslove hipa (desni potomak čvora sa ključem 57 sadrži ključ 69 što nije dozvoljeno; slično važi i za ključeve 11 i 19), što se ne može jednostavno videti u datom nizu.

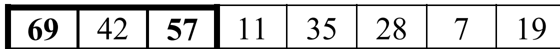




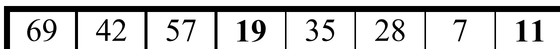
Preuređivanje niza počinje od prvog elementa niza (korena) i redom prolazi kroz sve elemente niza i po potrebi zamenjuje mesta datom elementu i elementu u njegovom roditeljskom čvoru, kao što je to prikazano na sledećoj sekvenci:



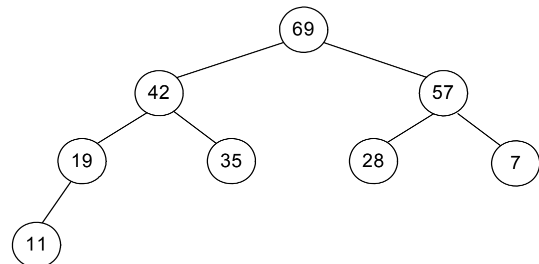




…



Rezultujuće stablo:

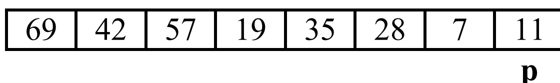


Sortiranje niza:

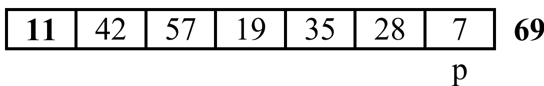
Ključ koji se nalazi u korenu (A) razmenjuje mesto sa poslednjim ključem u nesortiranom nizu (P). Čvor gde se smešta ključ (A) predstavlja sortirani deo niza i izuzima se iz daljeg razmatranja (grafički se to predstavlja njegovim razvezivanjem od ostatka stabla).

Za ključ (P), koji se smešta u koren stabla, mora da se proveri da li zadovoljava uslove hipa. Ako ne zadovoljava, taj ključ se propagira ka listovima stabla, a propagacija se završava kada su uslovi hipa zadovoljeni.

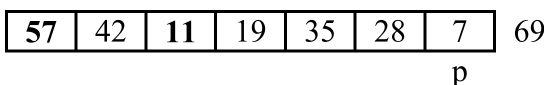
Sledeće slike ilustruju postupak sortiranja:



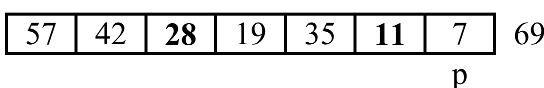
Ključ u korenu (69) menja mesto sa ključem na kraju nesortiranog dela niza (indeksira se promenljivom p).



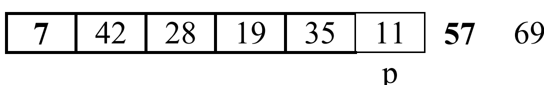
Nakon smeštanja ključa na kraj niza, indeks p se smanjuje za 1. Ključ 69 se sada nalazi u sortiranom delu niza.



Ključ koji je smešten u koren stabla je manji od ključeva u sinovima korena. Tada on menja mesto sa većim ključem (u ovom slučaju ključem 57).



Slična situacija je i sa ključem 28. Nakon ove izmene, zadovoljeni su uslovi hipa u okviru nesortiranog dela niza.



Ključ u korenu zamenjuje mesto sa ključem na kraju sortiranog niza

42	**35**	28	19	**7**	11	57	69
					p		

Nakon preuređivanja niza da bi se zadovoljili uslovi hipa, dobija se ovakva situacija

35	**19**	28	**11**	7	**42**	57	69
				p			

28	19	**7**	11	**35**	42	57	69
			p				

19	**11**	7	**28**	35	42	57	69
		p					

11	**7**	**19**	28	35	42	57	69
	p						

7	**11**	19	28	35	42	57	69
p							

Prednost algoritma *heapsort* u odnosu na stablo selekcije:

- nije potreban dodatni memorijski prostor za konstrukciju stabla, jer se sve izmene vrše nad originalnim nizom
- nema replikacije ključeva, a binarno stablo može da se formira nad osnovnim nizom čime se postiže efikasnija implementacija
- prilikom sortiranja ne vrše se ponekad nepotrebna ažuriranja i poređenja

Kako bi trebalo izmeniti algoritam *heapsort* da bi vršio sortiranje po nerastućem poretku?

