УНИВЕРЗИТЕТ "УНИОН-НИКОЛА ТЕСЛА"

ПОСЛОВНИ И ПРАВНИ ФАКУЛТЕТ БЕОГРАД

STRUKTURE PODATAKA I ALGORITMI

- PREDAVANJA -

Predavač: Docent prof. dr Borivoje M. Milošević

2018

Студијски програм: Информационе технологије

Назив предмета: Структуре података и алгоритми

Наставник: Боривоје Милошевић

Статус предмета: ОБ

Број ЕСПБ: 6

Услов:нема

Циљ предмета

Стицање основних знања о структурама података, фундаменталним алгоритмима, анализи и принципима конструкције алгоритама.

Исход предмета

Студент поседује знања о структурама података, принципима конструкције и анализи алгоритама, која је у стању да примени на решавање нових проблема.

Садржај предмета

Теоријска настава

Низови: дифиниција низова, операције са низовима, типови података string. Ланчане листе: дефиниција структуре, типови ланчаних листи - једноструко повезане,двоструко повезане, цикличне, основне операције (обилазак, додавање, брисање), напреднеоперације, статичка и динамичка имплементација ланчаних листи.Ред, Магацин, Дек: дефиниција структуре, статичка и динамичка имплементација реда,магацина и дека, основне операције (обилазак, додавање, брисање) код статичке и динамичкеимплементације. Хеш таблице: дефиниција структуре, дефиниција појмова (хеш функција, колизија исинонима), решавање колизије(отворено адресирање, уланчавање синонима), имплементацијахеш таблице, основне операције (тражење, читање/брисање).Стабла: основни појмови, бинарна и општа стабла, операције (обилазак, додавање и брисање чворова), уређена бинарна стабла, статичка и динамичка имплементација стабла. Увод у конструкцију и анализу алгоритама. Алгоритми сортирања временске сложености O(N log N); сортирање линеарне сложености, доња граница сложености сортирања. Анализа алгоритама: асимптотска анализа најгорег или просечног случаја; асимптотске ознаке O, o, Ω, Θ; временска и просторна сл ожености; израчунавање коначних сума, рекурентне релације, основна теорема. Графови: основни појмови, претрага у дубину, претрага у ширину. Алгоритамске стратегије: алгоритми грубе силе; похлепни (greedy) алгоритми; рекурзивна стратегија заснована на разлагању (divide-and-conquer); претрага (backtracking), гранање са одсецањем (branch-and-bound), хеуристике. Тражење узорка у тексту. Примери нумеричких алгоритама. Имплементација рекурзије. Свођење репне рекурзије на итерацију

Практична настава

Настава се обавља на рачунарима у потпуности прати горе наведене теме а студенти самостално примењују стечена знања на рачунарима.

Литература

1.Strukture podataka u jeziku C++. Praktikum / Slobodanka Đorđević-Kajan, Leonid Stoimenov, Aleksandar Dimitrijević. - Niš : Elektronski fakultet, 2005. (Edicija: Pomoćni udžbenici)

2.Stevan Berber, Miodrag Temerinac "Osnovi algoritama i struktura DSP = Fundamentals of Algorithms and Structures for DSP", ФТН Нови Сад 2013.

3.T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, Cambridge, 2001.

4. Jon Kleinberg, Eva Tardos, Algorithm Design, Pearson International Edition, USA, 2006.

Број часова активне наставе	**Теоријска настава : 2**	**Практична настава: 2**

Методе извођења наставе

Предавања, рачунарске вежбе, лабораторијске вежбе, консултације, семинарски радови домаћи задаци и писмени испит.

Оцена знања (максимални број поена 100)

Предиспитне обавезе	**поена**	Завршниспит	**поена**
активност у току предавања	**10**	писмени испит	**30**
практична настава		усмени испт	**20**
колоквијум-и	**20**	----------	
семинарски	**20**		
Укупно:	**50**		**50**

UVOD

Svaki programer složiće se sa tvrdnjom da pri razvoju računarskog programa moramo voditi brigu o dve stvari:

- o strukturama podataka i
- o algortimima.

Strukture podataka čine "statički" aspekt programa – to je ono sa čime se radi. Algoritmi predstavljaju "dinamički" aspekt – to je ono šta se radi.

Računarski programi mogli bi uporediti sa kuvarskim receptom: kao što recept na svom početku sadrži popis sastojaka (ulje, luk, brašno, meso, . . .) tako i program mora započeti sa definicijama podataka.

Kao što recept mora opisivati postupak pripreme jela (seckanje, kuvanje, mešanje, .), tako i program mora imati izvršne naredbe koje opisuju algoritam.

Rezultat primene kuvarskog recepta je jelo dobijeno od polaznih sastojaka primenom zadanog postupka. Rezultat izvršavanja programa su izlazni podaci dobijeni transformacijom ulaznih podataka primenom zadatog algoritma.

STRUKTURE PODATAKA

Strukture podataka mogu se shvatiti kao elementi nad kojima se grade algoritmi. Zbog toga je njihovo poznavanje (odnosno poznavanje tehnika rada sa njima i poznavanje složenosti operacija sa njima) neophodno za savladavanje tehnika konstrukcije algoritama. Strukture podataka su široko proučavana oblast.

Strukture podataka služe za efikasno upravljanje velikim količinama podataka koje upotrebljavamo kao što su velike baze podataka i usluge indeksiranja interneta.

Veoma često su strukture podataka i ključne za dizajniranje efikasnih algoritama. Neke formalne metode samog dizajna i programski jezici naglašavaju kao najvažniji faktor u dizajnu softvera baš strukture podataka a ne algoritme kako se očekuje.

Koristan pojam koji ćemo u vezi sa strukturama podataka koristiti je *apstraktni tip podataka*. Najvažnija karakteristika apstraktnog tipa podataka je dakle spisak operacija koje se nad njim mogu izvršavati.

U programima se obično tačno specificira tip svakog podatka (celi, realni, logički, i slično).

STRUKTURE PODATAKA

Medjutim, u nekim slučajevima pri konstrukciji algoritama konkretan tip podataka nije bitan. Na primer, ako se radi sa FIFO listom (skraćeno od First In First Out, odnosno prvi unutra prvi napolje) od interesa su operacije upisa na listu i skidanja sa nje.

Elementi se sa liste skidaju istim redom kojim su u nju upisivani; primer FIFO liste je red za karte ispred bioskopa.

Korisno je da se u ovom slučaju ne precizira tip podataka, nego samo operacije sa njima.

STRUKTURE PODATAKA

Drugi primer apstraktnog tipa podataka je lista u kojoj su članovima pridruženi brojevi - prioriteti.

Skidanje elemenata sa liste vrši se redom prema prioritetu članova liste, a ne prema redosledom upisivanja.

Ova struktura podataka zove se lista sa prioritetom. I u ovom slučaju nepotrebno je specificirati tip samih podataka - članova liste; šta više, ni tip podataka prioriteta. Dovoljno je da su prioriteti elementi potpuno uredjenog skupa.

STRUKTURE PODATAKA

Koncentrišući pažnju na prirodu struktura podataka sa aspekta potrebnih operacija, a zanemarujući preciznu realizaciju, dobija se opštiji algoritam.

Tako, na primer, tehnike realizacije liste sa prioritetom malo zavise od konkretno izabranog tipa podataka. Dakle, ako uočimo da nam je potreban neki apstraktni tip podataka, možemo odmah da ga primenimo.

Time konstrukcija algoritama postaje modularnija i sastoji se od medjusobno malo zavisnih celina.

STRUKTURE PODATAKA

- Navedimo četiri osnovne operacije na strukturama podataka koje se primenjuju na svim vrstama struktura:

- **pristup i obrada** preko svih podataka (eng. traversing):
  pristup svakom elementu strukture tačno jedanput, da bi se određeni podatak obradio (procesirao) primer - učitavanje polja ili matrice

- **pretraživanje** (eng. searching):
  pronalaženje lokacije elementa strukture koji sadrži željenu vrednost, ili pronalaženje svih elemenata strukture koji ispunjavaju jedan ili više uslova

- **dodavanje** novog elementa u strukturu

- **brisanje** određenog elementa iz strukture

STRUKTURE PODATAKA

- Strukture podataka se dele na linearne i nelinearne.
- Struktura je linearna ako njeni elementi čine niz, odnosno slede linearnu listu. Dva su osnovna načina prikaza linearnih struktura u memoriji računara:
  - linearna veza između elemenata ostvarena sledom memorijskih lokacija - to je karakteristično za polja
  - linearna veza između elemenata ostvarena pokazivačima - karakteristično za povezane liste
- Nelinearne strukture su razgranata stabla i grafovi.
- Pored već navedenih operacija pristupa i obrade, pretraživanja, dodavanja i brisanja, na linearne strukture se primenjuju i operacije *sortiranja* (eng. sorting) po određenom redosledu i *spajanja* (eng. merging) - kombinovnja dve liste u jednu.
- Izbor određene vrste linearne strukture za datu situaciju zavisi od relativne učestalosti primene pojedine od navedenih operacija.

STRUKTURE PODATAKA

- Sledeće strukture podataka i operacije nad njima temelji su računarske nauke jer su osnovni elementi brojnih algoritama:

Ćelija (cell)	Promenljiva koju posmatramo kao zasebnu celinu.
Polje (array)	Sekvencijalni niz podataka istog tipa koje imaju zajedničko ime.
Slog (record)	Skup podataka koji mogu biti različitog tipa, niz slogova obično je deo datoteke ili tabele.
Lista (linked list)	Niz elemenata koji sadrže podatke i pokazivače na sledeći element.
Stek (stack)	Niz elemenata u kojem se dodavanje i brisanje mogu obavljati samo na jednom kraju niza.
Red (queue)	Niz elemenata u kojem je dodavanje moguće samo na jednom kraju a brisanje samo na drugom kraju.
Stablo (binary tree)	Hijerarhijska struktura u kojoj svaki element može imati samo jednog prethodnika.
Graf (graph)	Opšta struktura u kojoj svaki element može biti povezan sa više drugih elemenata.

Elementarne strukture podataka

- Pod elementom podrazumevamo podatak kome tip nije preciziran (na primer celi broj, skup celih brojeva, fajl i slično).

- Tako, na primer, kad se radi o sortiranju (uredjivanju elemenata linearno uredjenog skupa po veličini), ako su jedine operacije uporedjivanje i kopiranje, onda se isti algoritam može primeniti i na cele brojeve i na imena - razlika je samo u realizaciji, a ideje na kojima se zasniva algoritam su iste.

Ćelija - Cell

- Ćelija (eng. *Cell*) je varijabla koju posmatramo kao zasebnu, pretežno nedeljivu celinu.
- Svaka ćelija ima svoj tip, adresu (ime) i sadržaj (vrednost). Tip i adresa su nepromenljivi a sadržaj se može menjati u toku programa. Gradivni je element mnogih drugih struktura, poput polja.
- Predstavlja relativan pojam jer možemo analizirati i njenu unutrašnju građu u određenim okolnostima.
- Svaka ćelija ima tip i adresu.
- Sadržaj ćelija je odgovarajućeg tipa.
- Predstavlja gradivni element polja.

Memorija je kao velika tabela numerisanih mesta, gde bitovi podataka mogu biti zapamćeni.

Broj mesta je njegova Adresa.
Samo jedan byte vrednosti može biti zapamćen na jednom mestu.

Neke vrednosti “logical” podataka zahtevaju više od jednog mesta, kao npr. karakter string “Hello\n”

Tip označava logičku veličinu podatka za ssmeštaj u memoriji. Neki jednostavni tipovi podataka su:

```
char
char [10]
int
float
int64_t
```

a single character (1 slot)
an array of 10 characters
signed 4 byte integer
4 byte floating point
signed 8 byte integer

Addr	Value
0	
1	
2	
3	
4	‘H’ (72)
5	‘e’ (101)
6	‘l’ (108)
7	‘l’ (108)
8	‘o’ (111)
9	‘\n’ (10)
10	‘\0’ (0)
11	
12	

Šta je promenljiva?

Promenljiva imenuje mesto u memoriji gde se pamti Vrednost za nju određenog Tipa.

Prvo moramo da definišemo promenljivu dajući joj ime i specificirajući njen tip, a opciono i njenu inicijalnu vrednost.

declare vs define?

```
char x;
char y='e';
```

Inicfijalna vrednost x nije definisana

Inicijalna vrednost

Ime

Koja imena su legalna?

Tip je jedan karakter (char)

extern? static? const?

symbol table?

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	'e' (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

Kompajler je postavlja negde u memoriji.

Vektor - niz: $A_i(a_1,a_2,a_3,\ldots,a_N)$

- Vektor (niz) je niz elemenata istog tipa. Veličina vektora je broj elemenata u njemu. Veličina vektora se mora unapred zadati (odnosno fiksirana je), jer se unapred zna veličina dodeljene memorije.
- Na primer, vektor od 100 elemenata - imena, od kojih se svako zapisuje u 8 bajtova, zauzima u memoriji 800 bajtova. Ako je prvi elemenat na lokaciji *x*, a veličina elementa je *a* bajtova, onda je *k*-ti elemenat vektora na lokaciji $x + (k - 1)a$, što se lako izračunava - nezavisno od rednog broja elementa.
- Ovo je efikasna i zbog toga često korišćena struktura podataka. Njena osnovna osobina je da je jednako vreme pristupa svim elementima.
- Na višem programskom jeziku se o poziciji elementa i ne razmišlja, jer taj deo posla obava prevodilac (kompajler).
- Zakučak je da vektor treba koristiti kad god je to moguće. Nedostaci ove strukture podataka su da su svi njeni elementi istog tipa i da se njena veličina ne može menjati dinamički, u toku izvršavanja algoritma.

Vektor - niz : $A_i(a_1,a_2,a_3,\ldots,a_N)$

Nizovi

Niz ili vektor je jednodimenzionalno polje

To je skup podataka istog tipa

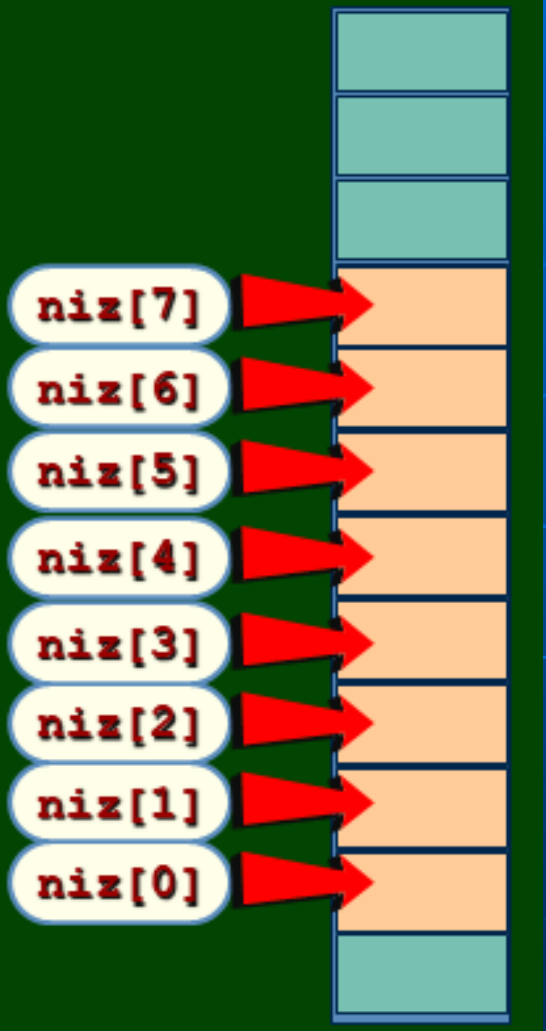
U memoriji se smešta u niz uzastopnih mem. lokacija

Svi podaci u nizu imaju jedno ime = IME NIZA

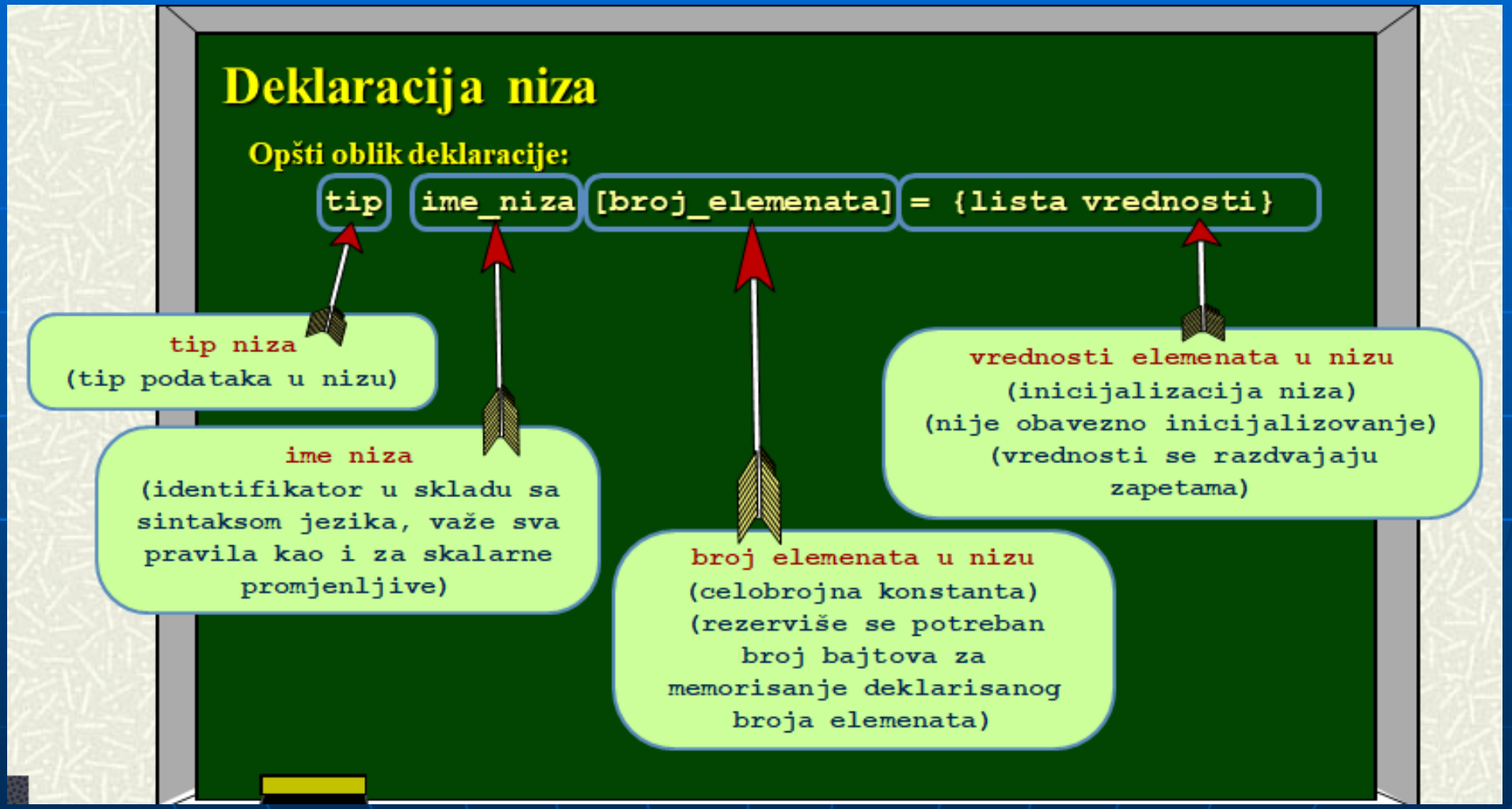
Ime niza predstavlja početnu adresu niza u memoriji

Svaki element u nizu određen je:

- ✓ imenom niza, i
- ✓ indeksom (pomerajem u odnosu na početak niza)



Vektor - niz : $A_i(a_1,a_2,a_3,...,a_N)$



Vektor - niz : $A_i(a_1,a_2,a_3,...,a_N)$

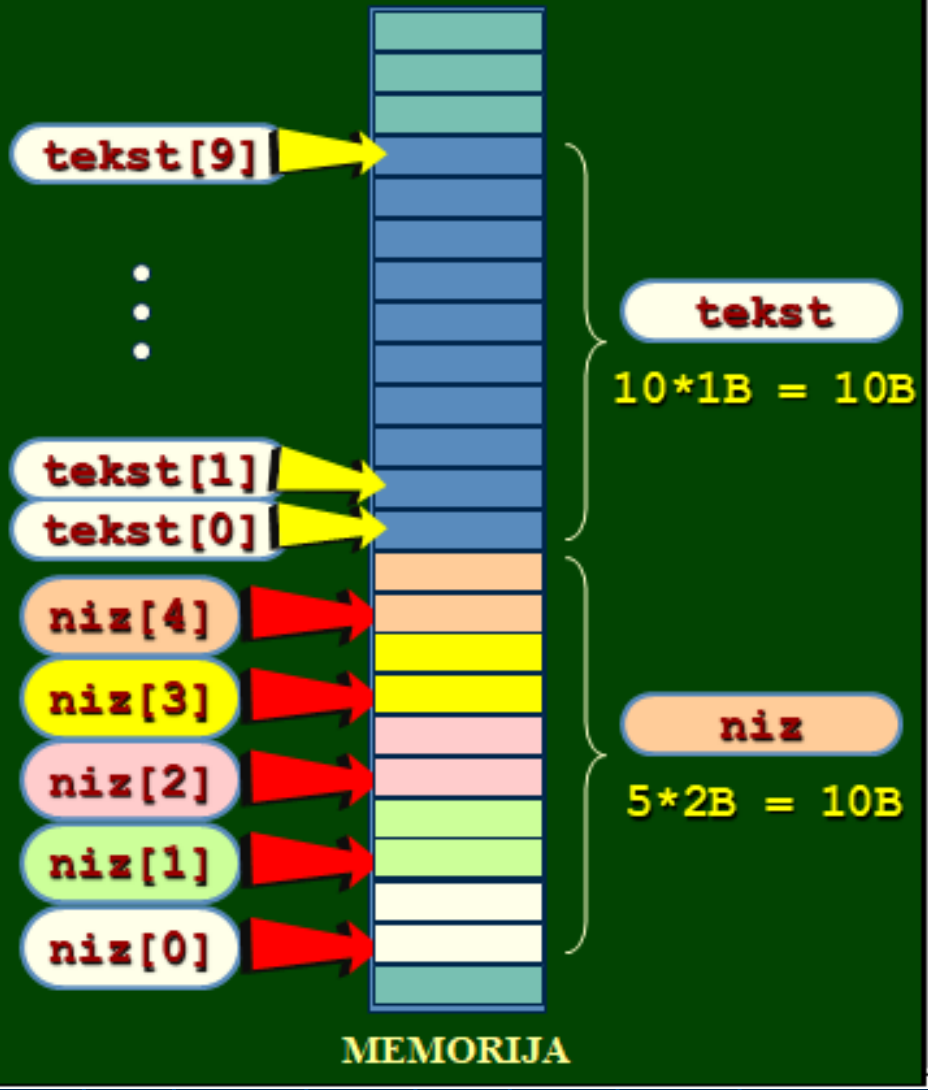
Deklaracija niza

Primer deklaracije:

```
int niz[5];
char tekst[10];
```

Primer:

```
#define MAX 5
#define LENG 10
main()
{
   int niz[MAX];
   char tekst[LENG];
   ...
}
```



Vektor - niz : $A_i(a_1,a_2,a_3,...,a_N)$

Deklaracija niza

Primer deklaracije sa inicijalizacijom:

```
int A[3]={1,2,3};
```

A[0]	A[1]	A[2]
1	2	3

```
int B[]={1,2,3,4};
```

B[0]	B[1]	B[2]	B[3]
1	2	3	4

```
int C[5]={10};
```

C[0]	C[1]	C[2]	C[3]	C[4]
10	0	0	0	0

```
int D[5]={0};
```

D[0]	D[1]	D[2]	D[3]	D[4]
0	0	0	0	0

Multidimenzionalni nizovi

- Multidimenzionalni nizovi su nizovi većih dimenzija, što znači da imamo nizove čiji su elementi nizovi, koji opet kao elemente mogu imati nizove. Multidimenzionalni nizovi deklarišu se na sledeći način:
- **Tip naziv[velicina1] [velicina2] [velicina3]..[velicinaN];**

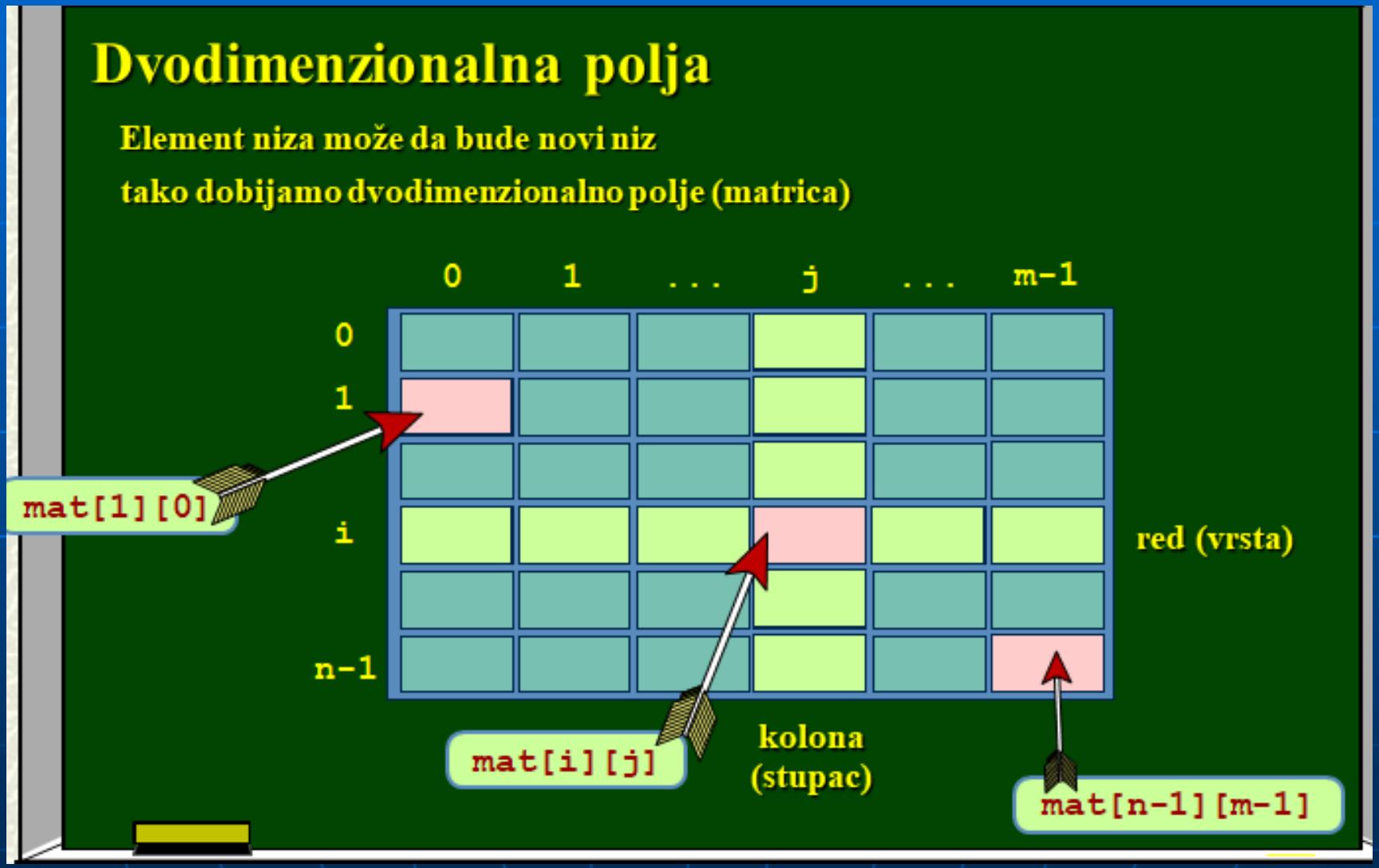
Matrice: Od multidimenzionalnih nizova najčešće se koristi dvodimenzionalni niz koji se još naziva i matrica, a to je niz čiji su elementi jednodimenzionalni nizovi i deklariše se na sledeći način:

Tip dvodimenzionalniNiz [m][n];

Ovakav dvodimenzionalni niz se može predstaviti i kao tabela koja ima m vrsta i n kolona. Ako bismo imali matricu a[3][4], njene vrednosti bi se tabelarno mogle predstaviti na sledeći način:

	Kolona 0	Kolona 1	Kolona 2	Kolona 3
Vrsta 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Vrsta 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Vrsta 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Multidimenzionalni nizovi

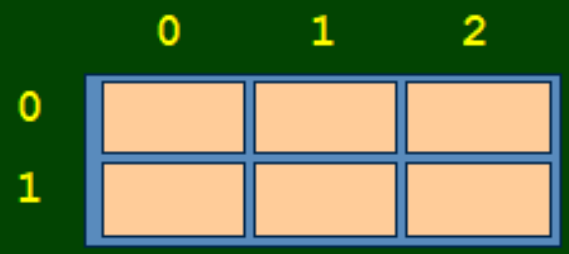


Multidimenzionalni nizovi

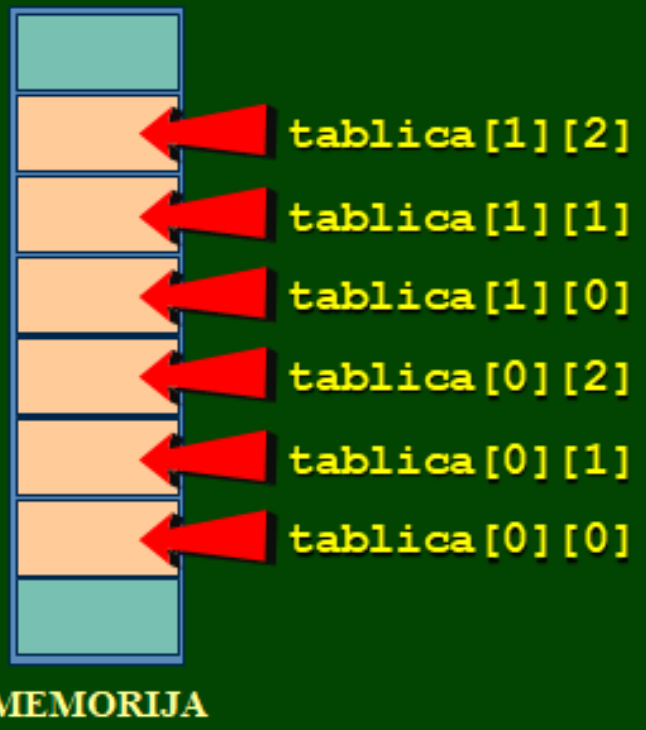
Deklaracija matrice

Primer deklaracije:

```
char tablica[2][3];
```



logička predstava

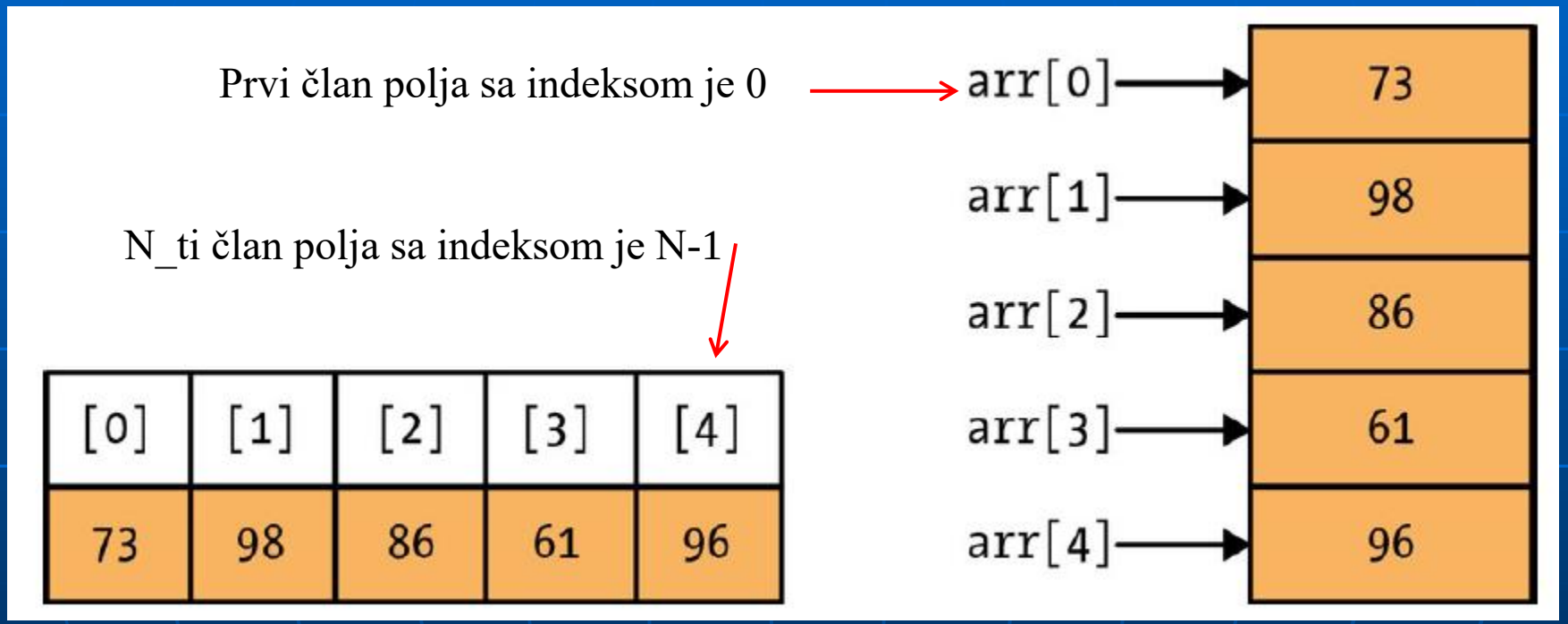


fizička predstava

Polje (array)

- Više ćelija istog tipa memorisani su na uzastopnim adresama.
- Broj ćelija polja je unaprijed zadat i nepromenljiv. Broj ćelija je unapred zadat brojem koji predstavlja veličinu niza i ta veličina je nepromenljiva.
- Na poljima se jednostavno primenjuju operacije prolaza, pretraživanja i sortiranja, stoga se polja koriste za memorisanje relativno permanentnih (statičnih) skupova podataka.
- Sa druge strane, u situacijama gde se veličina strukture ili sami podaci konstantno menjaju, povezane liste su pogodnije strukture od polja.
- Polje zauzima sekvencijalni niz memorijskih lokacija, a elementima polja pristupa se preko indeksa. Indeksi celobrojne konstante.
- Dakle, fizički redosled memorijskih lokacija određuje povezanost elemenata polja.

Polje (array)


Prvi član polja sa indeksom je 0
N_ti član polja sa indeksom je N-1
[0] [1] [2] [3] [4]
73 98 86 61 96
arr[0]
arr[1]
arr[2]
arr[3]
arr[4]
73
98
86
61
96

Polje (array)

- Osnovni nedostaci polja su sledeći:
  - teško je dodavati i brisati elemente nakon što se elementima pridruže vrednosti
  - u većini jezika problematično je povećati alokaciju memorije za već određene elemente polja
- Zbog navedenih nedostataka polja se smatraju statičkim strukturama podataka.
- Algoritam "pristupa i obrade" (eng. traversing) svih elemenata polja vrlo često se koristi naprimer kod učitavanja ili ispisa elemenata polja ili kao deo drugih algoritama:

1. postavi brojač na početnu vrednost jednaku indeksu početnog elementa polja
2. ponavljaj korake 3 i 4 tako dugo dok je brojač manji od indeksa krajnjeg elementa polja
3. pristupi elementu polja i primjeni željenu obradu na njemu
4. povećaj vrednost brojača za 1

Polje (array)

Višedimenzionalna polja

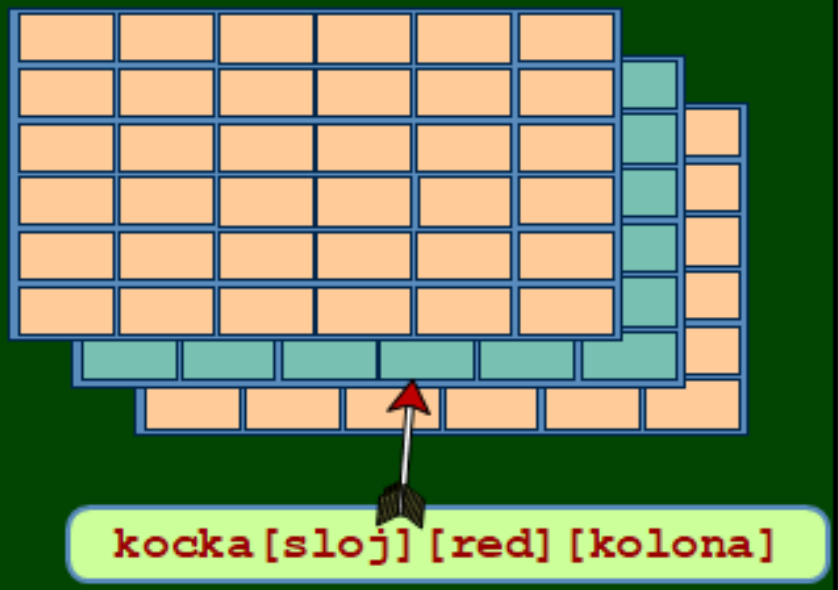
C omogućava i manipulaciju višedimenzionalnim poljima

Trodimenzionalno polje

```
tip ime[d1][d2][d3];
```

Primer deklaracije:

```
int kocka[3][6][6];
```

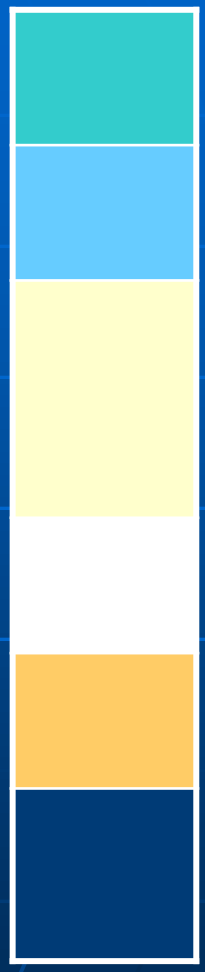


Primer 4-dim. polja:

```
char P4[d1][d2][d3][d4];
```

Slog - zapis (Record)

- Slog je skup međusobno povezanih imenovanih (ali heterogenih) podataka koje obično nazivamo polja ili atributi. Zapisi nekih vrsta datoteka su zapravo skupovi slogova, odnosno takve datoteke su linearne liste slogova. Važno je naglasiti osnovne razlike između sloga i polja:
  - elementi sloga mogu biti nehomogeni podaci (različitih tipova) ali su memorisani na uzastopnim adresama
  - ne mora postojati "prirodni" redosled elemenata sloga, elementi se označavaju imenima atributa
- Skup slogova možemo posmatrati i kao tabelu u kojoj svaki red ima svoje ime i odgovara pojedinom polju (atributu) sloga.
- Takva struktura čini osnovu tzv. relacionih baza podataka. Pojedini atribut sloga mora imati isti tip u svim slogovima.



Slog - zapis (Record)

- Slog (rekord) je takodje niz elemenata, koji medjutim ne moraju biti istog tipa - jedino se fiksira kombinacija tipova elemenata.
- Veličina sloga takodje mora unapred biti definisana. Elementima sloga može se kao i elementima vektora pristupiti za konstantno vreme: za ovu svrhu koristi se poseban vektor sa početnim adresama elemenata sloga, dužine jednake broju elemenata sloga.
- Tako, na primer, ako je slog definisan kao na slici, onda je adresa elementa *Int*6 jednaka 2*4+3*20*4+3*4+1 = 261.
- Adrese ovog i ostalih elemenata prevodilac izračunava samo jednom, prilikom prolaska kroz definiciju sloga, i smešta ih u spomenuti pomoćni vektor.

```
primer = record
    Int1 : integer;
    Int2 : integer;
    Ar1 : array[1..20] of integer;
    Ar2 : array[1..20] of integer;
    Ar3 : array[1..20] of integer;
    Int3 : integer;
    Int4 : integer;
    Int5 : integer;
    Int6 : integer;
    Ime1 : array[1..12] of character;
    Ime2 : array[1..12] of character
end
```

Slog - zapis (record)

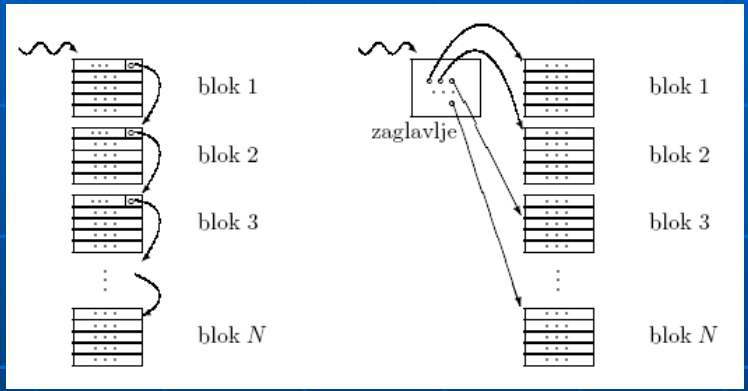
- Primer sloga sa podacima o studentima - prvi red tabele sadrži nazive atributa u slogu, a prikazan je niz od šest slogova:

IME	PREZIME	MATIČNI BROJ	DATUM ROĐENJA	UPISANA GODINA
Ime 1	Prezime 1	0038512345	05.04.1984.	1
Ime 4	Prezime 4	0038512766	12.08.1984.	1
Ime 8	Prezime 8	0038576238	30.01.1982.	3
Ime 2	Prezime 2	0038636686	25.10.1983.	2
Ime 3	Prezime 3	0038578798	05.04.1984.	2
Ime 5	Prezime 5	0037987987	09.07.1980.	4

tip char tip char tip date tip int

Sekvencijalna datoteka

- **Jednostavna – sekvencijalna datoteka** Zapravo se radi o odsustvu bilo kakve organizacije. Zapise datotekeređamo u onoliko blokova koliko je potrebno. Blokovi koji čine datoteku mogu biti povezani u vezanu listu (svaki blok sadrži adresu idućeg bloka) ili može postojati tablica adresa svih blokova.



- Traženje zapisa sa zadanom vrednošću ključa zahteva sekvencijalno čitanje cele datoteke (ili bar pola datoteke u poretku), sve dok ne nađemo na traženi zapis. Ako je datoteka velika, moramo učitati mnogo blokova, pa pristup traje dugo. Da bi ubacili novi zapis, stavljamo ga na prvo slobodno mesto u prvom nepopunjenom bloku, ili priključujemo novi blok ukoliko su svi postojeći blokovi popunjeni.

Pokazivač (Pointer)

- Ćelija koja pokazuje na neku drugu ćeliju
- Služi za uspostavljanje veze između delova strukture
- Sadržaj pokazivača je adresa ćelije na koju treba ukazati
- Pokazivači (engl. Pointer) su ćelije koje služe za uspostavljanje veze između delova strukture tako što pokazuju gde se nalazi neka druga ćelija. Pokazivači predstavljaju tip podataka čije su vrednosti memorijske adrese.

Dakle, sam sadržaj pokazivača je adresa ćelije na koju treba ukazati. Ova struktura nam omogućava da steknemo uvid u to kako se promenljive smeštaju u memoriji računara ali i da možemo, ukoliko neravnomerno rasporedimo podatke u memoriji, u svakom trenutku znati gde se nalazi sledeći element koji nam je potreban. Posebno je značajan u radu sa složenijim strukturama podataka kao što su liste.

Liste

- Često je potrebno da se broj elemenata dinamički menja. Prilikom rada sa velikim brojem elemenata na primer u radu sa nizovima ili ukoliko ne znamo unapred koliko nam je elemenata potrebno, neefikasno je rezervisati memorijski prostor prema najgorem slučaju.
- Pored toga, često je potrebno umetanje ili brisanje nekog elementa iz sredine liste, što je kod rada sa nizovima takođe neefikasno. Zbog ovih razloga se koriste dinamičke strukture podataka, a lista (engl. *Linked list*) je najjednostavnija od njih. Elementi su međusobno povezani pomoću pokazivača.
- Jedan element, koji se često naziva čvorom liste, čini vrednost broja, karaktera ili drugog elementa sa kojim radimo i pokazivač na sledeći element. Poslednji element u listi pokazuje na NULL vrednost.
- Postoje i dvostruko povezane liste u kojima element uključuje i pokazivač koji sadrži i adresu prethodnog elementa, pa je kretanje kroz listu u tom slučaju dodatno olakšavajuće. Kod dvostruko ulančanih listi pokazivač prethodnik prvog elementa takođe pokazuje na NULL.

Povezana lista (linked list)

- Često je potrebno da se broj elemenata dinamički menja.
- Kad se unapred ne zna broj elemenata vektora, moguće je rezervisati dovoljno veliki vektor i tako rešiti problem - što je često dobro rešenje.

- Ipak, neefikasno je rezervisati memorijski prostor prema najgorem slučaju. Pored toga, često je potrebno umetanje elemenata ili brisanje elemenata iz sredine liste - što je neefikasno kod vektora. Zbog navedenih razloga koriste se tzv. dinamičke strukture podataka. Najjednostavnija od ih je povezana lista.

- Ako je cilj efikasno umetanje i izbacivnjae elemenata, mora se napustiti sekvencijalni zapis koji karakteriše vektor - elementi se moraju
- zapisivati nezavisno, a medjusobno se povezuju (nadovezuju) pomoću pokazivača, promenljive koja sadrži adresu nekog drugog elementa.

Povezana lista (linked list)

Povezana lista je linearna struktura elemenata u kojoj je redosled odredjen pokazivačima. Povezana lista je vektor parova (elemenat, pokazivač), pri čemu pokazivač sadrži adresu narednog para.