Zadatak 12.6

Realizovati klasu `Heap` u programskom jeziku C++ koja implementira funkcionalnosti prioritetnog reda celih brojeva koristeći heap kao strukturu podataka. Koristeći realizovane operacije, realizovati i metodu koja sortira cele brojeve metodom *Heapsort*. Koja je složenost operacija umetanja i brisanja iz tako realizovanog prioritetnog reda, a koja složenost algoritma sortiranja?

Rešenje:

```
// heap.h
#ifndef HEAP_H
#define HEAP_H
#include <iostream>
using namespace std;

class Heap{
public:
	Heap(int n);
	Heap(int *niz, int n){
		capacity=n;
```

```
        size=0;
        this->niz=niz;
        formHeap();
    }
    int getCapacity() const { return capacity;   }
    int getSize() const { return size; }
    int PQGet() const { return niz[0]; }
    int PQDelete();
    void PQInsert(int k);
    static void sort(int *niz, int n){
        Heap h(niz,n);
        niz=h.sort();
    }
    virtual ~Heap(){delete [] niz; }

protected:
    int* niz;
    int capacity;
    int size;

    void formHeap(int n);
    void formHeap() { formHeap(capacity); }
    static int leftSon(int i) const { return (i+1)*2-1; }
    static int rightSon(int i) const { return (i+1)*2; }
    static int father(int i) const { return (i-1)/2; }
    int* sort();
    friend ostream & operator<<(ostream &os, const Heap &heap);
};
#endif

// heap.cpp
#include "heap.h"

Heap::Heap(int n){
   niz = new int[n];
   capacity=n;
   size=0;
}
void Heap::formHeap(int n){
    if(n<1) return;
    if (n>capacity) n=capacity;
    size=1;
    for(int i=1; i<n; i++) PQInsert(niz[i]);
}

int Heap::PQDelete(){
    int first=niz[0];
    niz[0]=niz[--size];
    int f=0;
    while(f<size-1){
        int s1=leftSon(f), s2=s1+1;
        if(s1>size-1) break;
```

```
          int x=s1, y=niz[s1];
          if(s2<=size-1 && y<niz[s2])
          {
               x=s2;
               y=niz[s2];
          }
          if(niz[f]>=y) break;
          niz[x]=niz[f];
          niz[f]=y;
          f=x;
     }
     return first;
}

void Heap::PQInsert(int k){
    int f=father(size);
     int  s=size++;
     while(s>0 && niz[f]<k){
          niz[s]=niz[f];
          s=f;
          f=father(f);
      }
     niz[s]=k;
}

int* Heap::sort(){
     while(size>0){
          int first=PQDelete();
          niz[size]=first;
     }
     int *temp=niz; niz=NULL;
     return temp;
}

ostream & operator<<(ostream &os, const Heap &heap) {
     for(int i=0; i<heap.getSize(); i++){
          os<<heap.niz[i]<<"  ";
          if(i%15==14) os<<endl;
     }
     os<<endl;
    return os;
}
```

```
// test.cpp
#include <iostream>
#include <ctime>
#include "heap.h"
using namespace std;

void main(){
     int n=1000000, i;
     int *niz=new int[n];
```

```
    srand((unsigned)time(0));
    for(i=0; i<n; i++) niz[i]=rand()%10000;
    Heap::sort(niz,n);
    bool sorted=true;
    for(int i=0; i<n-1; i++)
     if(niz[i]>niz[i+1]) {
         sorted=false;
         break;
    }
    if(sorted) cout<<"sortiran"<<endl;
    else cout<<"nesortiran"<<endl;
    delete [] niz;
}
```