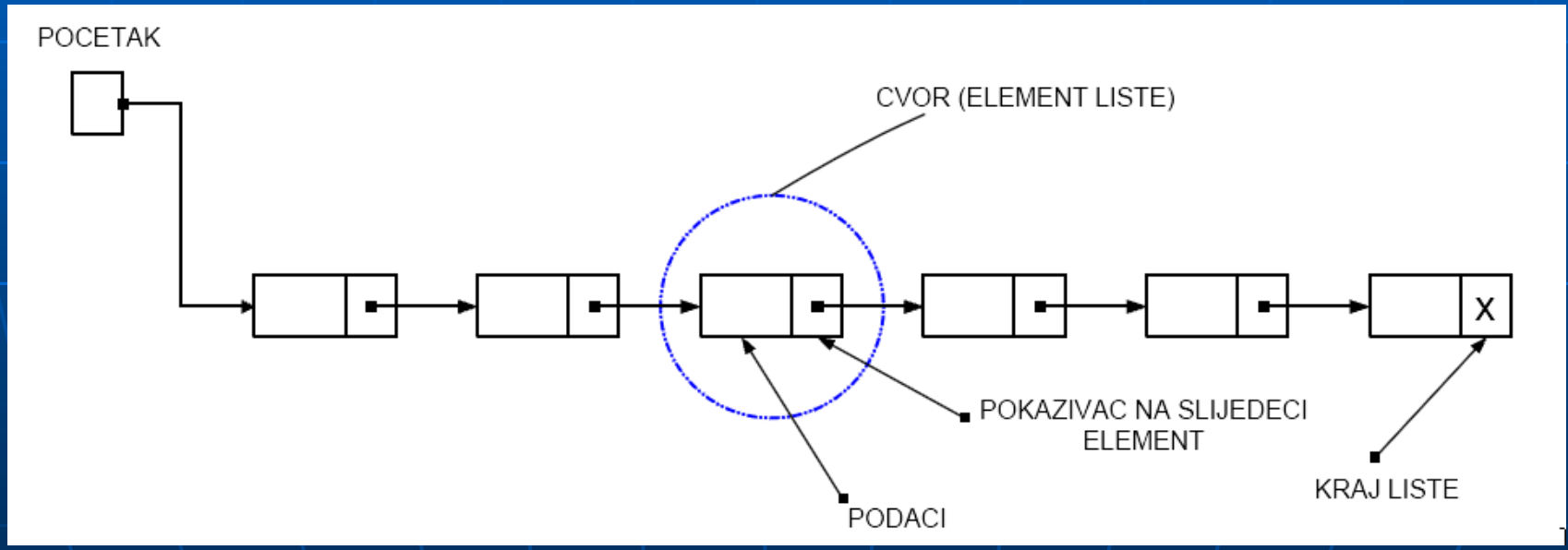
- Za razliku od polja, svaki element liste sadrži pokazivač na sledeći element.
- Pošto pokazivač sadrži adresu sledećeg elementa u listi, sukcesivni elementi liste ne moraju biti spremljeni u sukcesivnom nizu memorijskih lokacija.
- Na taj način fizički redosled memorijskih lokacija nema nikakav uticaj na redosled elemenata u listi.
- Za razliku od polja, vrlo jednostavno je dodavanje i brisanje elemenata iz liste. Svaki element (čvor) povezane liste sastoji se od dva dela:

1. Podatak
2. Pokazivač koji sadrži adresu sledećeg čvora u listi

Povezana lista (linked list)

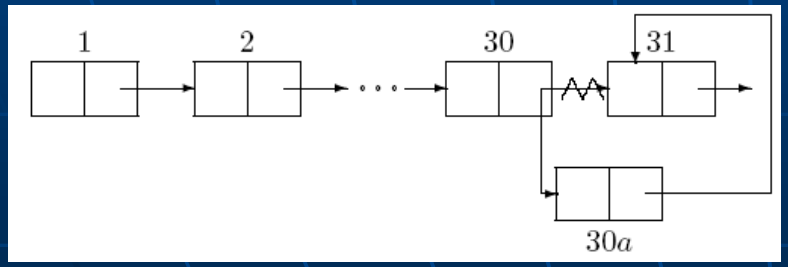
- Zadnji čvor sadrži tzv. ''null'' pokazivač koji označava kraj liste.
- Lista sadrži posebni pokazivač koji sadrži adresu prvog čvora u listi.

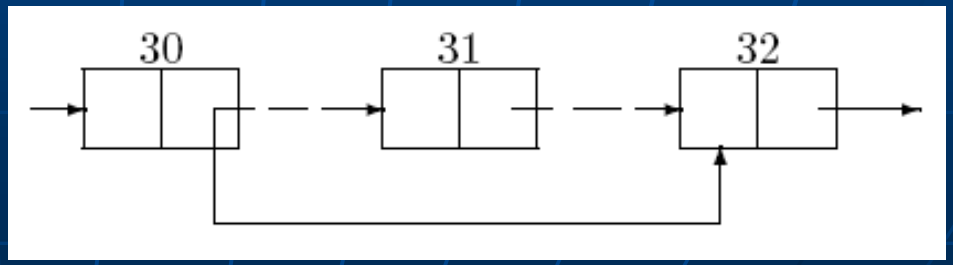
Primer povezane liste sa 6 čvorova:



Povezana lista (linked list)

- Nedostaci ove strukture podataka su u tome što zahteva više prostora u memoriji (uz svaki elemenat ide i pokazivač), a *k*-tom ($k > 1$) elementu može se pristupiti samo preko svih prethodnih.
- Prednost povezane liste je u tome što se upis i brisanje lako realizuju, za razliku od vektora, kod koga ove operacije zahtevaju veliki broj pomeranja elemenata.
- Za umetanje novog elementa izmedju neka dva elementa povezane liste dovoljno je samo promeniti dva pokazivača. Na slici levo je prikazano umetanje novog, elementa 30*a*, izmedju 30. i 31. elementa povezane liste.
- Slično, za brisanje elementa iz liste dovono je promeniti jedan pokazivač. Na slici desno je prikazano brisanje 31. elementa povezane liste.

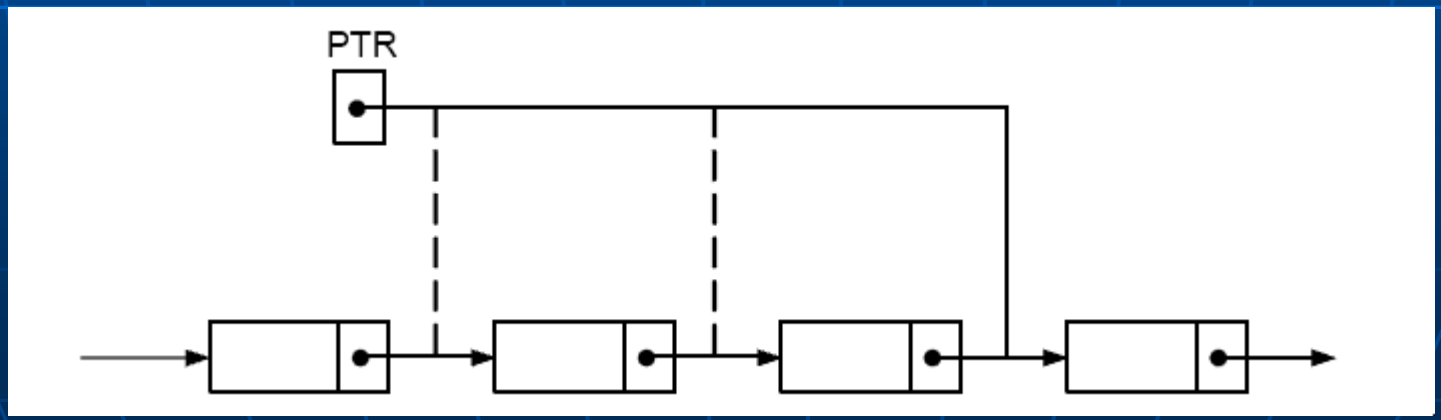




Povezana lista (linked list)

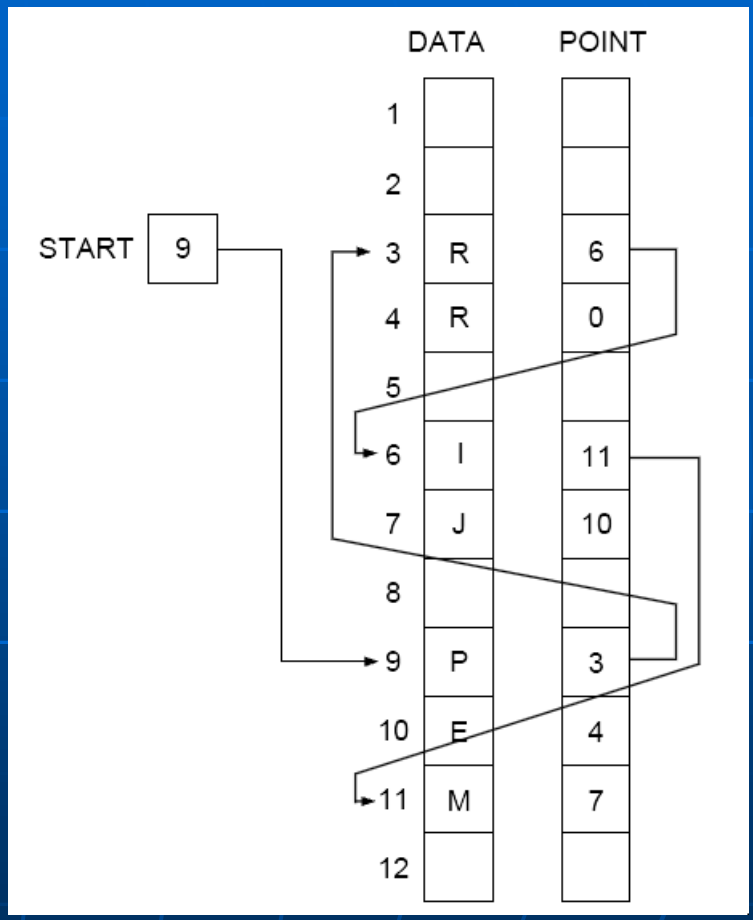
- Algoritam operacije ”pristupa i obrade” svih elemenata povezane liste (eng. traversing a linked list):

1. Postavi pokazivač (PTR) na početak
2. Ponavljaj korake 3 i 4 dok je PTR različit od ”null”
3. Pristupi elementu liste i izvrši željenu obradu
4. Postavi pokazivač (PTR) na sledeći element



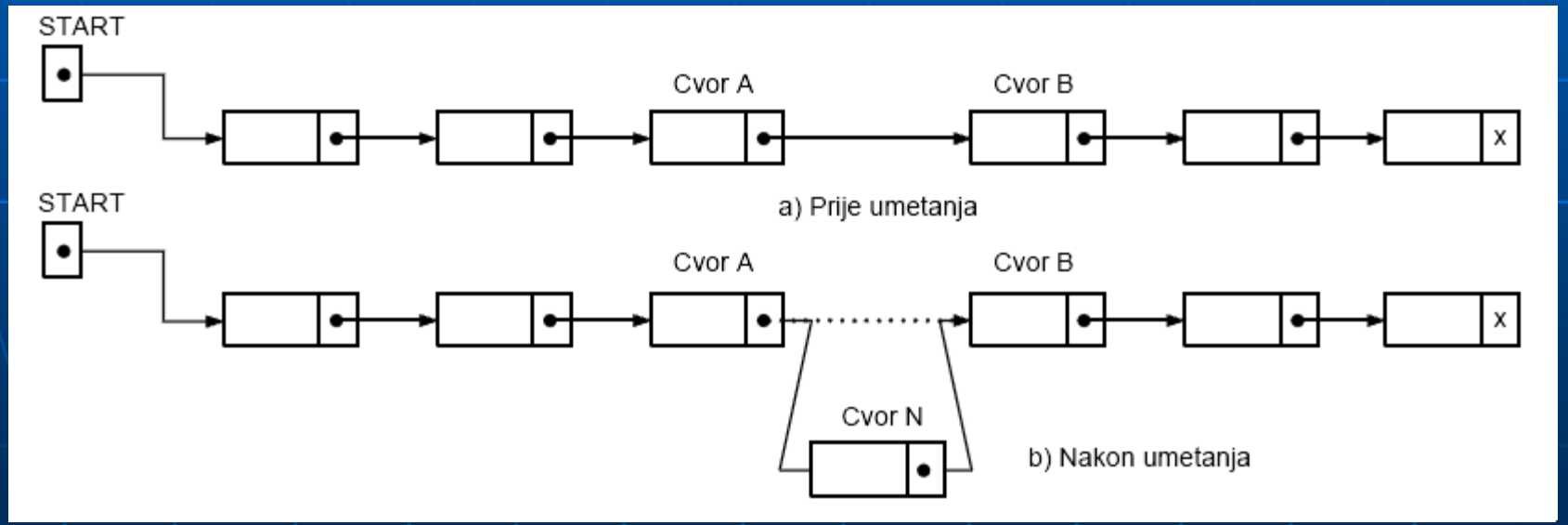
Povezana lista (linked list)

Najčešći način prikaza povezane liste u memoriji je korišćenjem dva paralelna polja i promenljivom (pokazivačem) koja sadrži adresu prvog člana liste.



Povezana lista (linked list)

- Na slici je šematski prikazano dodavanje elementa u povezanu listu.
- Pre dodavanja novog elementa čvor A pokazuje na sledeći čvor B.
- Posle umetanja novog čvora između čvorova A i B, čvor A pokazuje na novi čvor N, a čvor N pokazuje na čvor B.



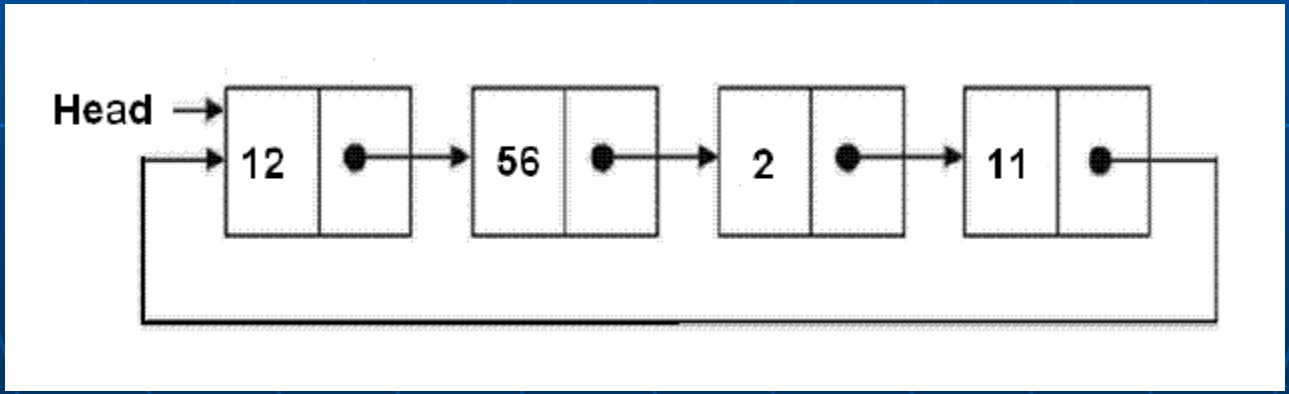
Povezana lista (linked list)

- Lista je konačan niz (od nule ili više podataka istog tipa. Podaci koji čine listu nazivaju se elementima liste.

  - ✓ $[a_1,a_2,a_3,\ldots a_n]$
  - ✓ n - dužina liste
  - ✓ Ako je n=0 lista je prazna
  - ✓ Definisan je prethodnik i sledbenik u listi
  - ✓ Broj elemenata nije fiksan
  - ✓ Identitet elementa liste određen je njegovim pozicijom

Kružne (ciklične, cirkularne) liste.

- Takođe, u strukturama podataka koriste se i kružne (ciklične, cirkularne) liste.
- U jednoj takvoj listi, poslednji element liste ukazuje na prvi element liste. To omogućava da se do bilo kog elementa može doći počevši od bilo kog drugog elementa.



Primer liste

Polinom

$P(x)=a^n x^{en}+ a^n x^{en}+\dots a^n x^{en}$

Gde je $0<e_1<e_2\dots<e_n$

Zapravo se radi o listi

[(a1 ,e1),(a2 ,e2), … ,(an ,en)]

Operacije nad listama

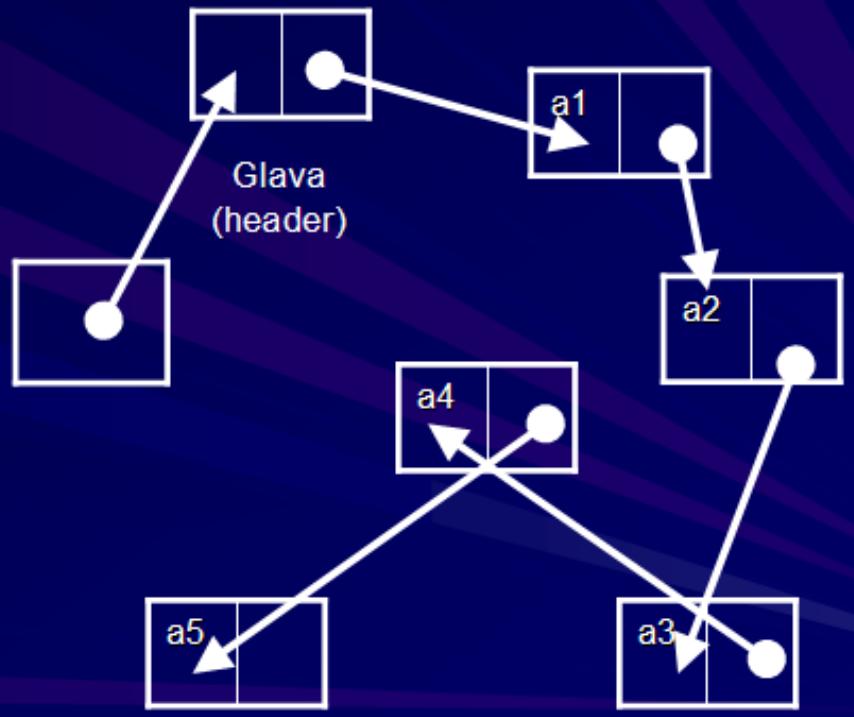
- **END(L)** – funkcija koja vraća poziciju na kraj liste
- **MAKE_NULL(L)** – pretvara listu u praznu listu i vraća poziciju END(L)
- **INSERT(x,p,L)** – ubacuje podatak x na poziciju p u listi L
- **DELETE(p,L)** – izbacuje element p iz liste L
- **FIRST(L)** – funkcija vraća prvu poziciju u listi, za praznu listi vraća END(L)
- **NEXT(p,L)**, PREVIOUS(p,L) vraća poziciju ispred odnosno iza u listi
- **RETRIVE**(p,L)

Implementacija listi

- Pomoću polja

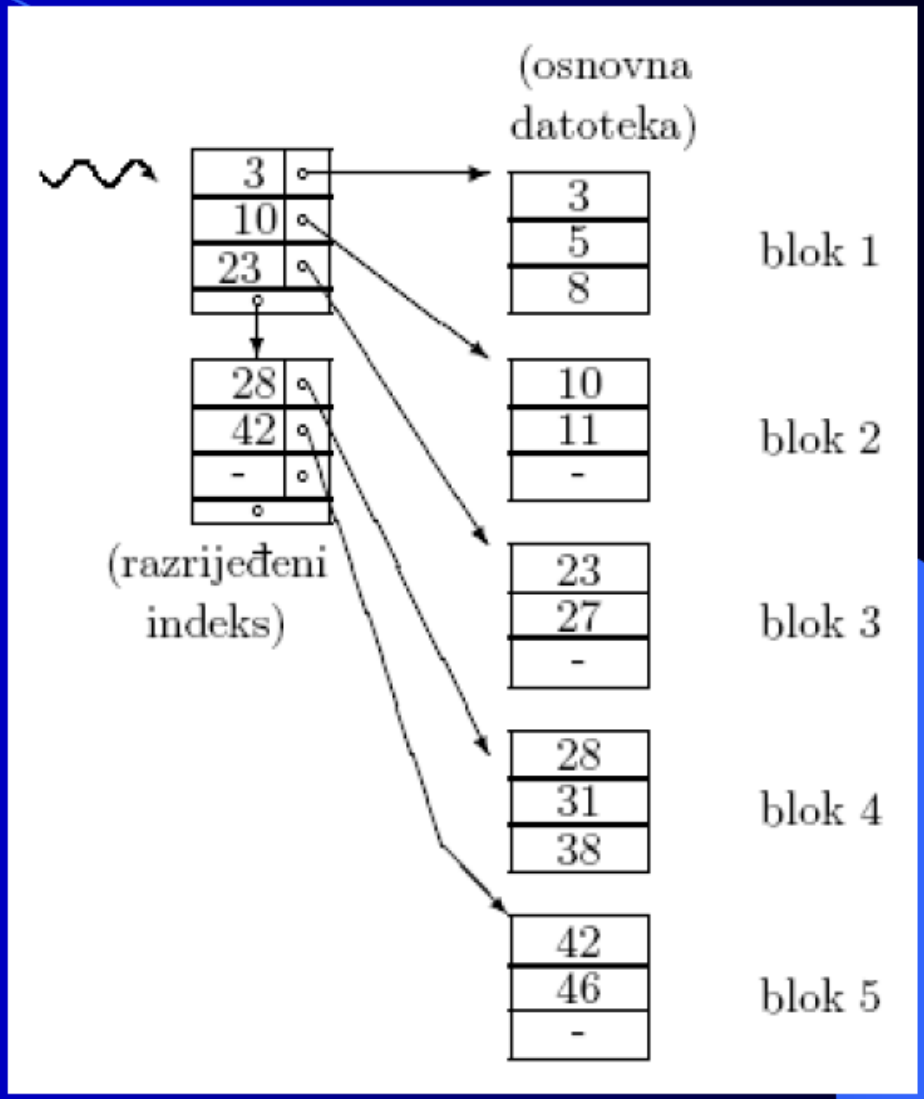
1	a_1
2	a_2
3	a_3
4	a_4
5	a_5
6	a_6

- Pomoću pokazivača



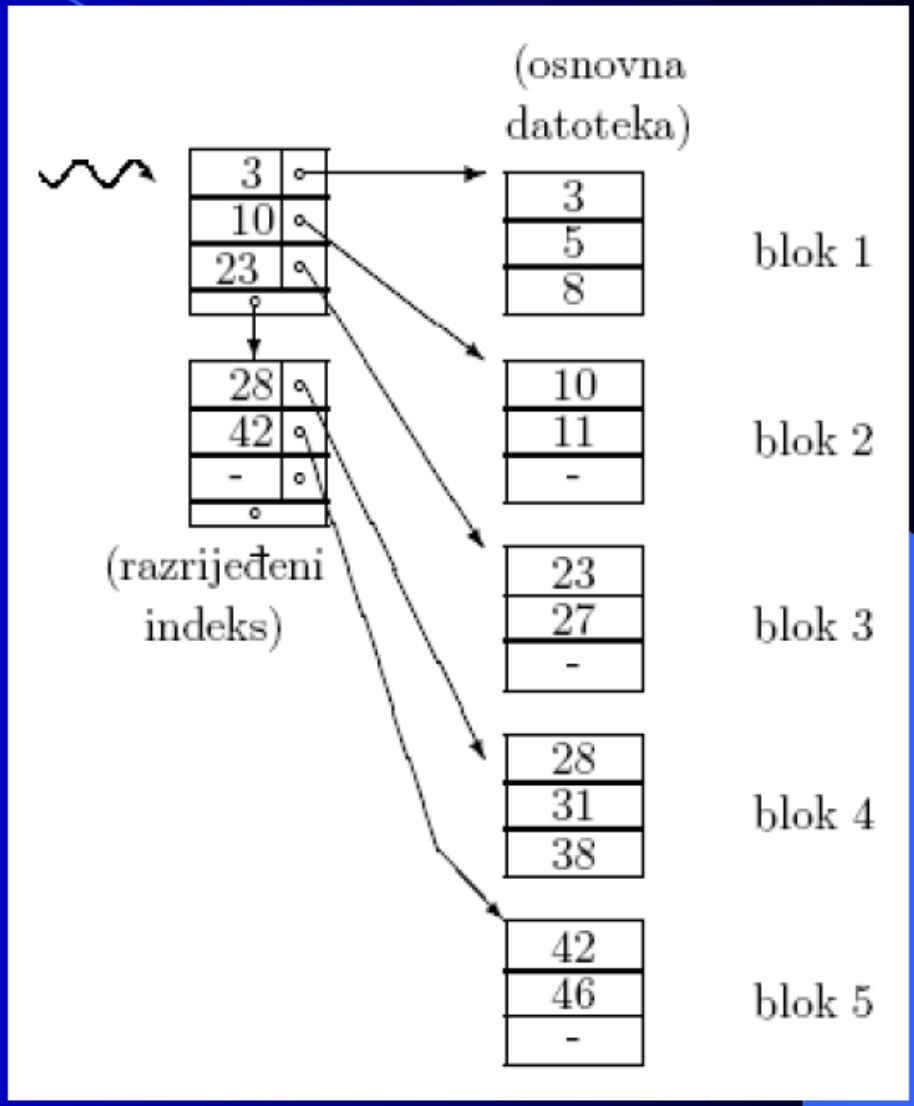
Datoteka sa indeksom

- Indeks je mala pomoćna datoteka koja olakšava traženje u velikoj (osnovnoj) datoteci. Izložićemo dve varijante datoteke sa indeksom.
  - **Indeks sekvencijalna organizacija** zahteva da zapisi u osnovnoj datoteci budu sortirani po vrednostima ključa (na primer rastući). Blokovi ne moraju biti sasvim popunjeni. Dodajemo tzv. razređeni indeks. Svaki zapis u indeksu odgovara jednom bloku osnovne datoteke i oblika je (k, a), gde je k najmanja vrednost ključa u dotičnom bloku, dok je a adresa bloka.