Heap se formira tako što se, počev od drugog elementa niza, dodaje jedan po jedan ključ i održava struktura heap-a (prvi element se preskače, jer je sam za sebe heap veličine jednog elementa). Ključ se u heap umeće tako što se inicijalno doda na kraj vektora u koji je smešten heap, odnosno kao krajnji desni list najdubljeg nivoa stabla, a potom propagira ka vrhu stabla pomerajući sve čvorove koji su manji za jedno mesto prema dnu stabla, dok ne nađe svoje mesto. Na taj način se ujedno realizuje operacija `PQInsert` prioritetnog reda.

Prvi element heap-a će uvek predstavljati prvi element koji treba izvaditi iz prioritetnog reda, jer je najveći. U ovoj implementaciji, on se dohvata metodom `PQGet()`, a briše se metodom `PQDelete()`. Posle brisanja, na vrh heap-a dolazi poslednji element heap-a, a potom se vrši njegova propagacija ka dnu stabla dok ne nađe svoje mesto. Sortiranje se vrši u statičkoj metodi klase `Heap` koja prvo formira heap od niza koji joj se prosleđuje kao argument, potom u petlji uzima jedan po jedan element iz heap-a i dodaje ga na kraj niza.
Iz ovoga je jednostavno zaključiti da je složenost traženih operacije logaritamska, a složenost operacije dohvatanja prvog elementa je konstantna. Složenost sortiranja metodom *Heapsort* je O(n log n).

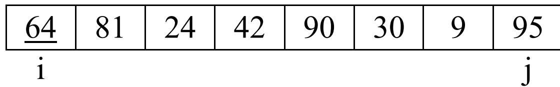
Zadatak 12.7

Objasniti postupak i realizaciju algoritma particijskog sortiranja (*quicksort*). Demonstrirati algoritam na primeru sortiranja neuređenog niza 64, 81, 24, 42, 90, 30, 9 i 95. Za pivot izabrati prvi element u particiji. Izvesti performanse u najboljem i najgorem slučaju. Kako se može izbeći najgori slučaj?

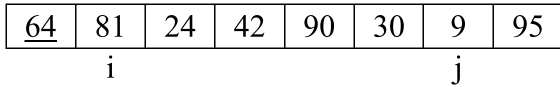
Rešenje:
Quicksort se bazira na strategiji "podeli i osvoji" (divide and conquer) : složen problem se razlaže na nekoliko jednostavnijih potproblema koji su po prirodi isti kao i složen (osnovni problem). Novi potproblemi se dalje razlažu na još jednostavnije potprobleme i proces razlaganja se nastavlja sve dok rezultujući potproblem nije trivijalan za rešavanje.
Generalna ideja: podeliti osnovni niz na dva podniza razdvojenih elementom koji se garantovano nalazi na svojoj konačnoj poziciji (poziciji koju će zauzimati nakon sortiranja) i primeniti ovaj postupak na dobijene podnizove.

```
QUICKSORT(a, low, high)
if (low ≥ high) then return
j = PARTITION(a, low, high)
QUICKSORT(a, low, j - 1)
QUICKSORT(a, j + 1, high)
```

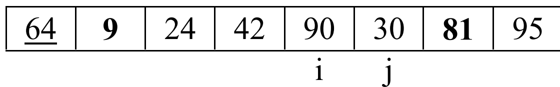
Quicksort se najjednostavnije formuliše u svom rekurzivnom obliku. Funkcija PARTITION preuređuje osnovni niz i vraća indeks pozicije koja razdvaja dva podniza. **PIVOT** je element koji razdvaja podnizove nakon preuređivanja: svi elementi pre pivota su manji ili jednaki pivotu, svi nakon pivota veći ili jednaki. **Izbor pivota utiče** na performanse algoritma.
U postavci je rečeno da se za pivot bira prvi element particije. Na sledećim slikama koje ilustruju rad algoritma, pivot je podvučen. Na početku, ceo niz je jedna particija.



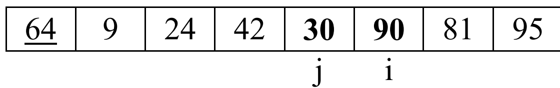
Funkcija PARTITION se generalno sastoji od tri ciklusa: jedan (spoljašnji) koji se ponavlja dok se indeksi **i** i **j** ne susretnu ili mimoiđu (**i** polazi od početka particije, **j** sa kraja), i dva unutrašnja ciklusa, od kojih jedan inkrementira **i** sve dok je i-ti element manji ili jednak pivotu, a drugi dekrementira **j** sve dok je **j**-ti element veći od pivota.



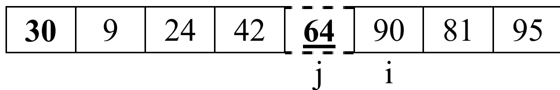
i-ti element je veći od pivota, a **j**-ti manji. U takvoj situaciji, ako je **i** manji od **j** (još uvek se nisu susreli), ključevi pod indeksima **i** i **j** menjaju mesto.



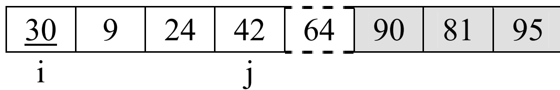
Nakon učinjene zamene, proces konvergiranja **i** i **j** se nastavlja sve do susretanja ili mimoilaženja. Kada god se desi da je **i**-ti element veći od pivota a **j**-ti manji, oni menjaju mesto



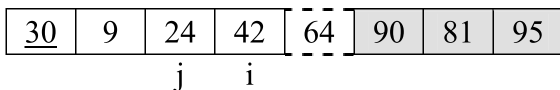
Kada se **i** i **j** susretnu ili mimoiđu, j indeksira mesto gde se nalazi poslednji element manji ili jednak pivotu.
Tada pivot i **j**-ti element menjaju mesto.



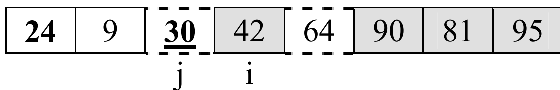
Pivot se sada nalazi na svojoj konačnoj poziciji: svi elementi levo od njega su manji, svi ostali su veći. Tako su dobijene dve neuređene particije. Ovime se završava funkcija PARTITION, koja kao rezultat vraća indeks j gde je smešten pivot.

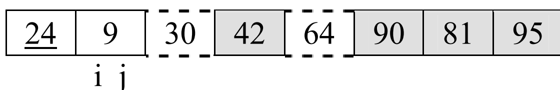


Čitav proces se ponavlja za donju a zatim za gornju particiju

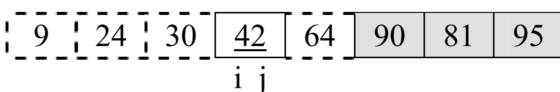


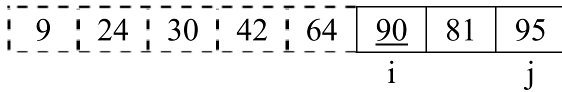
i i **j** su se mimoišli (nije bilo zamena): pivot i **j**-ti element menjaju mesto.

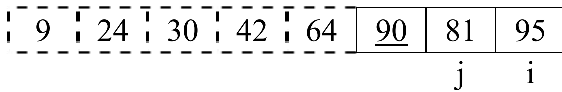


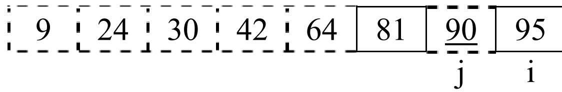


i i **j** su se susreli (nije bilo zamena): pivot i **j**-ti element menjaju mesto.









Performanse algoritma *quicksort*:

- najbolji slučaj: O(n log n)
- najgori slučaj: O(n^2), nastaje kada je u svakom koraku pivot inicijalno na svom mestu i nakon svakog particionisanja postoji samo jedna rezultujuća particija. Na primer: ulazni niz je već sortiran neopadajuće, a pivot je prvi element particije.