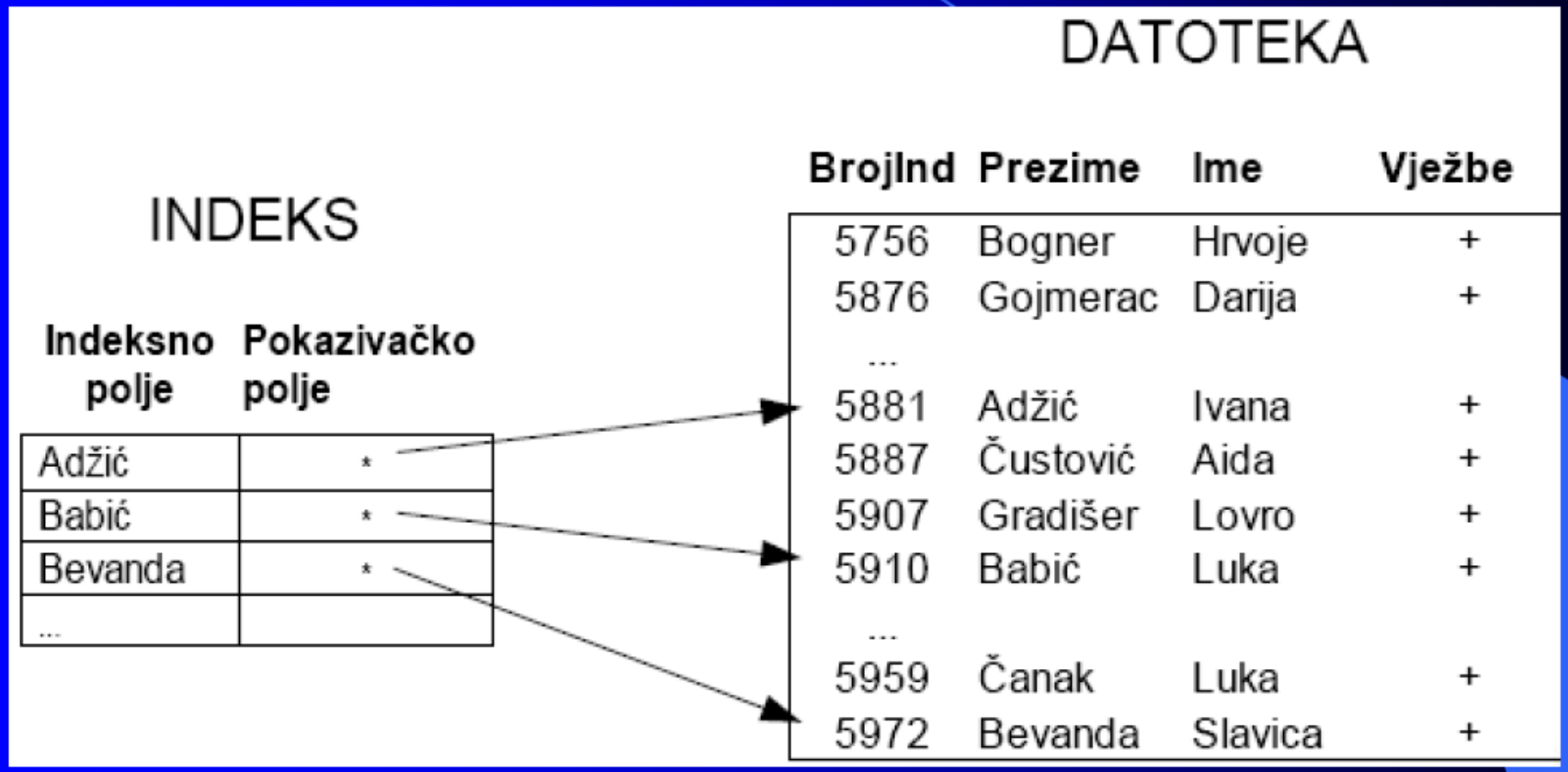


Datoteka sa indeksom - indeks sekvencijalna organizacija

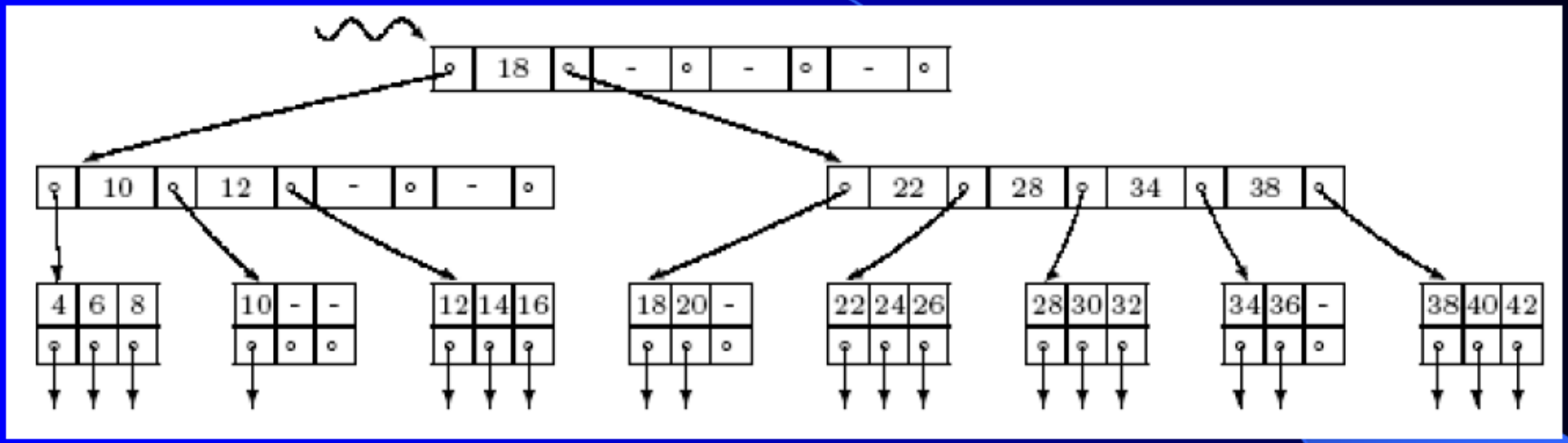
- Da bi u osnovnoj datoteci našli zapis sa zadanom vrednošću ključa k_0, čitamo indeks i tražimo najveći k_1 takav da je $k_1 \leq k_0$ i pritom par (k_1, a_1) postoji u indeksu. Zatim učitamo i pretražimo blok sa adresom a_1. Pristup po primarnom ključu zahteva (u najgorem slučaju) onoliko čitanja bloka koliko ima blokova u indeksu +1.



Datoteka sa indeksom - indeks direktna organizacija



Datoteka sa indeksom B-stablo



- **Traženje.** Želimo u B-stablu pronaći par (k, a), gde je k zadana vrednost ključa, a je pointer adresa traženog zapisa u osnovnoj datoteci. U tu svrhu sledimo put od korena do lista koji bi morao sadržavati k. To se radi tako da redom čitamo unutrašnje čvorove oblika (a_0, k_1, a_1, k_2, a2, . . . ,k_r, a_r), i uporedimo k sa k_1, k_2, . .., k_r. Ako je ki ≤ k < ki+1, dalje čitamo čvor na koga ukazuje ai. Ako je k < k1, dalje čitamo čvor sa adresom a_0. Ako je k ≥ k_r, koristimo adresu a_r. Kad nas taj postupak konačno dovede u list, tražimo u njemu zadani k.

Invertovana datoteka

- Uvodimo **sekundarni indeks** - to je mala (pomoćna) datoteka koja olakšava pristup zapisima velike (osnovne) datoteke na osnovu podatka koji nije ključ. Sekundarni indeks za podatak A sastoji se od zapisa oblika

  (v, {p1, p2, p3, . . .}), gde je

- v vrednost za A,
- a {p1, p2, p3, . . .} je skup pointera na zapise u osnovnoj datoteci u kojima A ima vrednost v.

Ukoliko postoji ovakav indeks, kaže se da je osnovna datoteka invertovana po podatku A. Datoteka koja je invertovana po svakom od svojih podataka zove se potpuno invertovana datoteka.

Invertovana datoteka

- Kao primer, posmatramo tabelarni prikaz datoteke nastavnika na fakultetu. Ako invertujemo tu datoteku po svakom podatku osim IME, dobijamo gusti primarni indeks za ključ MAT_BR i tri sekundarna indeksa za ODELJ, STAROST i ZVANJE. Indeks za STAROST sadrži intervale umesto pojedinačnih vrednosti.

pointer	*MAT_BR*	*IME*	*ODELJ*	*STAROST*	*ZVANJE*
a1	12453	Matić, R.	Matematika	35	Dr.sc.
a2	16752	Pavić, D.	Matematika	26	Dipl.inž.
a3	27453	Milošević, B.	Računarstvo	37	Dr.sc.
a4	34276	Dimić, J.	Fizika	55	Dr.sc.
a5	37564	Katić, K.	Računarstvo	45	Dipl.inž.
a6	43257	Radić, M.	Matematika	32	Mr.sc.
a7	45643	Janić, Z.	Fizika	24	Dipl.inž.
a8	56321	Popović, G.	Računarstvo	34	Dr.sc.
a9	57432	Simić, J.	Matematika	52	Dr.sc.

Invertovana datoteka

intervali

MAT_BR	*pointer*
12453	a1
16752	a2
27453	a3
34276	a4
37564	a5
43257	a6
45643	a7
56321	a8
57432	a9

Pointer za zapis	*STAROST*
a2,a7	21-30
a1,a3, a6 ,a8	31-40
a5	41-50
a4,a9	51-65

? STAROST=23

ODELJ	*Pointer za zapis*		*ZVANJE*	*Pointer za zapis*
Fizika	a4,a7		Dipl.inž.	a2,a5,a7
Matematika	a1,a2, a6,a9		Mr.sc.	a6
Računarstvo	a3,a5, a8		Dr.sc.	a1,a3,a4 a8 a9

Pristup na osnovu bilo kojeg podatka ostvaruje se tako da u odgovarajućem indeksu pronađemo zadanu vrednost podatka, pročitamo popis pointera, i za svaki od pointera pročitamo zapis u osnovnoj datoteci. Invertovana organizacija omogućuje i brz odgovor na pitanja o postojanju ili broju zapisa sa zadatim svojstvima.

Stabla

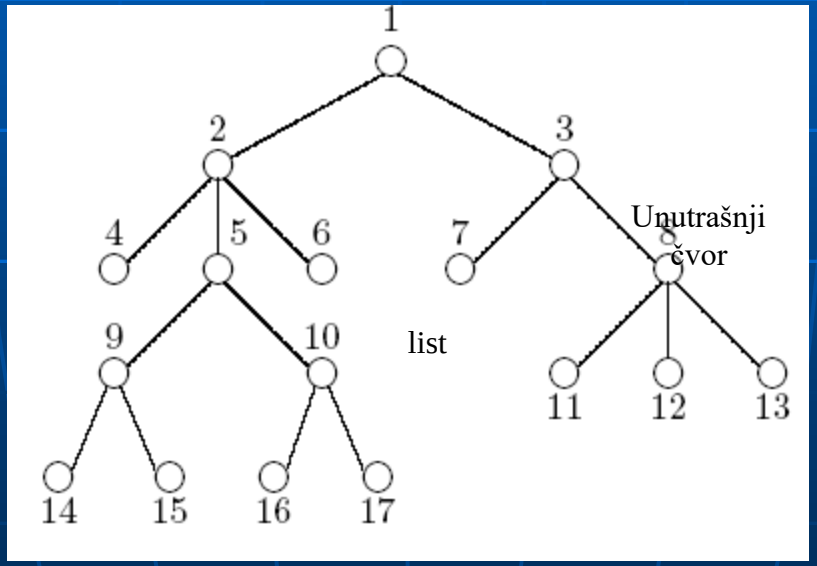
- Vektori i povezane liste su jednodimenzionalne strukture podataka koje
- odražavaju samo redosled elemenata. Ponekad je medjutim potrebno predstaviti i složenije odnose između elemenata.

- Stablo (drvo) je hijerarhijska struktura, ali se može koristiti i za izvodjenje nekih operacija nad linearnim strukturama. Na ovom mestu bavićemo se samo hijerarhijskim ili korenskim stablom.

- Korensko stablo čini skup čvorova i grana, koje povezuju čvorove na specijalan način . Jedan čvor je izdvojen, predstava vrh hijerarhije i zove se koren stabla. Čvorovi vezani sa korenom čine nivo 1 hijerarhije; (novi) čvorovi vezani sa čvorovima nivoa 1 (sem korena) čine nivo 2 hijerarhije, itd. Svaka grana u stablu povezuje neki čvor sa njegovim (jedinstvenim) prethodnikom (ocem); jedino koren nema oca.

Stabla

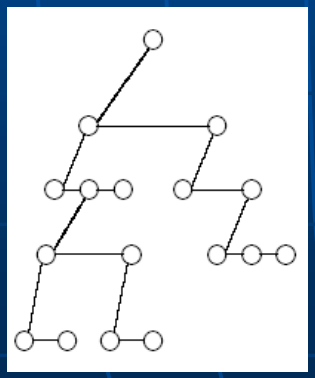
- Osnovna karakteristika stabla je da nema ciklusa (zatvorenih puteva), zbog čega izmedju svaka dva njegova čvora postoji jedinstveni put. Ćvor u stablu vezan je sa ocem i nekoliko sinova. Za čvor *v* koji se nalazi na putu od čvora *u* do korena kaže se da je predak qvora *u*; u tom slučaju je čvor *u* potomak čvora *v*. Maksimalni broj sinova čvora u grafu zove se stepen stabla. Obično je za sinove svakog čvora definisan redosled, tako da se sinovi mogu identifikovati svojim rednim brojem.

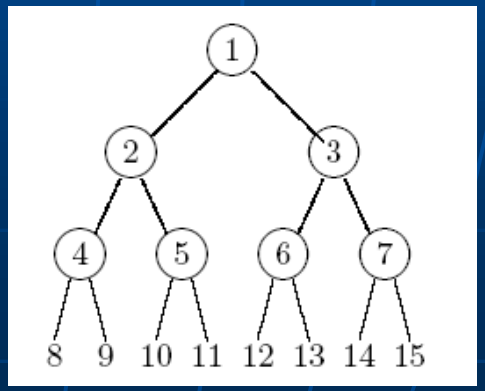


Čvor ima ključ iz nekog potpuno uredjenog skupa (npr. celi ili realni broj). Svaki čvor može da ima polje za podatak - što zavisi od primene.

Stabla

- U zavisnosti od toga da li se koriste pokazivači ili ne, predstavljanje stabla je eksplicitno, odnosno implicitno.
- Pri *eksplicitnom* predstavljanju se čvor sa k sinova predstavlja slogom čiji je deo vektor sa k pokazivača ka sinovima. Pogodno je da svi čvorovi budu istog tipa - sa m pokazivača, gde je m najveći broj sinova nekog čvora, tj. stepen stabla.
- Za *implicitno* predstavljanje stabla ne koriste se pokazivači: svi čvorovi se smeštaju u vektor, a veze izmedju čvorova odredjene su njihovom pozicijom u vektoru. Ako je sa A označen vektor u koji se smeštaju čvorovi binarnog stabla, onda se koren sšexta u $A[1]$; njegov levi, odnosno desni sin zapisuju se u $A[2]$, odnosno $A[3]$, itd. Indukcijom se može pokazati da ako je čvor v zapisan u elementu $A[i]$, onda su njegov levi, odnosno desni sin zapisani u elementu $A[2i]$, odnosno $A[2i+1]$.





Hip

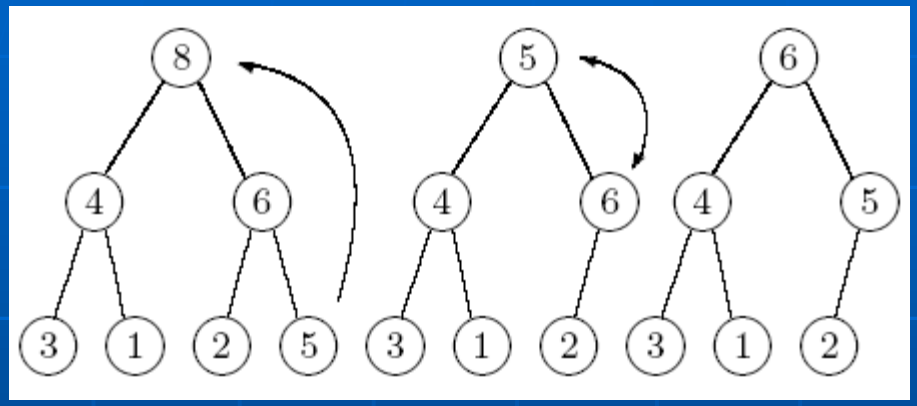
- Hip je binarno stablo koje zadovoava uslov hipa: ključ svakog čvora je veći ili jednak od kučeva njegovih sinova. Neposredna posledica definicije (zbog tranzitivnosti relacije) je da je u hipu ključ svakog čvora veći ili jednak od kučeva svih njegovih potomaka.

- Hip je pogodan za realizaciju liste sa prioritetom, apstraktne strukture podataka za koju su definisane dve operacije:

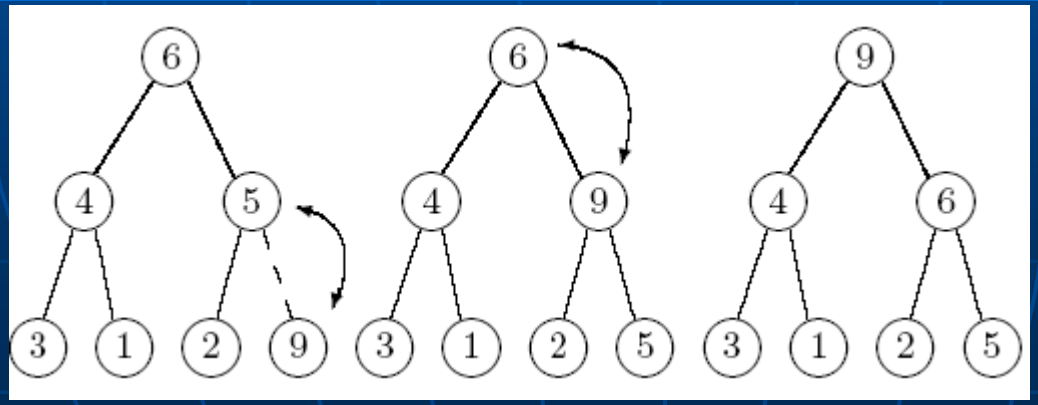
  - umetni(*x*) - umetni ključ *x* u strukturu, i
  - ukloni() - ukloni (obriši) najveći ključ iz strukture.

Hip

- Uklanjanje najvećeg elementa sa hipa:



- Umetanjanje najvećeg elementa u hip:

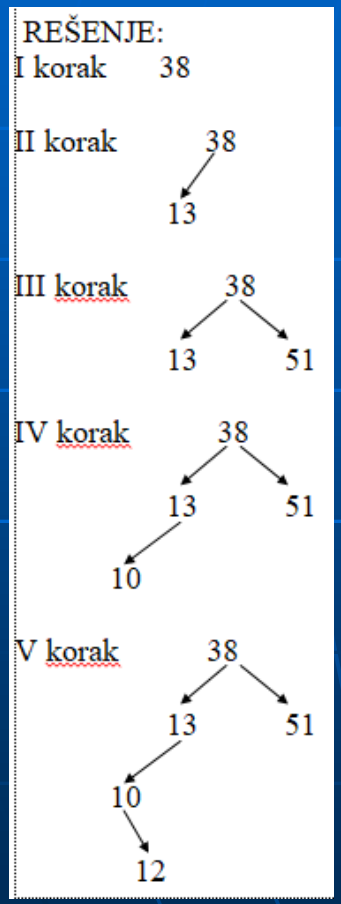


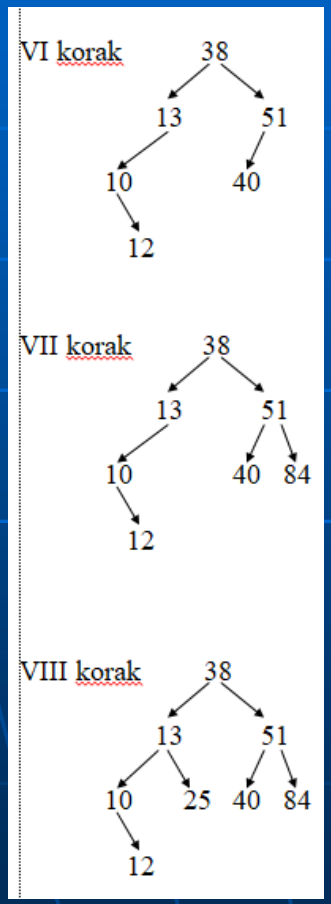
Binarno stablo pretrage

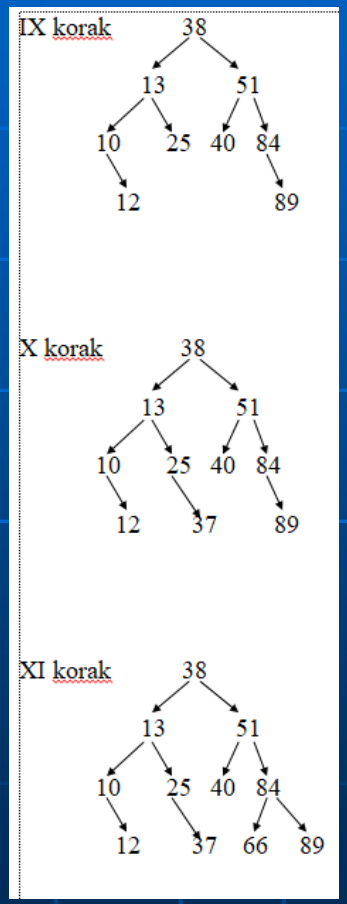
- Svaki čvor binarnog stabla ima najviše dva sina, pa se može fiksirati neko preslikavanje skupa njegovih sinova u skup (levi, desni). Levom, odnosno desnom sinu korena stabla odgovara levo, odnosno desno podstablo. U binarnom stablu pretrage (BSP) **ključ svakog čvora veći je od kučeva svih čvorova levog podstabla, a manji od kučeva svih čvorova desnog podstabla.**
- Pretpostavamo zbog jednostavnosti da su ključevi svih čvorova različiti. BSP omogućuje efikasno izvršavanje sledeće tri operacije:
  - Nadji(*x*) - nadji elemenat sa ključem *x* u strukturi, ili ustanovi da ga tamo nema (pretpostavlja se da se svaki ključ u strukturi nalazi najviše jednom);
  - Umetni(*x*) - umetni ključ *x* u strukturu, ako on već nije u njoj; u protivnom ova operacija nema efekta, i
  - Ukloni(x) - ako u strukturi podataka postoji elemenat sa ključem x, ukloni ga.