Poboljšanje ovog metoda: omogućiti fleksibilniji izbor pivota

- izabere se slučajan element tekuće particije
- bira se jedan od nekoliko članova particije

Da li je metod stabilan? Kako bi trebalo modifikovati funkciju PARTITION da bi se vršilo sortiranje po nerastućem redosledu?

Zadatak 12.8

U koju grupu metoda unutrašnjeg sortiranja spada pobitno razdvajanje (*radix exchange*)? Objasniti postupak i demonstrirati ga na primeru neuređenog niza: 10 21 8 15 27 18 12 30.

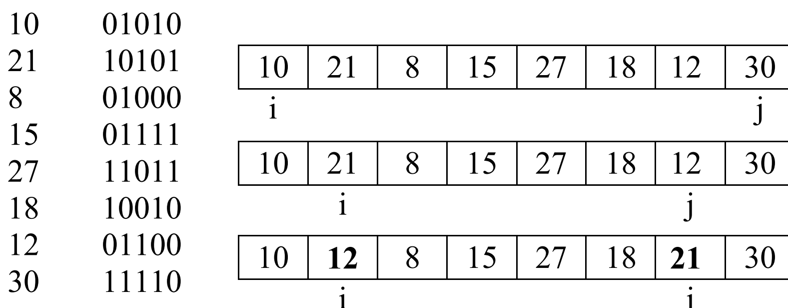
Rešenje:

Pobitno razdvajanje spada u metode koje porede ključeve, a osnovni način sortiranja je zamena pozicije ključeva za koje se utvrdi da su u nepravilnom poretku. Karakteriše ga posmatranje ključeva u njihovoj binarnoj reprezentaciji. Ulazni niz se deli na particije, slično algoritmu *quicksort*. Umesto poređenja ključeva, porede se pojedinačni bitovi njihove binarne reprezentacije, počevši od bitova najveće težine.

Inicijalno, ceo niz se tretira kao jedna particija. Unutar date particije, ključevi se razdvajaju na dve particije:

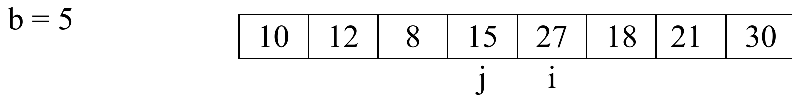
- ključevi sa posmatranim bitom vrednosti 0 (donja particija)
- ključevi sa posmatranim bitom vrednosti 1 (gornja particija)

Procedura se ponavlja sve dok veličina particije nije 1 ili dok se ne obrade svi biti u binarnoj reprezentaciji.



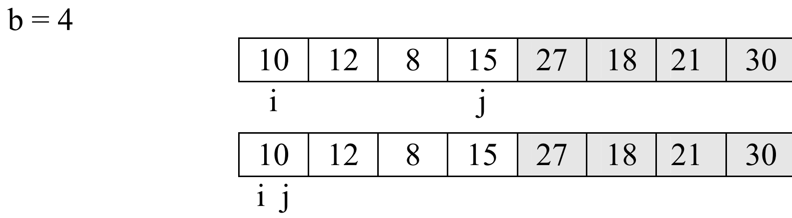
U prvom prolazu algoritma se posmatra bit najveće težine (b=5). Slično funkciji PARTITION kod *quicksort* algoritma, indeks **i** se povećava sve dok je posmatrani bit **i**-tog elementa 0, a indeks **j** se smanjuje sve dok je posmatrani bit **j**-tog elementa 1. U ovom slučaju, prvo se nailazi na elemente 21 i 12, koji menjaju mesto.

b = 5



Nakon mimoilaženja **i** i **j**, **j** predstavlja gornju granicu elemenata kod kojih posmatrani bit ima vrednost 0, a **i** predstavlja donju granicu za one kod kojih on ima vrednost 1. Time su razdvojene dve particije.