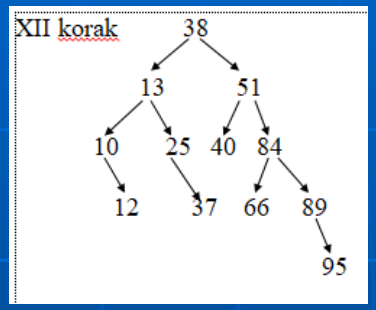
Binarno stablo pretrage

- Zadatak 1: Skicirati binarno stablo pretraživanja koje nastaje dodavanjem sledećih brojeva (u zadatom redosledu) u prazno stablo: 38, 13, 51, 10, 12, 40, 84, 25, 89, 37, 66, 95. Iz kreiranog stabla obrisati čvor sa vrednošću 51 (potrebno je opisati postupak brisanja čvora).





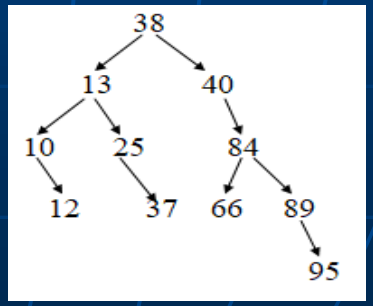




DELETE 51

Ovaj čvor ima oba deteta, tako da brisanje možemo odraditi na sledeći način.

Čvor koji se briše zamenjuje se sa najvećim elementom iz levog podstabla, u ovom slučaju sa čvorom 40.

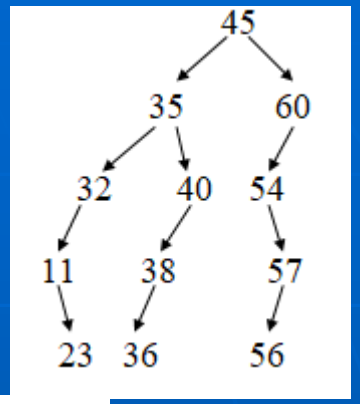


Binarno stablo pretrage

- Zadatak 2: Dato je binarno stablo pretraživanja.

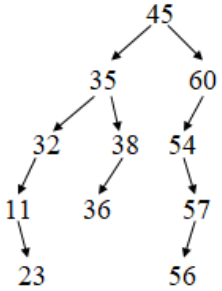
Potrebno je opisati:

a) postupak (korak po korak) brisanja čvora sa vrednošću 40;
b) postupak (korak po korak) brisanja čvora sa vrednošću 45;
c) kako izgleda *inorder* obilazak rezultujućeg stabla.



a) DELETE 40

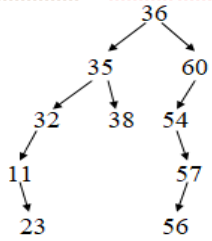
Čvor 40 ima samo jedno dete tako da pri njegovom brisanju na njegovo mesto dolazi upravo čvor 38.



b) DELETE 45

Ovaj čvor ima oba deteta, tako da brisanje mižemo odraditi na način.

Čvor koji se briše zamjenjuje se sa najvećim el iz levog podstabla, u ovom slučaju sa čvorom 36.



c) **Inorder** obilazak:

LEVO podstablo + KOREN + DESNO podstablo

11 23 32 35 36 38 54 56 57 60

Preorder obilazak:

KOREN + LEVO podstablo + DESNO podstablo

54 35 32 11 23 38 36 60 57 56

Postorder obilazak:

LEVO podstablo + DESNO podstablo + KOREN

23 11 32 36 38 35 56 57 60 54

AVL stabla

- AVL stabla (koja su ime dobila po autorima, Adeljson{Veljski, Landis, 1962.) su prva struktura koja garantuje da složenost ni jedne od operacija traženja, umetanja i brisanja u najgorem slučaju nije veća od O(log *n*), gde je *n* je broj elemenata.
- Ideja je uložiti dopunski napor da se posle svake operacije stablo uravnoteži, tako da visina stabla uvek bude O(log *n*). Pri tome se uravnoteženost stabla definiše tako da se moše lako održavati.
- *Definicija* : AVL stablo je binarno stablo pretrage kod koga je za svaki čvor apsolutna vrednost razlike visina levog i desnog podstabla manja ili jednaka od jedan.
- Kao što pokazuje sledeća teorema, visina AVL stabla je O(log *n*):

  Za AVL stablo sa *n* unutrašnjih čvorova visina *h* zadovoljava uslov

$$h < 1.4405 \log_2(n + 2) – 0.327.$$

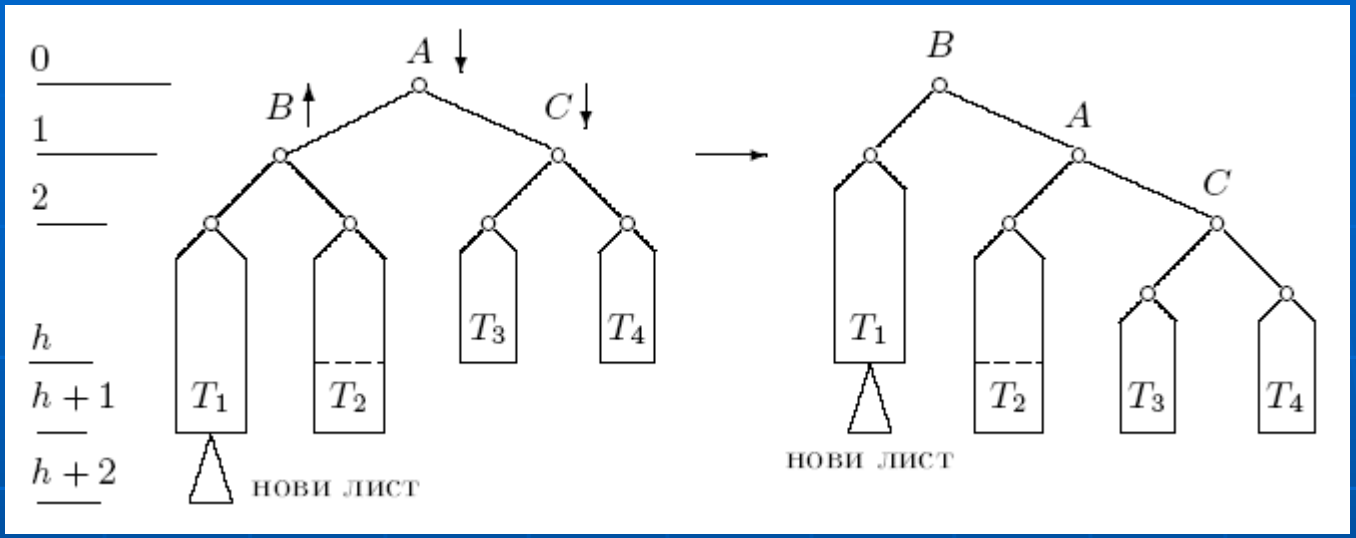
AVL stabla

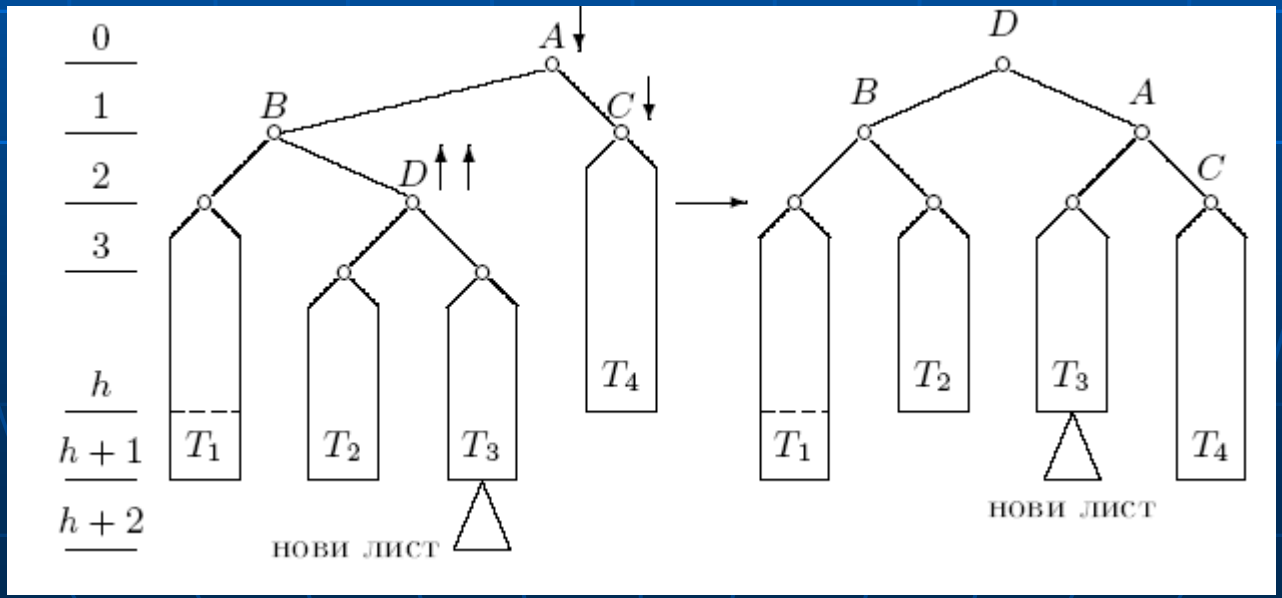
- Prilikom umetanja novog broja u AVL stablo postupa se najpre na način
- uobičajen za binarno stablo pretrage: pronalazi se mesto čvoru, pa se u stablo dodaje novi list sa ključem jednakim zadatom broju. Bez umanjenja opštosti može se pretpostaviti da je novi čvor umetnut u levo podstablo.

- Ako je visina levog podstabla manja ili jednaka od visine desnog podstabla, ili se njegova visina ne menja posle umetanja, onda stablo i posle umetanja ostaje uravnoteženo - visina levog podstabla se umetanjem novog lista ne može povećati za više od jedan.

- U protivnom se, ako je visina levog podstabla veća, i posle umetanja se poveća za jedan, dobija se neuravnoteženo stablo.

Uravnoteženje AVL stabla posle umetanja: rotacija im dvostruka rotacija





Heš tabele

- Heš tabele spadaju medju najkorisnije strukture podataka. Koriste se za umetanje i traženje, a u nekim varijantama i za brisanje.

- Osnovna ideja je jednostavna. Ako treba smestiti podatke sa ključevima iz opsega od 1 do *n*, na raspolaganju je vektor dužine *n*, onda se podatak sa ključem *i* smešta na poziciju *i*, *i* = 1, 2, ..., *n*. Ako su kučevi podataka iz opsega od 1 do 2*n*, još uvek je zgodno podatke smestiti na isti način u vektor dužine 2*n* - time se postiže najveća efikasnost, koja nadoknadjuje utrošak memorijskog prostora.

- Situacija se menja ako je opseg vrednosti kučeva od 1 do *M* veliki (ako je, na primer, *M* najveći prirodni broj koji se može smestiti na konkretnom računaru) - tada otpada varijanta sa korišćenjem vektora dužine *M*.

Heš tabele

- Pretpostavimo da treba smestiti podatke o 250 studenata, pri čemu se svaki od njih identifikuje svojim matičnim brojem sa 13 dekadnih cifara. Umesto kompletnog matičnog broja kao indeks u vektoru mogu se koristiti samo njegove tri poslednje cifre: tada je za smeštanje podataka dovoljan vektor dužine 1000.

- Metod ipak nije potpuno pouzdan: moguće je da neka dva studenta imaju iste tri poslede cifre matičnog broja; načine rešavanja ovog problema razmotrićemo kasnije. Mogu se iskoristiti i četiri poslednje cifre matičnog broja, ili tri poslednje cifre kombinovane sa prvim slovom imena studenta, da bi se još više smanjila mogućnost ovakvog dogadjaja.

- Sa druge strane, korišćnje više cifara zahteva tabelu veće dimenzije, sa relativno manjim iskorišćenim delom.

Heš funkcije

- Ako je veličina tabele m prost broj, a ključevi su celi brojevi, onda je jednostavna i dobra heš funkcija data izrazom:

$$h(x) = x \bmod m$$

- Ako pak m nije prost broj (na primer m = 2k), onda se može koristiti funkcija:

$$h(x) = (x \bmod p) \bmod m$$

gde je p prost broj takav da je $m << p << M$. Neugodna je situacija kad su svi ključevi oblika r + kp, za neki celi broj r, jer će svi ključevi imati istu vrednost heš funkcije r. Tada ima smisla uvesti još jedan nivo randomizacije ("razbacivača"), time što bi se sama heš funkcija birala na slučajan način.

Na primer, broj p mogao bi se birati sa unapred pripremljene liste prostih brojeva. Druga mogućnost je da se koriste heš funkcije oblika:

$$h(x) = (ax + b \bmod p) \bmod m$$

gde su $a \neq 0$ i b slučajno izabrani brojevi manj od p.

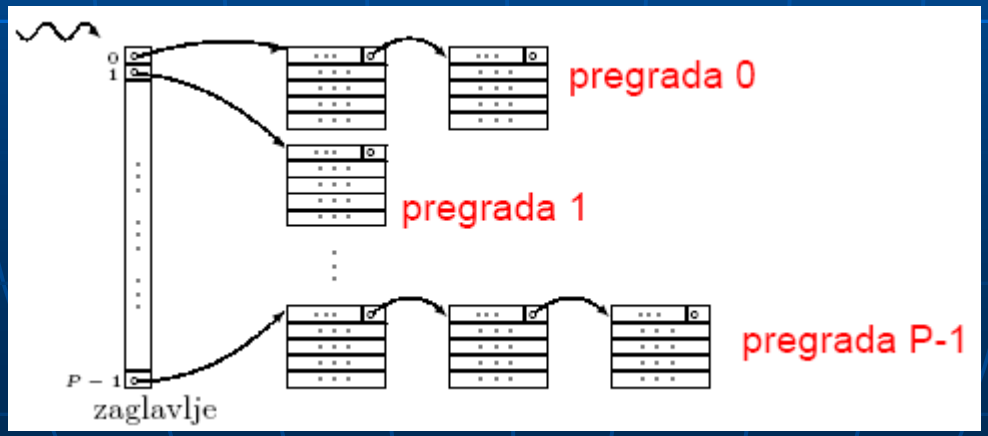
Heš datoteka

- Zapise datoteke smeštamo u P pregrada, označenih rednim brojevima 0, 1, 2, . . . , P − 1. Svaka pregrada se sastoji od jednog ili više blokova. Zadaje se tzv. hash funkcija (h), koja daje redni broj h(k) pregrade u kojoj se treba nalaziti zapis sa vrednošću ključa k.

- Skup mogućih vrednosti ključa obično je znatno veći od broja pregrada. Važno je da (h) uniformno distribuira vrednosti ključa na odgovarajuće pregrade. Tada se neće dešavati da se pregrade neravnomerno pune.

Heš datoteka

- Zapis sa zadanom vrednošću ključa k tražimo tako da izračunamo h(k), i sekvencijalno pretražimo h(k)-ti pregradu. Ako je hash funkcija zaista uniformna, i ako je broj pregrada dobro odabran, tada ni jedna od pregrada nije suviše velika. Pristup na osnovu ključa zahteva svega nekoliko čitanja bloka (obično oko 2 čitanja).



Heš datoteka

- Kao primer, promatramo zapise o studentima fakulteta koje želimo spremati u heš datoteku sa 1024 (= 2^{10}) pregrada. Definicija tipa zapisa u jeziku C izgleda ovako:
  - typedef struct { int SNO; // Matični broj
  - char SNAME[20]; // Ime
  - enum {1,2,3,4} LEVEL; // Godina
  - enum {M,F} GENDER; // Pol
  - } STUDENT;
- Podelimo 10 bitova u rednom broju pregrade na sledeći način: 4 bita za SNO, 3 bita za SNAME, 2 bita za LEVEL, 1 bit za GENDER.
- Zadajemo sledeće heš funkcije, gde su v1, v2, v3, v4 vrednosti za SNO, SNAME, LEVEL, GENDER respektivno:
  - h1(v1) = v1 % 16 (ostatak kod deljenja sa 16),
  - h2(v2) = (broj znakova u v2 koji su različiti od blanko) % 8,
  - h3(v3) = v3 − 1,
  - h4(v4) = (0 za v4 jednak M, 1 inače).
- Tada je redni broj pregrade za zapis (58651, Smith, 3, M) zadan sa (1101|101|10|0)2 = (748)10.

Stek i slobodno skladište

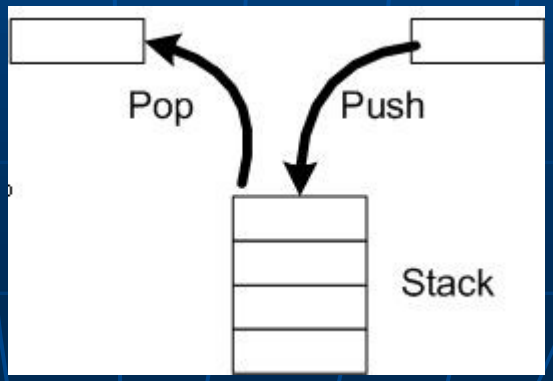
- U okviru organizacije podataka pomenućemo pet područja memorije:

  1. prostor globalnih imena,
  2. slobodno skladište,
  3. registri,
  4. prostor za kod i
  5. stek.

- Globalne promenljive se nalaze u prostoru globalnih imena. Registri se koriste za interne domaćinske funkcije. Kod se nalazi u prostoru za kod, naravno, kao što je pamćenje vrha steka i pokazivača instrukcija.
- Lokalne promenljive su na steku, zajedno sa parametrima funkcije. Skoro sva preostala memorija se dodeljuje slobodnom skladištu. Problem sa lokalnim promenljivama je taj što one nisu trajne: Po povratku iz funkcije, one se odbacuju. Globalne promenljive rešavaju taj problem po cenu neograničenog pristupa širom programa, što vodi do kreiranja koda koji je težak za razumevanje i održavanje.
- Stavljanje podataka u slobodno skladište rešava oba ova problema. O slobodnom skladištu možete razmišljati kao o masivnoj sekciji memorije, u kojoj hiljade sekvencijalno označenih kockica leže, čekajući vaše podatke.

Stek

- Stek (engl. *Stack)* je struktura koja funkcioniše na principu LIFO (*last in, first out*). Zahvaljujući ovom principu element koji je zadnji bio ubačen prvi je na redu za izbacivanje, tj. redosled izbacivanja elemenata je suprotan redosledu ubacivanja.
- Stek ima dve osnovne operacije koje su konstantne složenosti, a to su *push*, koja dodaje element sa zadatim podatkom na vrh steka, i *pop* koja skida element sa vrha steka.
- Može se implementirati na razne načine, ali najčešće se to čini korišćenjem nizova ili jednostruko povezanih listi.
- Zbog svoje široke primene i čestog korišćenja, stek se takođe posmatra i kao posebni apstraktni tip.



U računarstvu, **stek** je privremeni apstraktni tip podataka i struktura podataka, baziran na principu LIFO (LIFO - Last In First Out - poslednji koji ulazi, prvi izlazi). Stekovi se u velikoj meri koriste na svim nivoima modernog računarskog sistema. Stek-mašina je računarski sistem koji privremene podatke skladišti prvenstveno u stekovima, umesto u hardverskim registrima.

Stek i slobodno skladište

- Tipičan stek skladišti lokalne podatke i informacije o pozivu za ugneždene procedure. Ovaj stek raste na dole od početka. Pokazivač na stek pokazuje na trenutni vrh steka.

  - Operacija *push* umanjuje pokazivač, i kopira podatke na stek;
  - operacija *pop* vraća podatke sa steka, a zatim pokazivač pokazuje na novi vrh steka.

- Svaka procedura pozvana unutar programa skladišti svoje povratne informacije (žuto) i lokalne podatke (u drugim bojama) gurajući ih na stek. Ovakva implementacija steka je izrazito česta, ali je ranjiva na napade prekoračenja bafera. Tipičan stek je deo u memoriji računara sa fiksiranim početkom, i promenljivom veličinom. Na početku, veličina steka je nula. Pokazivač na stek, obično u vidu hardverskog registra, pokazuje na poslednju iskorišćenu adresu na steku; kada je stek veličine nula, pokazivač pokazuje na početak steka.