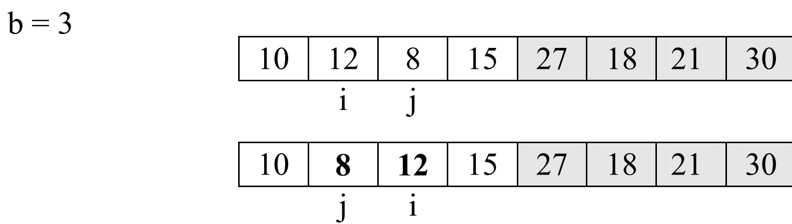
b = 4



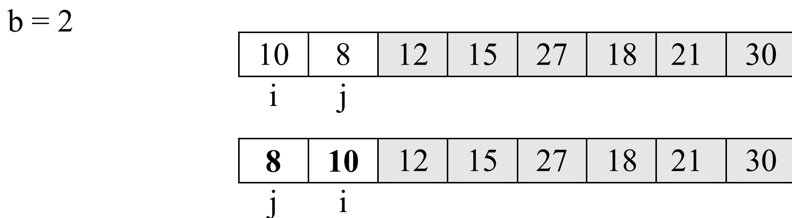
Nakon razdvajanja na particije, ista procedura se primenjuje na svaku od particija, s tim što se tada posmatra naredni bit, manje težine.

U ovom slučaju, bit 4 svih elemenata donje particije ima vrednost 1, pa se ova particija ne redukuje.

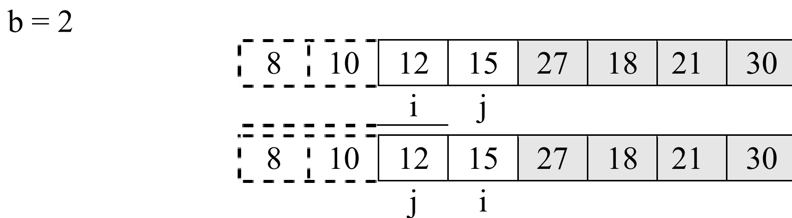
b = 3



b = 2

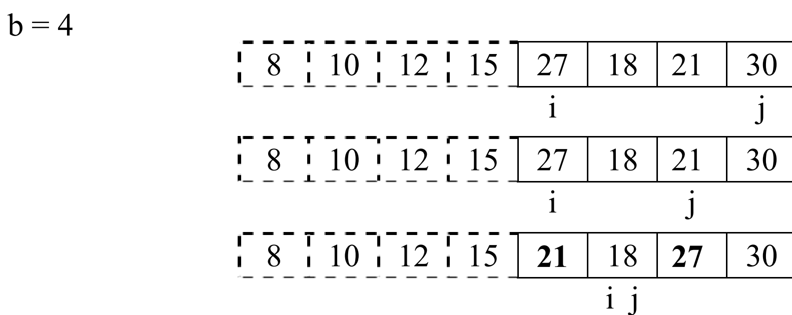


b = 2

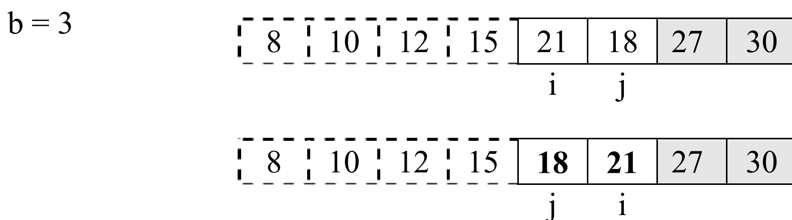


Nakon završenog sortiranja donje particije, vrši se sortiranje gornje particije

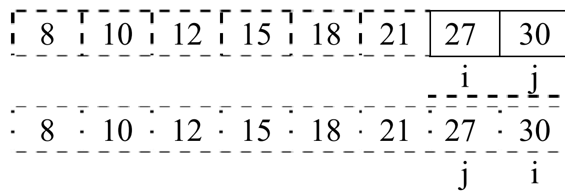
b = 4



b = 3



b = 3



Primetiti da je ovde sortiranje završeno pre nego što su provereni svi bitovi u binarnoj reprezentaciji ključeva.