  Dve operacije primenljive na sve stekove su:

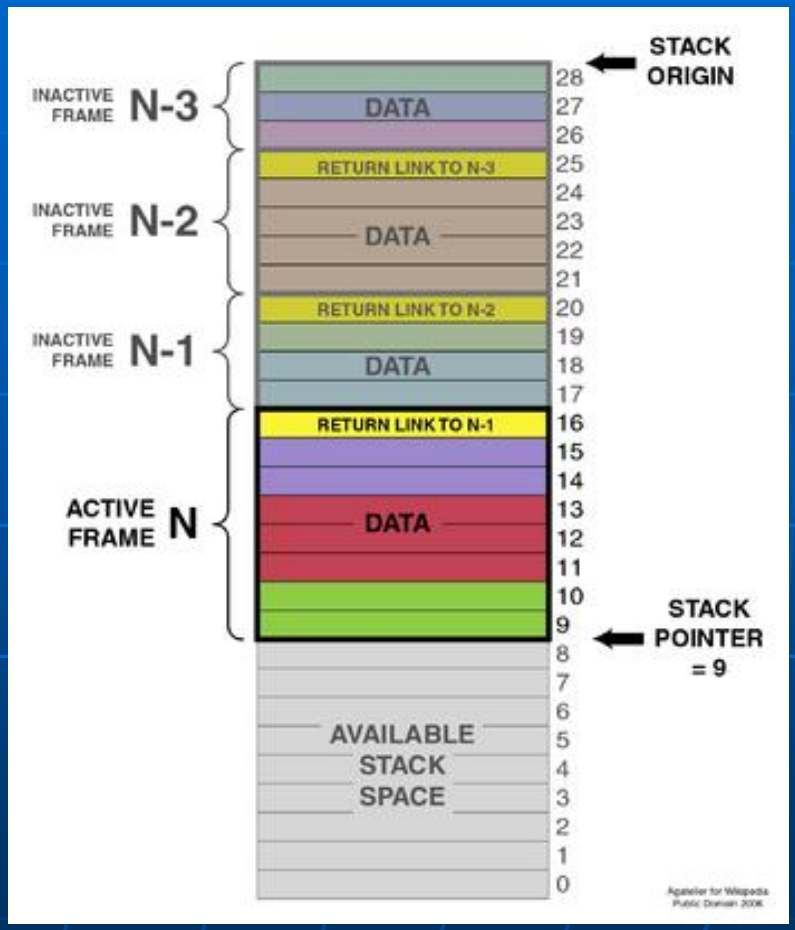
- *push* operacija, u kojoj se predmet postavlja na lokaciju na koju pokazuje pokazivač na stek, a adresa u pokazivaču se prilagođava za veličinu tog predmeta;
- *pop* ili *pull* operacija: predmet na trenutnoj lokaciji na koju pokazuje pokazivač se uklanja, a adresa u pokazivaču se prilagođava za veličinu tog predmeta

Stek i slobodno skladište

- Dve operacije primenljive na sve stekove su:

- *push* operacija, u kojoj se predmet postavlja na lokaciju na koju pokazuje pokazivač na stek, a adresa u pokazivaču se prilagođava za veličinu tog predmeta;

- *pop* ili *pull* operacija: predmet na trenutnoj lokaciji na koju pokazuje pokazivač se uklanja, a adresa u pokazivaču se prilagođava za veličinu tog predmeta



Stek i slobodno skladište

- Stek se automatski čisti po povratku iz funkcije. Sve lokalne promenljive izlaze iz opsega i uklanjaju se sa steka.

- Slobodno skladište se ne čisti sve dok se program ne završi i naš je zadatak da oslobodimo svaki memorijski prostor koji smo rezervisali. Prednost slobodnog skladišta je što memorija koju rezervišemo ostaje raspoloživa, dok je eksplicitno ne oslobodimo. Ako rezervišemo memorju na slobodnom skladištu u funkciji, memorija je još uvek raspoloživa po povratku iz funkcije.

- Prednost pristupanja memoriji na ovaj način, umesto korišćenja globalnih promenljivih, je to da samo funkcije sa pristupom pokazivaču imaju pristup podacima.

- U aplikacionim programima pisanim u višim programskim jezicima, stek se može efikasno primeniti bilo pomoću nizova bilo pomoću povezanih listi. U jeziku Lisp nema potrebe da se implementira stek, jer su funkcije push i pop dostupne za svaku listu. AdobePoscript je takođe dizajniran oko steka koga programer direktno može da vidi, i da njime manipuliše.

Stek i slobodno skladište

- - operatori new i delete -
- NEW : memoriju na slobodnom skladištu u C alociramo korišćenjem ključne reči new. Posle nje sledi tip objekta koji želimo da alociramo tako da kompajler zna koliko se memorije zahteva. New unsigned short int alocira dva bajta na slobodnom skladištu, a new long alocira četiri. Povratna vrednost iz new je memorijska adresa. Ona se mora dodeliti pokazivaču. Da biste kreirali unsigned short na slobodnom skladištu, napišite:
- *unsigned short int *pPointer;*
- *pPointer = new unsigned short int;*
- U svakom slučaju, pPointer sada pokazuje na unsigned short int na slobodnom skladištu. Njega možete korisriti kao i svaki drugi pokazivač na promenljivu i dodeliti vednost području memorije.
- UPOZORINJE: Svaki put kada alociramo memoriju, korišćenjem ključne reći new, moramo proveriti da bismo bili sigurni da pokazivač nije nula.

Stek i slobodno skladište

- Primer, izraz: ((1 + 2) * 4) + 3 može biti zapisan na sledeći način u postfiksnoj notaciji uz prednost da nisu potrebna pravila prednosti i zagrade:

  1 2 + 4 * 3 +

- Izraz se izračunava sleva na desno korišćenjem steka:

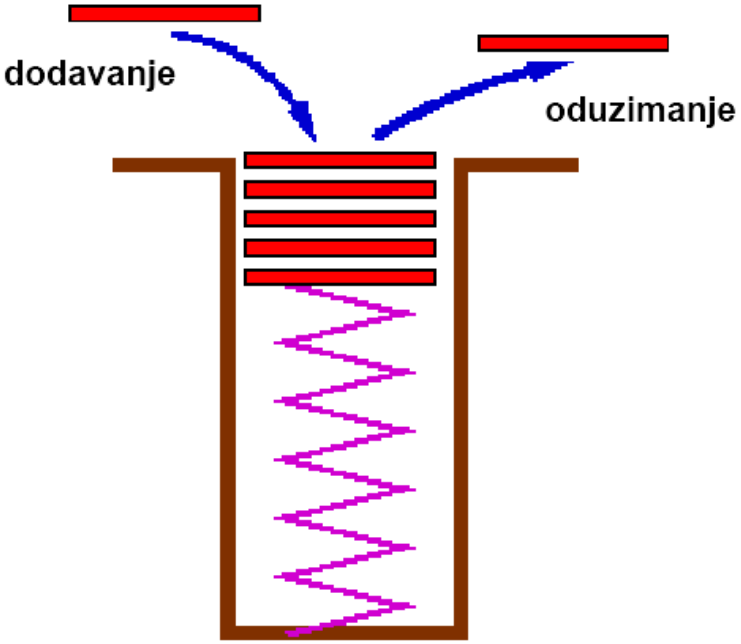
- *push* kad se dođe do operanda

- *pop* dva operanda i njihovo računanje, kad se dođe do operacije praćeno *push* rezultatom.

Улаз	Операција	Стек
1	*Push* операнд	1
2	*Push* операнд	1, 2
+	Сабирање и Pop последња два операнда и Push резултат	3
4	*Push* операнд	3, 4
*	Множење и Pop последња два операнда и Push резултат	12
3	*Push* операнд	12, 3
+	Сабирање и Pop последња два операнда и Push резултат	15

Stek

- Polja i povezane liste dozvoljavaju dodavanje i oduzimanje (brisanje) elemenata na bilo kom mestu - početku, kraju ili između.
- Postoje određene situacije u programiranju kada želimo ograničiti dodavanja i brisanja elemenata samo na kraj ili početak liste.
- Linearna struktura u kojoj se elementi mogu dodavati ili oduzimati samo na jednom kraju zove se stek (eng. stack).



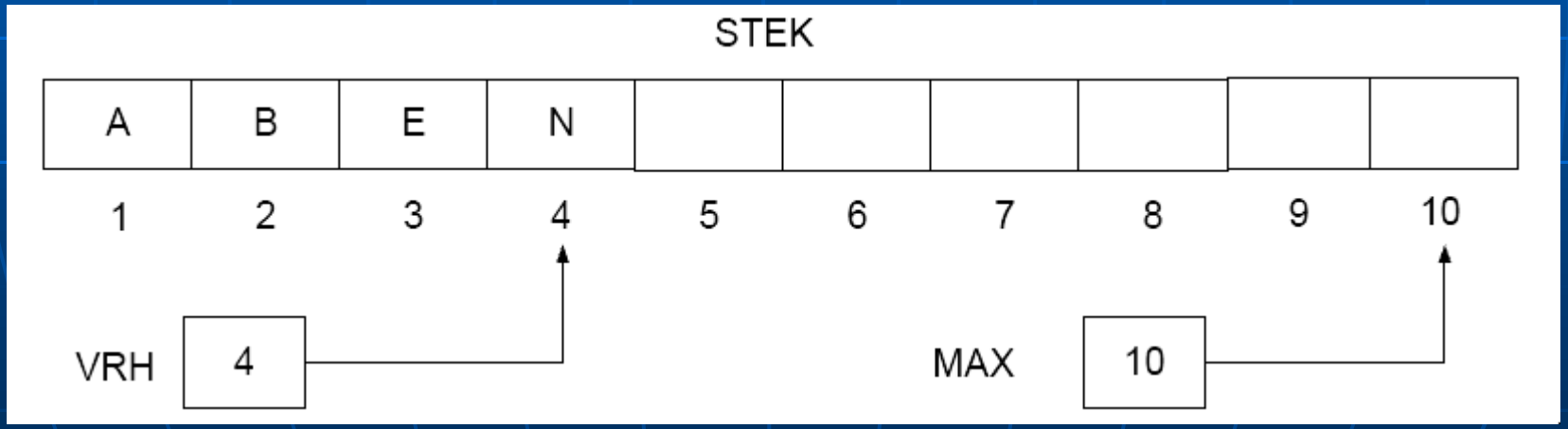
Osnovne operacije na stogu su:

- **dodavanje (umetanje) elementa na stog (eng. push)**
- **brisanje (oduzimanje) elementa iz stoga (eng. pop)**

Stek

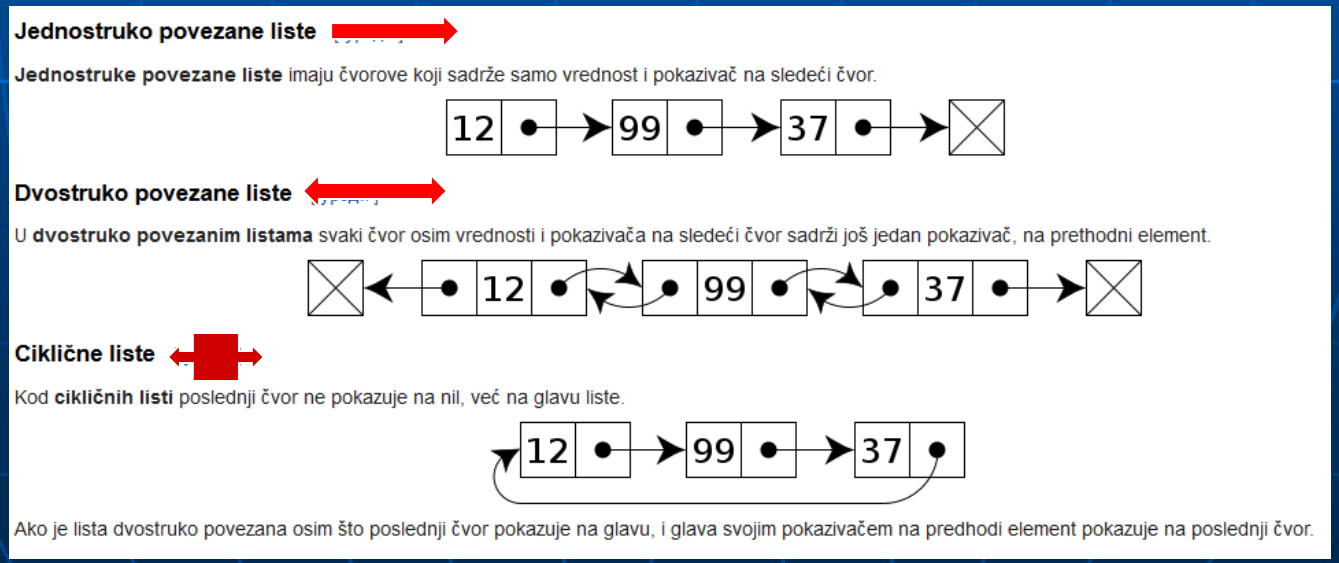
• Prikaz i manipulacija sa stekom može se realizirati statičkom strukturom podataka. U jednodimenzionalno polje zadane strukture dodaju se ili brišu pojedine stavke po principu "*LastIn-FirstOut*" (*LIFO*).

• Na slici je prikazan je primer implementacije strukture steka pomoću jednodimenzionalnog polja "STEK", i promenljive "VRH" koja sadrži trenutnu poziciju najvišeg (gornjeg) elementa i promenljive "MAX" koja sadrži maksimalni broj elemenata u steku.



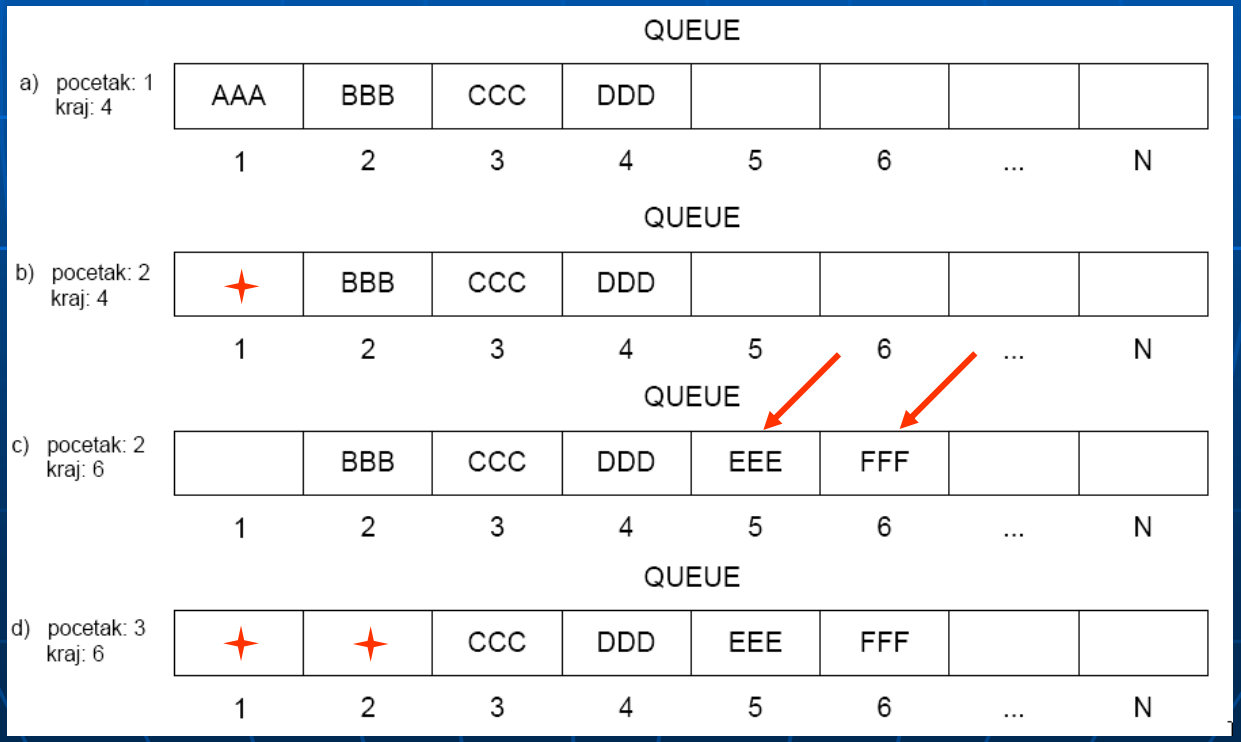
Povezana lista

- Povezana lista je struktuira podataka, koja je u osnovi predstavljena kao vektor parova (element, pokazivač), pri čemu pokazivač sadrži adresu narednog para. Tako postavljeni parovi nazivaju se čvorovima. Prolazak kroz listu moguć je jedino linearno - redom od početka, element po element, prateći pokazivače.
- Nedostaci ove strukture podataka su u tome što zahteva dodatni prostor u memoriji (uz svaki element ide i pokazivač), a *k-tom* (k>1) elementu se može pristupiti samo preko svih predhodnih. Prednost povezane liste je u tome što se upis i brisanje lako realizuju, potrebno je menjanje samo pokazivača (jednog u slučaju brisanja, dva u slučaju upisivanja)



Red (queue)

- Red (eng. queue) je linearna struktura u kojoj se elementi dodaju isključivo na jednom kraju, a oduzimaju isključivo na drugom kraju.
- Strukturu reda lako je predočiti analogijom sa istim pojmom u svakodnevnom životu - npr. redom ispred blagajne u dućanu. Svaka nova osoba koja dođe zauzima mesto na kraju reda, a osoba sa početka reda plaća svoju robu i odlazi.
- Drugi uobičajeni naziv za ovu strukturu je .FIFO. lista (eng. first in - first out).



Struktura stabla - tree

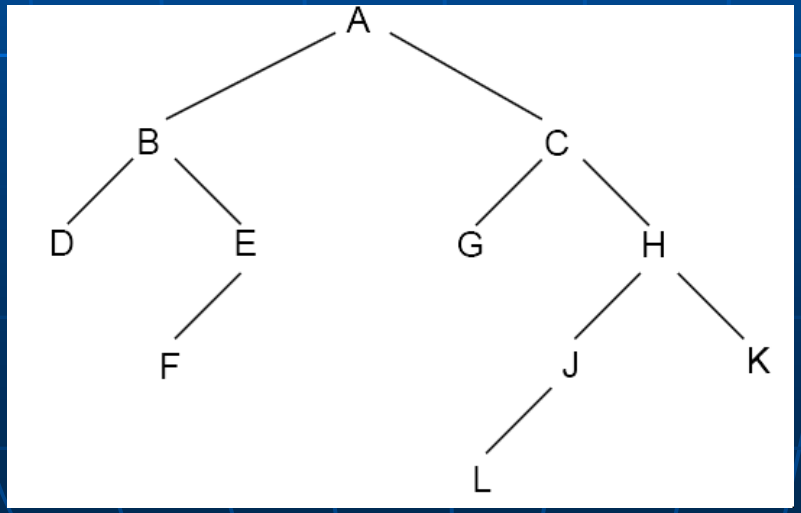
- Za razliku od prethodno razmatranih linearnih struktura, stablo (eng. tree) je nelinearna struktura koja se koristi za prikaz hijerarhijskih odnosa između elemenata.
- Primer strukture stabla koju koristimo u svakodnevnom radu je struktura direktorijuma, odnosno organizacija sistema datoteka na računaru.
- Strukture stabla koje ćemo ovde razmatrati sastoje se od čvorova (eng. nodes) i veza između njih koje uobičajeno nazivamo grane.
- Sastoji se od čvorova i grana između njih. Svaka grana povezuje jedan čvor sa njegovim čvorom sledbenikom. Često se koriste termini roditelja i deteta.
- Čvor koji se zove koren stabla nema roditelja, dok svaki drugi ima tačno jednog i on je predak svim čvorovima stabla.
- Čvor bez potomaka se naziva list. Ovakvo uredjenje nam omogućava da preko korena jedinstveno određenim putem dodjemo do bilo kog čvora stabla i jednostavno se mogu dodavati novi čvorovi.

Struktura stabla - tree

- Jedna od najvažnijih vrsta strukture stabla u računarstvu je tzv. binarno stablo. U binarnom stablu (eng. binary tree) svaki čvor ima ili najviše dva sledbenika ili samo jednog ili nijednog sledbenika.

- Na slici je prikazan je primer binarnog stabla sa 11 čvorova. Čvorovi A, B i C imaju dva sledbenika, čvorovi E i J po jednog sledbenika, a čvorovi D, F, G, L i K nemaju sledbenika.

- Čvorovi bez sljedbenika nazivaju se terminalni čvorovi.



Graf

- Struktura grafa (engl. *Graph*) je nelinearna struktura koja sadrži dva konačna skupa, skup čvorova koji se obeležava sa V i skup grana sa ozakom E. Dakle G= (V, E).
- Grana, često nazivana i brid, odgovara paru čvorova, tj. grane predstavljaju relaciju između čvorova. Na primer, graf može da predstavlja skup ljudi, a da grana povezuje dva čoveka ukoliko se oni poznaju. Jednostavan primer grafa bi bilo stablo.
- Graf je usmeren ukoliko su mu grane ure djeni parovi, odnosno neuređen ako nisu.
- Čvor usmerenog grafa u oznaci e = (i,j) ima smer od "inicijalne tačke“ (inicijalnog čvora) do "terminalne tačke" (terminalnog čvora).
- U usmerenom grafu mogu postojati dva čvora koja povezuju iste čvorove ukoliko su suprotnih smerova.

Graf (graph)

- Struktura grafa (eng. graph) je takođe nelinearna.

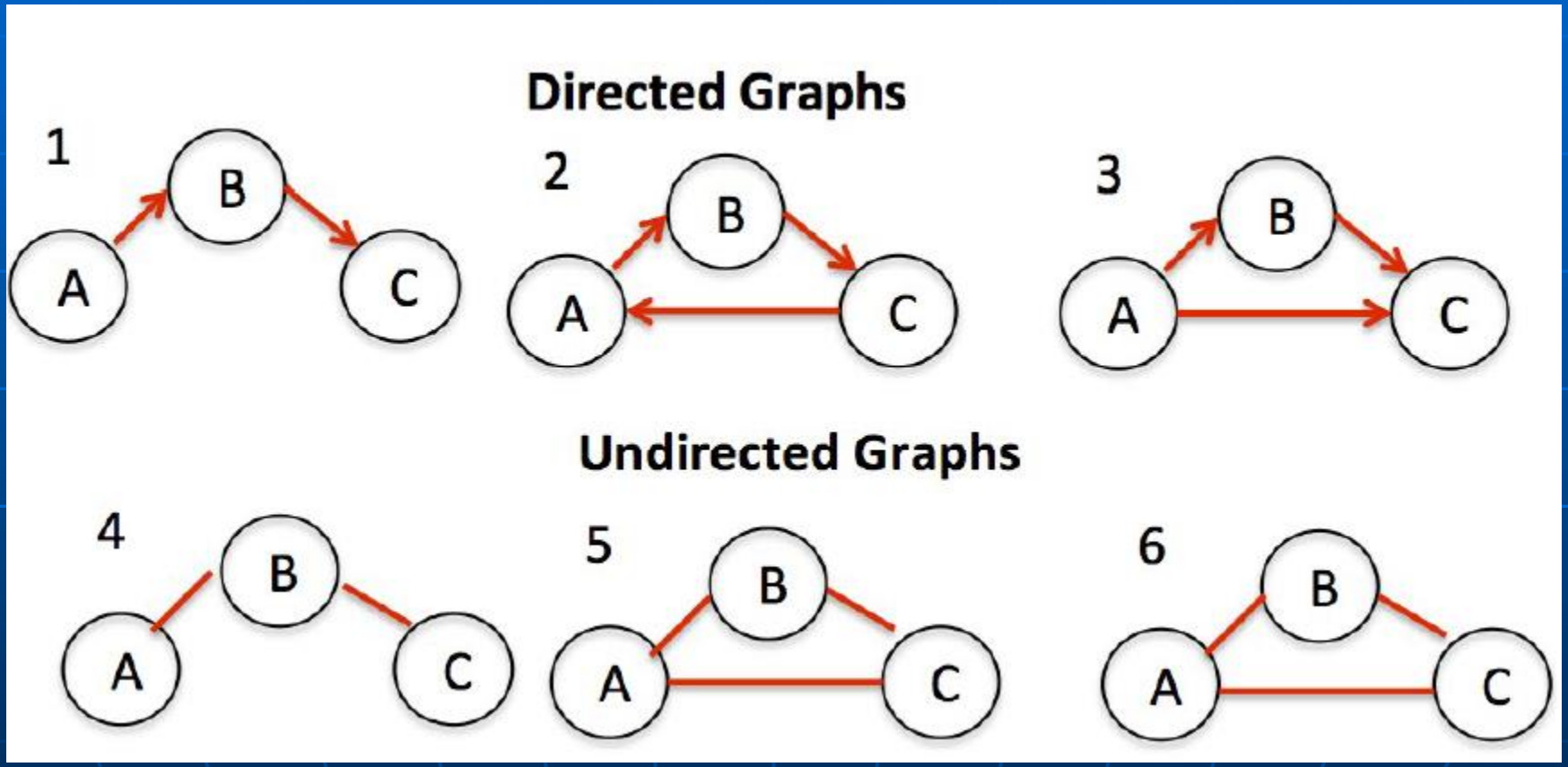
- Definicija grafa:
  - Graf *G* sadrži dva konačna skupa: skup tačaka *V*, koje nazivamo čvorovima, i skup linija povezivanja *E*, koje nazivamo bridovima. Pri tome svaki brid povezuje dva čvora.

$$G = (V, E)$$

- Definicija usmerenog grafa:
  - *Usmereni* graf $G = (V, E)$ je graf u kojem svaki brid e = (i,j) ima smer od "inicijalne tačke“ (čvora) do "terminalne tačke" (čvora). Pod uvslovom da su suprotnih smerova, u usmerenom grafu mogu postojati dva brida koja povezuju iste čvorove.

- Graf *G* je *povezan* ako i samo ako postoji jednostavna putanja između bilo koja dva čvora u *G*.

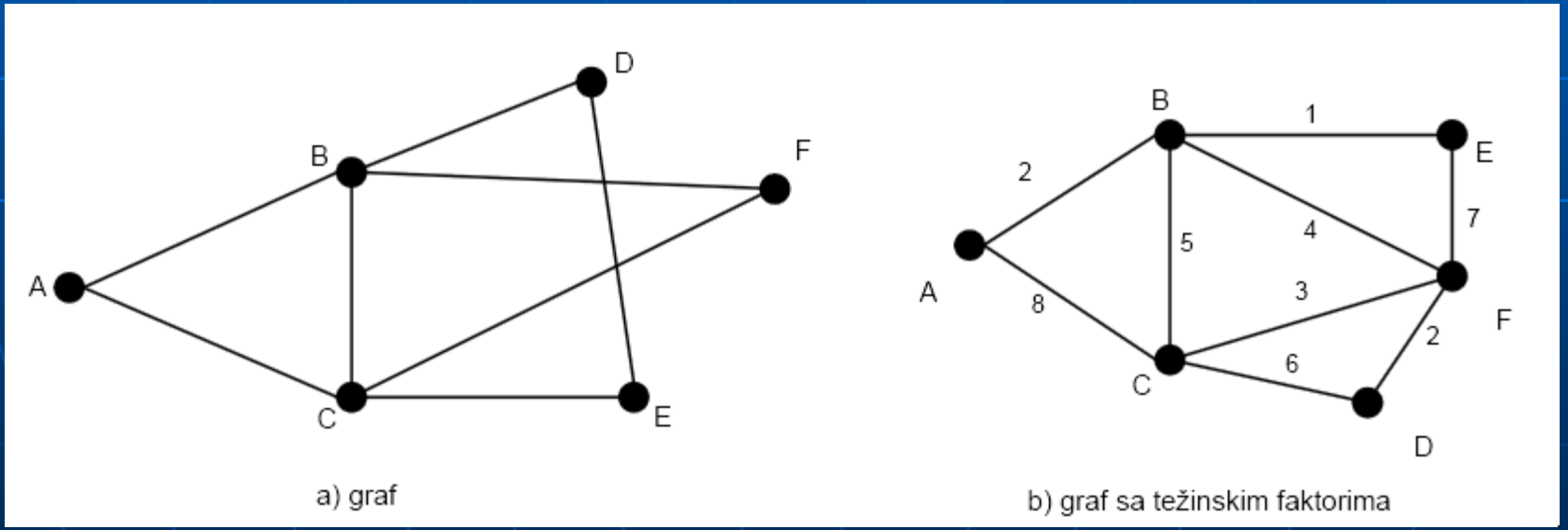
Usmereni i neusmereni grafovi



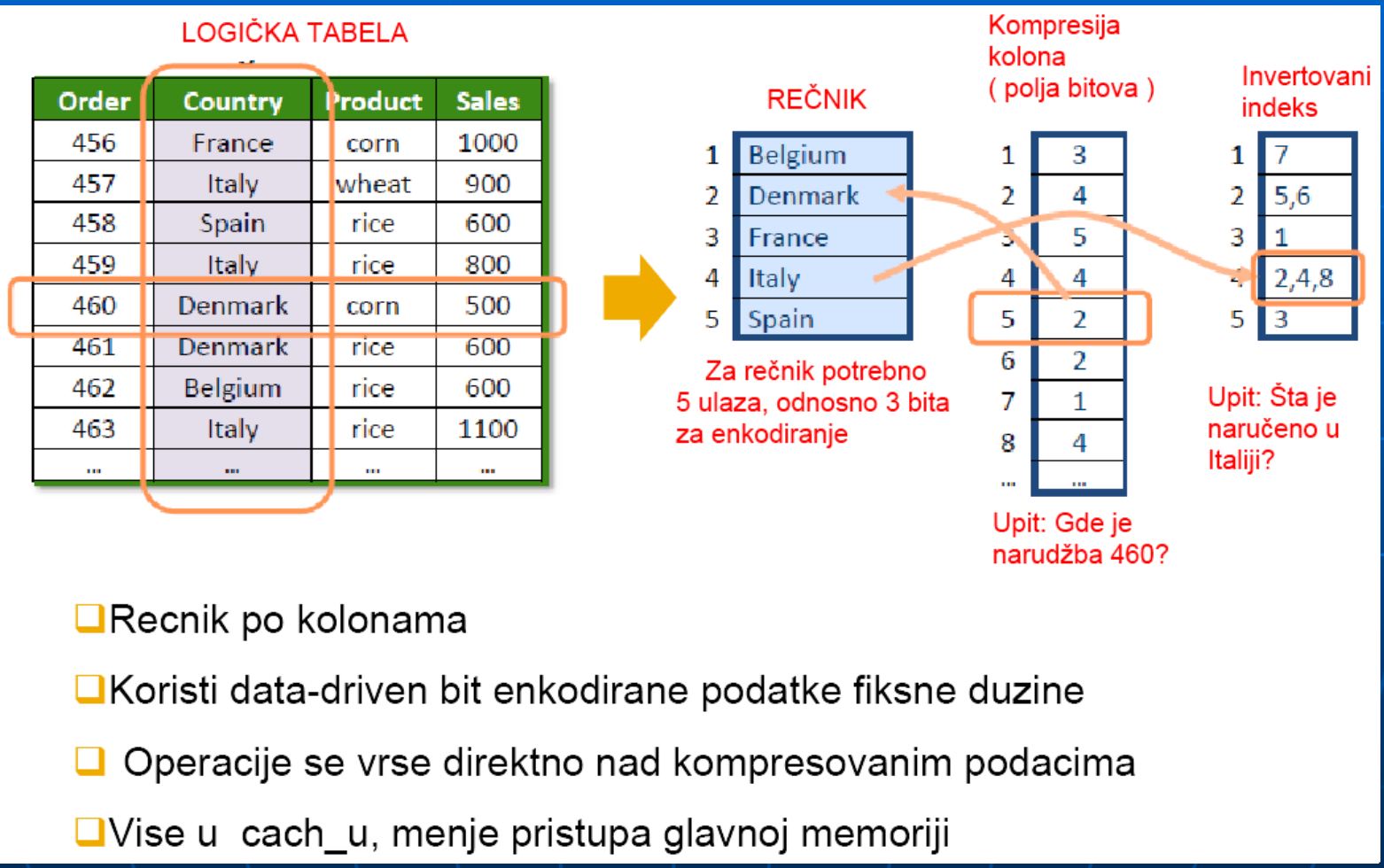
Graf (graph)

- Ako svakom bridu grafa pripada neki podatak, takav graf se naziva *označenim*.

- Ako su bridovima grafa pridružene pozitivne numeričke vrednosti, onda se takav graf naziva *težinski* ili graf sa težinskim faktorima.

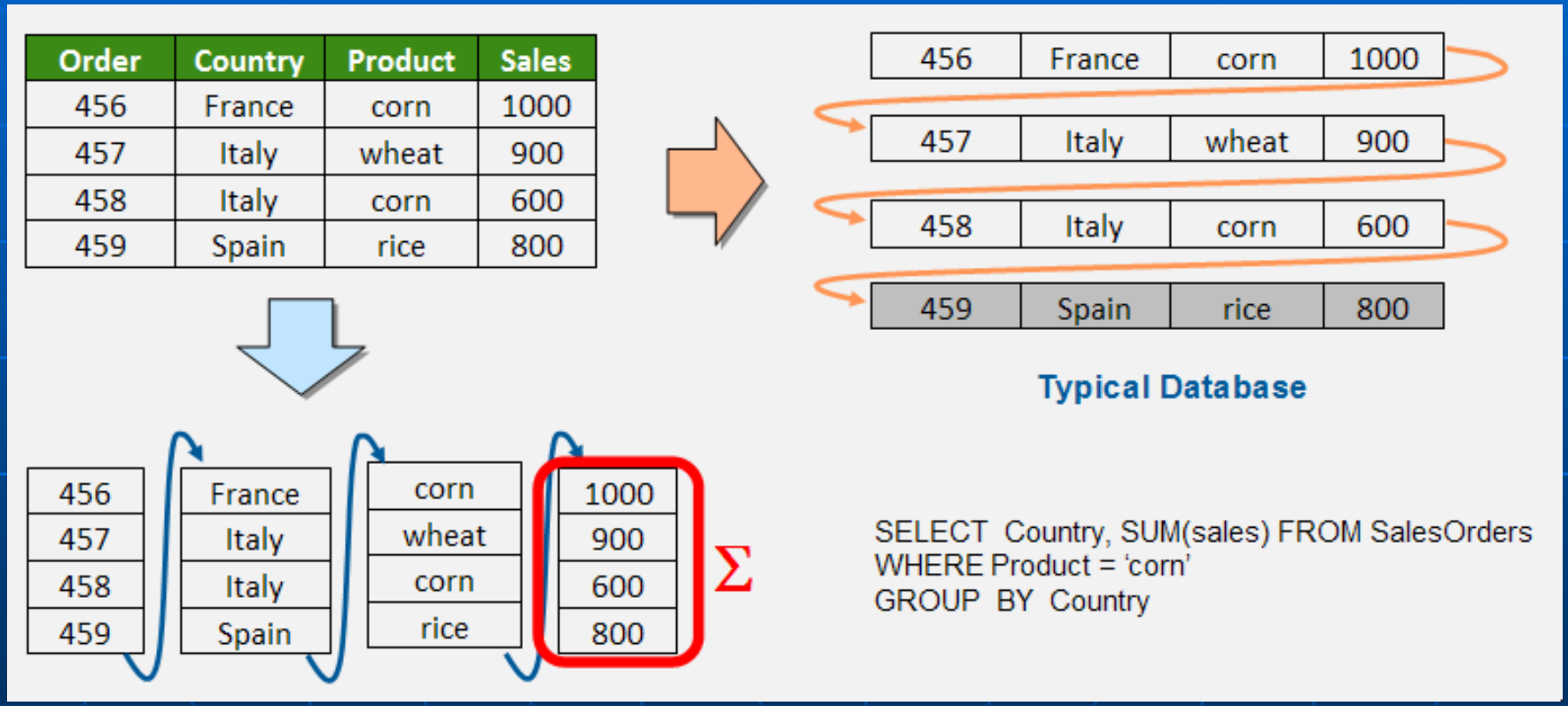


In Memory organizacija



- Recnik po kolonama
- Koristi data-driven bit enkodirane podatke fiksne duzine
- Operacije se vrse direktno nad kompresovanim podacima
- Vise u cach_u, menje pristupa glavnoj memoriji

In Memory organizacija

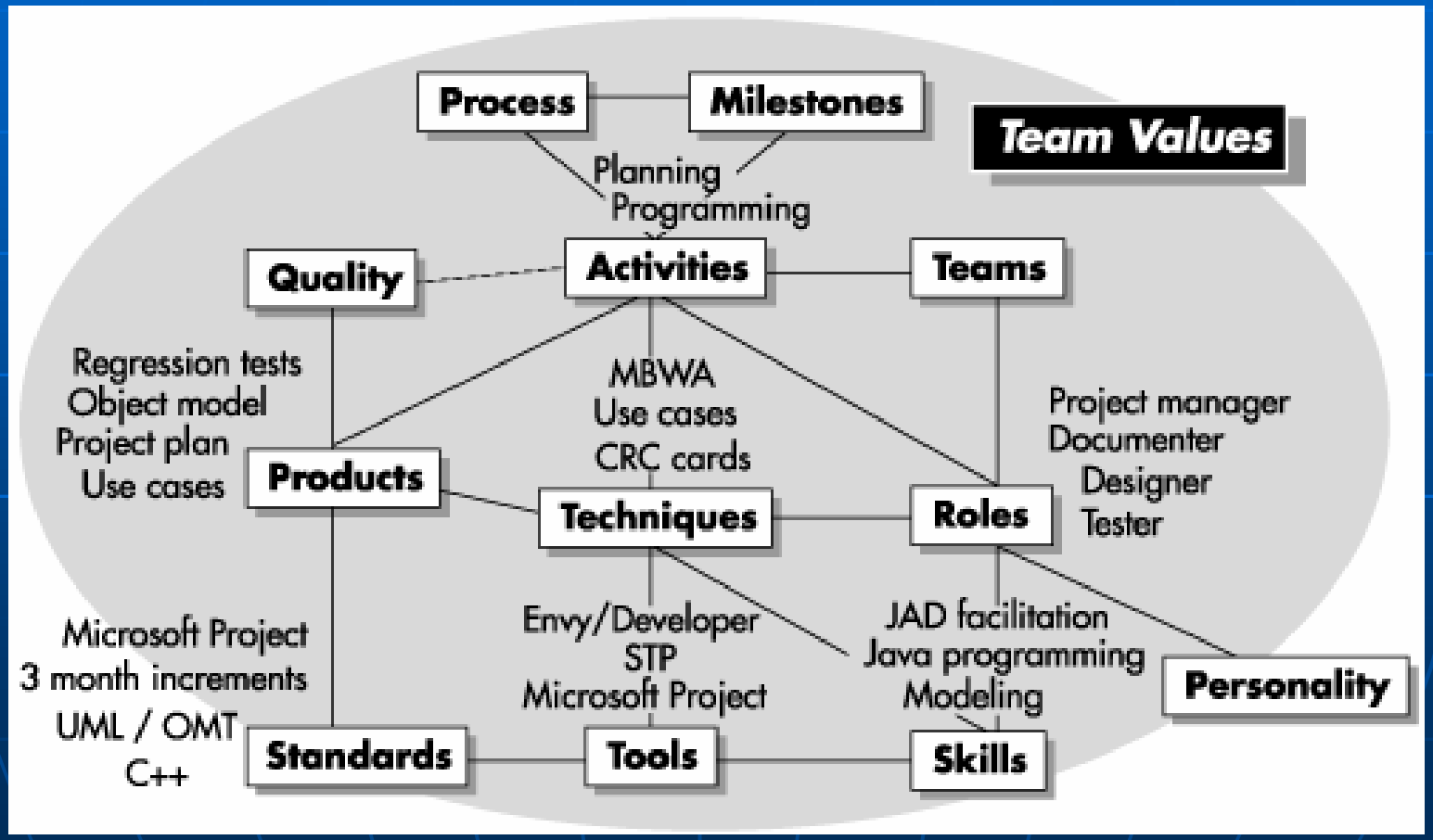


Metodologije za programiranje i razvoj softvera

- Metodologija se može definisati kao “skup međusobno povezanih metoda i tehnika“, gde se pod metodama podrazumevaju „sistematične procedure“ slične tehnikama.

- Metodologije za razvoj softvera bave se „metodama i tehnikama“ za razne elemente koji se pojavljuju u procesu razvoja softvera. Na sledećoj slici prikazani su najvažniji elemenati razvoja softvera od kojih zavisi uspešnost svakog softverskog poduhvata.

Metodologije za programiranje i razvoj softvera



Metodologije za programiranje i razvoj softvera

1. Softverski timovi (Teams) koji se sastoje od menadžera, dizajnera, testera, dokumentalista i sl.
2. (Roles) koji poseduju potrebna znanja i veštine (Skills),
3. Svojim svakodnevnim aktivnostima (Activities),
4. Kao što su planiranje i programiranje kroz procese (Process),
5. A korišćenjem različitih tehnika (Techniques)
6. i alata (Tools)
7. Stvaraju nove oftverske proizvode (Products)
8. Određenog kavliteta (Quality)
9. I po „de fakto" ili „de jure" standardima (Standards).
10. Naravno, ne treba zaboraviti da su timovi sastavljeni od ljudi koji imaju svoje osobenosti (Personality) o kojima takođe treba voditi računa.

Životni ciklus razvoja softvera

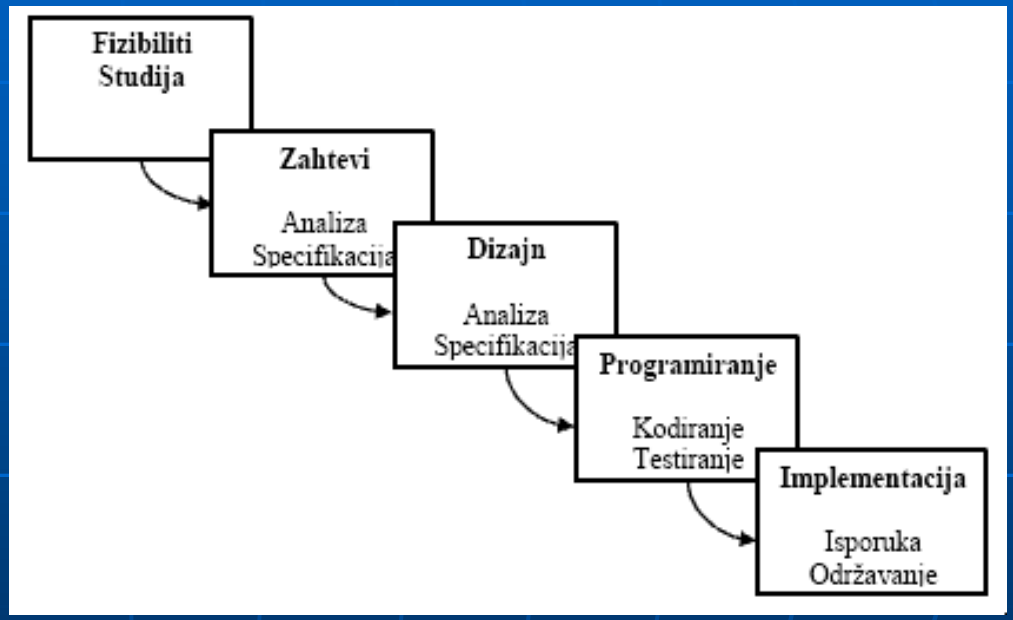
- Tokom vremena razvijeno je više modela za razvoj softverskih sistema poznatim pod nazivom SDLC (System Development Life Cycle) modeli.
- SDLC modeli imaju za cilj da daju jedan metodičan pristup za razvoj informacionih sistema i to za sve faze razvoja:
  - od izrade studije izvodljivosti (feasibility study),
  - do održavanja gotovih aplikacija.
- Ukratko ćemo navesti neke od najpoznatijih SDLC modela za razvoj softvera.

Tunel (Tunnel) model:

Model tunela : Ovo je ilustrativni način da se izrazi nepostojanje modela razvoja. U projektima sa tunelskim pristupom nemoguće je znati šta se dešava. Razvoj napreduje, ljudi rade - često vrlo naporno, ali nema pouzdane informacije o tome kako softver napreduje ili kakvog su kvaliteta razvijeni elementi. Ovaj model razvoja odgovara samo malim projektima sa vrlo ograničenim brojem učesnika.

Vodopad (Waterfall) model:

- To je klasičan SDLC model kod kojeg se faze u razvoju softvera linearno i sekvencijalno odvijaju jedna za drugom bez preklapanja faza i/ili vraćanja (iteracije) na prethodno završenu fazu.
- To se može islustrovati sledećom slikom:



Fizibiliti

- Fizibiliti studijom se određuje da li vredi uopšte započeti rad na nekom softverskom projektu.

- Ako je odluka da se projekat realizuje, onda se već u samoj fizibiliti studiji daju osnove plana izvođenja projekta i procena budžeta (i drugih resursa) potrebnih za realizaciju.

Zahtevi – projektni zadatak

- Zahtevi za izradu novog ili modifikaciju postojećeg sistema predstavljaju detaljan opis željene funkcionalnosti i drugih karakteristika softverskog sistema koji će biti razvijen.
- Zahtevi moraju biti što precizniji i u pisanoj formi, kako bi se na kraju projekta moglo ustanoviti da li realizovani sistem odgovara zahtevima.

Dizajn

- Dizajn se može podeliti na tri dela:

- Viši nivo kojim se određuje koji su sve programski moduli potrebni, šta su njihovi ulazi/izlazi i kako oni interaguju međusobno i sa drugim softverom i operativnim sistemom.

- Niži nivo na kojem se definiše način rada programskih modula, koji algoritmi i modeli će biti korišćeni, koje su programske biblioteke potrebne i sl.

- Dizajn podataka kao što je interfejs za ulaz/izlaz podataka, strukture podataka koje će se koristiti i sl.

Programiranje i testiranje

- U ovoj fazi se dizajn pretvara u programski kod.

Programski alati kao što su kompajleri i dibageri se pri tome koriste za generisanje izvornog koda dobrog kvaliteta a time i celokupne softverske aplikacije.

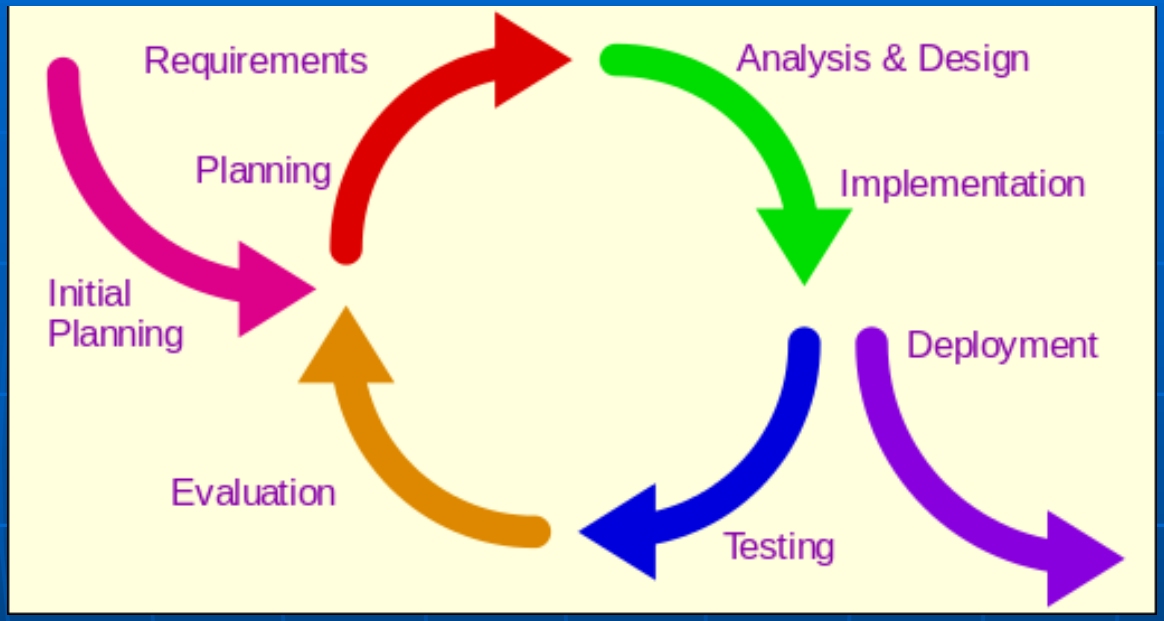
Testiranje manjih delova (modula) je pogodan način za kontrolu kvaliteta i pronalaženje grešaka što je ranije moguće.