Zadatak 12.9

Objasniti algoritam sortiranja brojanjem (*counting sort*). Demonstrirati rad algoritma po koracima na primeru sortiranja niza 3, 5, 1, 7, 3, 5, 4, 3. Pretpostaviti da je opseg ključeva 1..10. Da li je metod stabilan?

Rešenje:

Sortiranje brojanjem spada u metode kod kojih se sortiranje ne vrši poređenjem ključeva. Zasniva se na pretpostavci da je skup mogućih ključeva unapred poznat i da se njihove vrednosti nalaze u unapred poznatom opsegu.

Sortiranje brojanjem se izvršava u više faza:

1. brojanje koliko puta se određen ključ nalazi u nizu
2. određivanje gornje granice pozicije gde će ključ biti smešten nakon sortiranja
3. smeštanje ključeva na odgovarajuće pozicije

Algoritam teoretski ima vremensku složenost O(n+k) = O(n), ali pod uslovom da je broj ključeva k = O(n). Zahteva dodatni prostor za niz brojača koji koristi kao osnovno sredstvo za sortiranje i dodatni prostor za rezultujući (sortirani) niz.

Neka je niz koji se sortira označen sa A

A:

3	5	1	7	3	5	4	3

1. Faza: računa se ukupan broj pojavljivanja svakog ključa u nizu A. Rezultati brojanja se smeštaju u niz C, čija je dužina jednaka broju različitih ključeva koji se mogu pojaviti u nizu A. U postavci je naznačeno da je moguć opseg ključeva 1..10, što znači da je niz C dužine 10.

Nakon završetka ove faze, **i**-ti element niza C označava koliko puta se ključ **i** pojavio u nizu A.

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	0	3	1	2	0	1	0	0	0

2. Faza: određuje se gornja granica za poziciju ključeva koji imaju vrednost i. Ova faza se takođe obavlja nad nizom C, a svodi se na sekvencijalni prolazak kroz niz C, počevši od drugog elementa, uz obavljanje sledeće dodele: C[i]=C[i]+C[i-1]

Nakon završetka ove faze, niz C izgleda ovako

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	1	4	5	7	7	8	8	8	8

Na primer, u ulazu 7 niza C stoji vrednost 8. To znači da je 8 najviša pozicija na kojoj se može naći ključ 7 u rezultujućem (sortiranom) nizu u slučaju da ima više ključeva vrednosti 7. Ako ima samo jedan takav ključ, onda će on biti smešten na poziciju 8.

```
COUNTING-SORT
for i=1 to k do
   C[i]=0
end_for
for j=1 to n do
   C[A[j]]=C[A[j]]+1
end_for
for i=2 to k do
   C[i]=C[i]+C[i-1]
end_for
for j=n downto 1 do
   B[C[A[j]]]=A[j]
   C[A[j]]=C[A[j]]-1
end_for
```

3. Faza: formiranje sortiranog niza B, sekvencijalnim prolazom kroz niz A od poslednjeg elementa ka početku niza (zašto?)

Najpre se u nizu A odabere poslednji element – to je ključ 3. Zatim se u nizu C pristupi ulazu broj 3 i vrednost koja se tu nalazi predstavlja ulaz u nizu B u koji treba smestiti ključ 3.

```
for i:= n downto 1 do begin
   kljuc := A[i]
   ulaz := C[kljuc]
   B[ulaz] := kljuc
   C[kljuc] := C[kljuc] - 1
end
```

A

3	5	1	7	3	5	4	**3**

C

1	*2*	***3***	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	1	4	5	7	7	8	8	8	8

B

			3				

Dekrementiranje vrednosti `C[kljuc]` se vrši zbog toga što niz A može da sadrži više ključeva iste vrednosti. Naredni ključ 3 neće biti smešten u lokaciju 4, već u lokaciju 3. Izmena na pomenutoj lokaciji se vidi na sledećoj slici.

U nizu A se zatim bira naredni element – to je ključ 4.
U ulazu pod brojem 4 u nizu C se nalazi vrednost 5.
To znači da ključ 4 treba smestiti u ulaz 5 u nizu B.

A

3	5	1	7	3	5	**4**	3

C

1	*2*	*3*	***4***	*5*	*6*	*7*	*8*	*9*	*10*
1	1	3	5	7	7	8	8	8	8

B

			3	**4**			

A

3	5	1	7	3	**5**	4	3

C

1	*2*	*3*	*4*	***5***	*6*	*7*	*8*	*9*	*10*
1	1	3	4	7	7	8	8	8	8

B

			3	4		**5**	

A

3	5	1	7	**3**	5	4	3

C

1	*2*	***3***	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	1	3	5	6	7	8	8	8	8

B

		3	3	4		5	

A

3	5	1	**7**	3	5	4	3

C

1	*2*	*3*	*4*	*5*	*6*	***7***	*8*	*9*	*10*
1	1	2	4	6	7	8	8	8	8

B

		3	3	4		5	**7**

A

3	5	**1**	7	3	5	4	3

C

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	1	2	5	6	7	7	8	8	8

B

1		3	3	4		5	**7**

A

3	**5**	1	7	3	5	4	3

C

1	*2*	*3*	*4*	***5***	*6*	*7*	*8*	*9*	*10*
0	1	2	4	6	7	7	8	8	8

B

1		3	3	4	**5**	5	7

A

3	5	1	7	3	5	4	3

C

1	*2*	***3***	*4*	*5*	*6*	*7*	*8*	*9*	*10*
1	1	2	5	6	7	7	8	8	8

B

1	**3**	3	3	4	5	5	**7**

Metoda jeste stabilna, zato što zadržava inicijalni poredak među istim ključevima. Koje modifikacije bi trebalo uraditi da se prolaz kroz niz A u 3. fazi ne vrši od kraja niza prema početku, već od početka niza A prema njegovom kraju?

Zadatak 12.10

Napisati funkciju linearne složenosti na programskom jeziku C za sortiranje niza celih brojeva koji su u intervalu [0, 10000] i glavni program za testiranje funkcije. Napisati glavni program koji prikazuje upotrebu napisane funkcije.

Rešenje:

Optimizovana varijanta algoritma *Counting sort* za sortiranje pozitivnih brojeva u memoriji može se realizovati na sledeći način:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define K 10000
void sort(int *niz, int n){
	int *c=calloc(K,sizeof(int)), i, j;
	for(i=0; i<n; c[niz[i++]]++);
	i=j=0;
	while(j<n){
		if(c[i]!=0){
			niz[j++]=i;
			c[i]--;
		}
		else i++;
	}
	free(c);
}
void main(){
	int n=5000000, i;
	int *niz=calloc(n,sizeof(int));
	srand(time(0));
	for(i=0; i<n; i++) niz[i]=rand()%K;
	sort(niz,n);
	free(niz);
}
```

Ideja realizovana u funkciji `sort` je slična kao u sortiranju brojanjem. Prvo se prebroji koliko kojih elemenata ima u polaznom nizu i ta informacija čuva u nizu `c`, a potom se samo na osnovu te informacije konstruiše sortirani niz. Redom se dodaje onoliko nula koliko ih ima u polaznom nizu, pa potom onoliko jedinica koliko ih ima u polaznom nizu itd. Funkcija se završava kada se konstruiše niz od n brojeva. Primetiti da se ne alocira dodatna memorija, potrebna za smeštanje sortiranog niza: sortiranje se vrši u zadatom nizu neuređenih brojeva.

Obratiti pažnju na činjenicu da se ovo sortiranje odnosi na sortiranje brojeva, a ne ključeva, pa je nemoguće (i nema smisla) govoriti o stabilnosti ovog algoritma.

Korisni linkovi ka sajtovima sa simulacijama raznih algoritama za sortiranje nizova
http://www.ida.liu.se/~TDDB28/mtrl/demo/sorting/index.sv.shtml
http://home.earthlink.net/~mihailod/atic/sorting.html
http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html