Testiranje sistema kao celine vrši se da se proveri da li sistem radi na ciljnim platformama, i da li njegovo ponašanje odgovara zathevima koji su postavljeni na početku projekta.

Održavanje

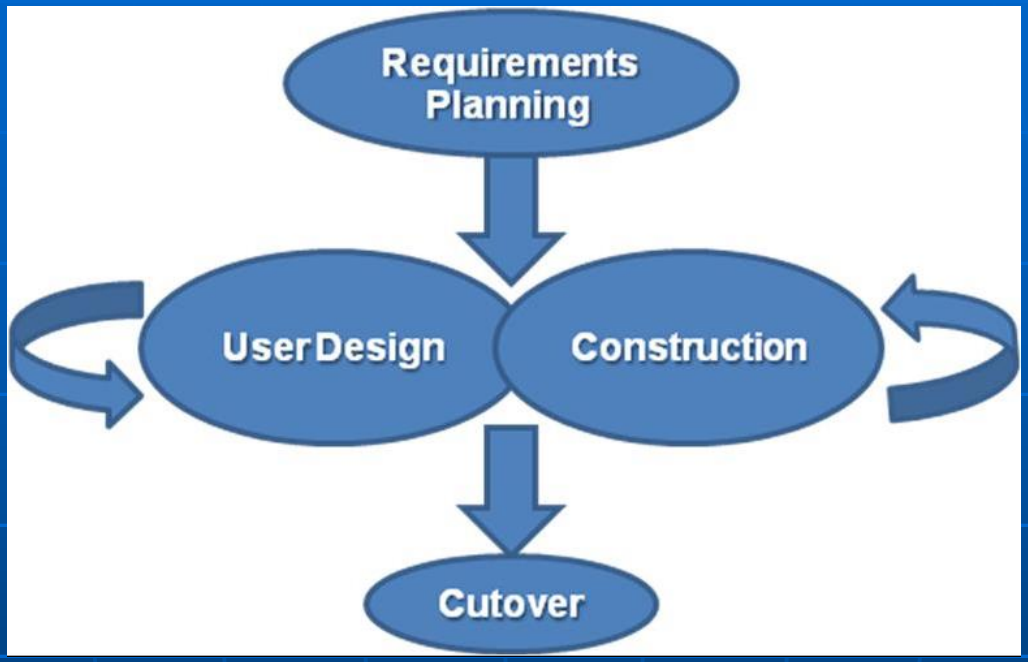
- Od trenutka kada je softverski sistem isporučen korisnicima počinje i potreba za njegovim održavanjem. Mogu se pojaviti grešeke prouzrokovane pogrešno unetim podacima od strane korisnika (takve podatke uvrstiti u plan testiranja), ili zbog neočekivanog i/ili nepravilnog korišćenja softvera (takve slučajeve uvrstiti u dokumentaciju).
- Korisnici, takođe, mogu zahtevati i dodatne funkcije koje nisu uključene u tekuću reviziju softvera, mogu tražiti da softver radi brže, ili čak postaviti pred razvojni tim i veće probleme.
- Proces razvoja softvera mora biti prilagođen za promene koje takđe moraju proći kroz sve gore navedene faze.

Iterativni Razvoj (Iterative Development)



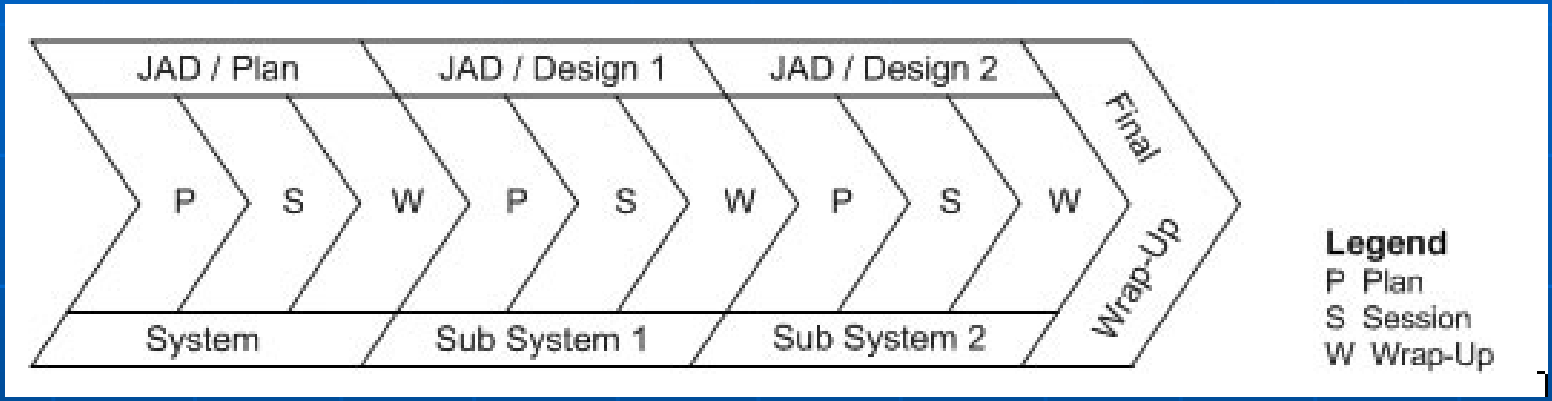
- Ovde se propisuje inicijalna konstrukcija malog (pa sve većeg) dela softverskog projekta kojom se pomaže da svi oni koji su uključeni u razvoj otkriju probleme i pitanja pre nego što oni postanu suviše ozbiljni.
- Iterativni procesi su pogodni za komercijalni pristup razvoju jer omogućavaju da se zadovolje potrebe budućih korsnika softvera koji ne znaju da definišu šta im je potrebno.
- Iterativne metode uključuju i druge ideje kao što su agilni softver i ekstremno programiranje, o čemu će biti reči kasnije.

Brzi razvoj aplikacija (Rapid Application Development - RAD):



- Ovaj metod baziran je na konceptu da se softverski proizvod može dobiti brže i bolje organizovanjem radionica (workshops) interesnih grupa i na taj način generišu softverski zahtevi. Podrazumeva razvoj prototipovima:
  - Smanjenje rizika. Prototipovi bi mogli da testiraju neke od najtežih mogućih delova sistema u ranoj fazi životnog ciklusa
  - Korisnici su bolji u korišćenju i reagovanju u odnosu na stvaranje specifikacije.
  - Prototipovi mogu biti korisni i mogu se razvijati u završene proizvode.

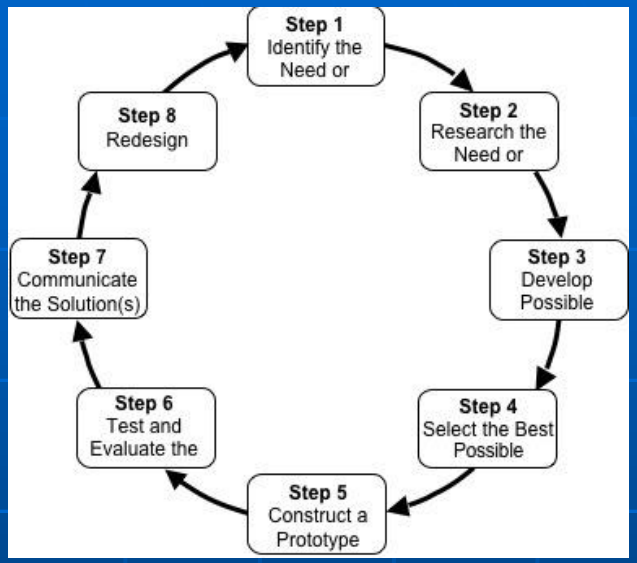
Zajednički razvoj (Joint application development - JAD)

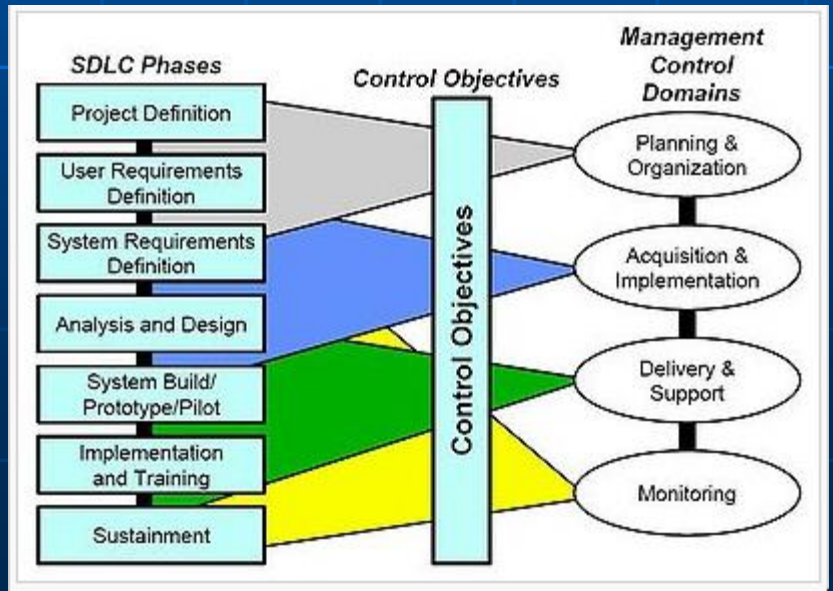


- Ovaj model podrazumeva uključivanje korisnika u dizajn i razvoj aplikacije kroz seriju kolaborativnih vorkšopova koji se nazivaju JAD sesijama.
- Jad koristi sudelovanje korisnika i grupnu dinamiku da tačno opišu korisničke poslovne potrebe i da zajednički razvijaju rešenje.

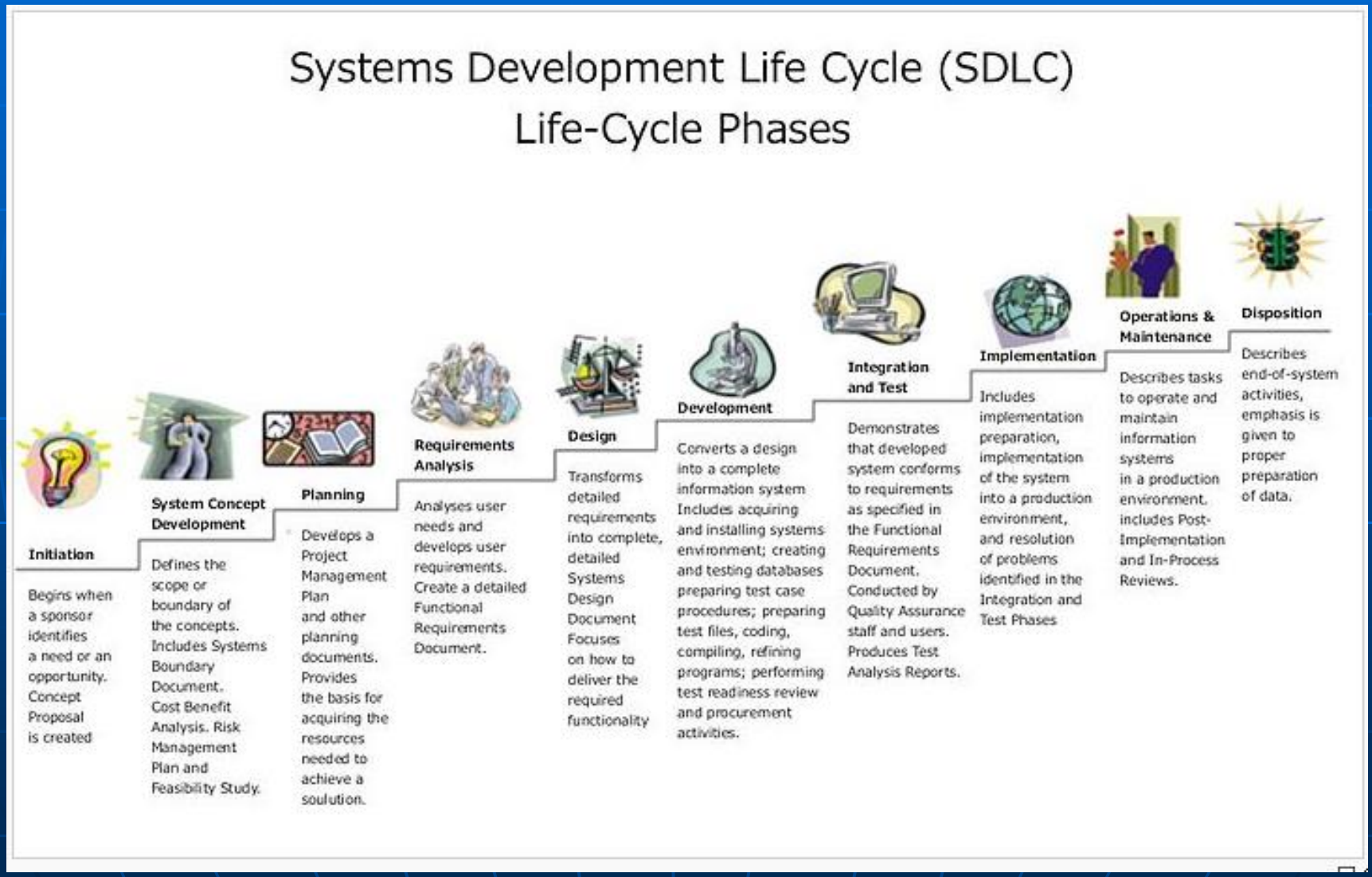
Prototipski razvoj (Prototyping Model - SDLC):

- U ovom metodu se najpre uradi prototip sistema koji predstavlja samo jednu ranu aproksimaciju finalnog sistema.
- Zatim se prototip testira i unapređuje sve do nivoa konačno prihvatljivog prototipa na osnovu kojeg se onda razvija željeni softverski proizvod.





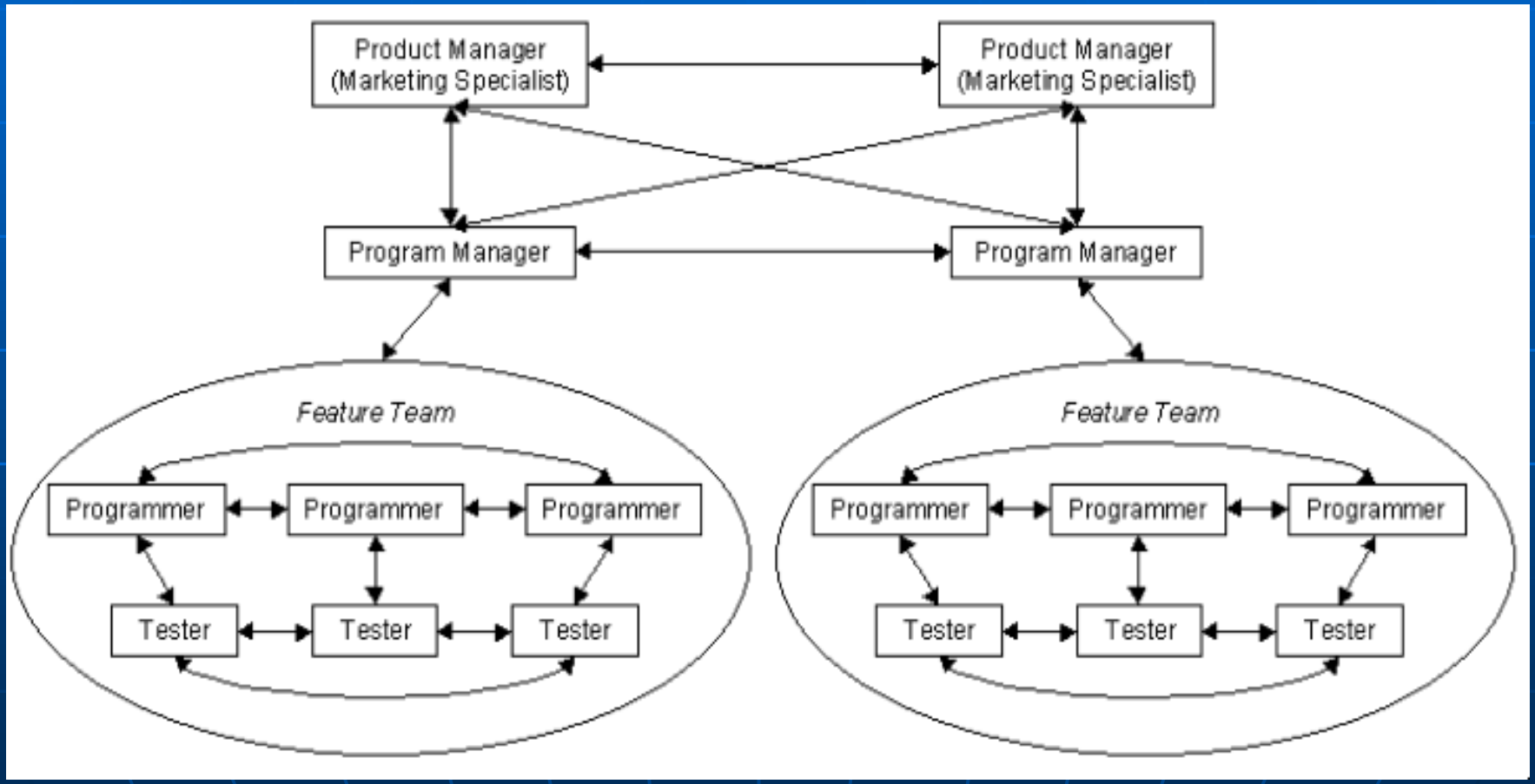
Prototipski razvoj (Prototyping Model - SDLC):



Sinhronizuj i stabilizuj (Synchronize-and-Stabilize):

- Ovaj model se zasniva na timovima koji rade paralelno na individualnim modulima zajedničke aplikacije, povremeno (često) sinhronizuju svoj kod sa kodovima ostalih timova do postizanja stabilnog sitema, a zatim nastavljaju dalji razvoj.
- Ovaj model razlaže velike projekte u male segmente, koje timovi sa srodnim veštinama mogu da obrade i završe efikasno.
- Ovaj metod se generalno koristi u Majkrosoftu i predstavlja osnovu Majkrosoftove metodologije poznate kao
- "Microsoft Solution Framework" (MSF).

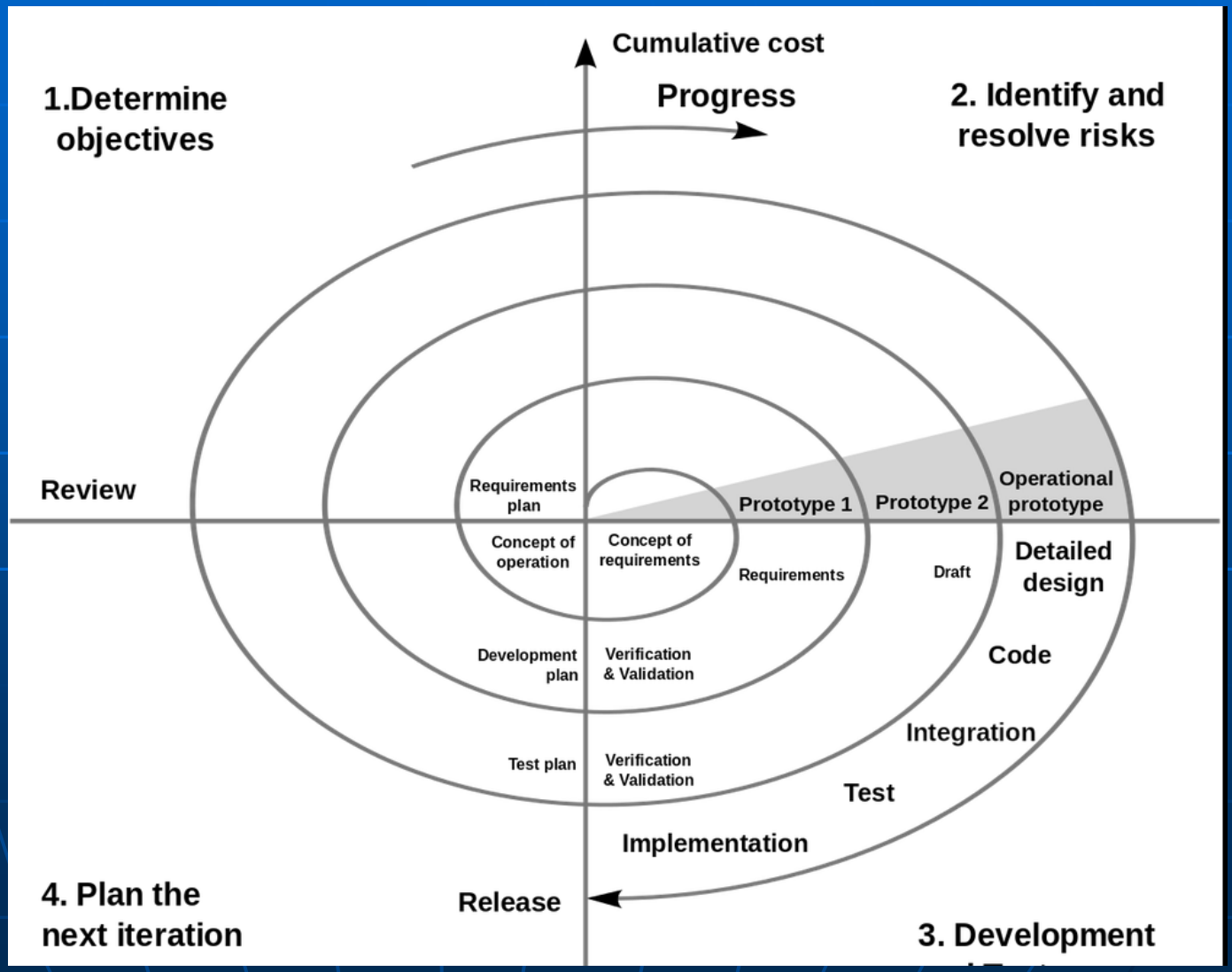
Sinhronizuj i stabilizuj (Synchronize-and-Stabilize):



Spiralni razvoj (Spiral Model):

Ovaj model razvoja softvera kombinuje vodopad i prototip modele.

On je pogodan za velike, skupe i komplikovane projekte.

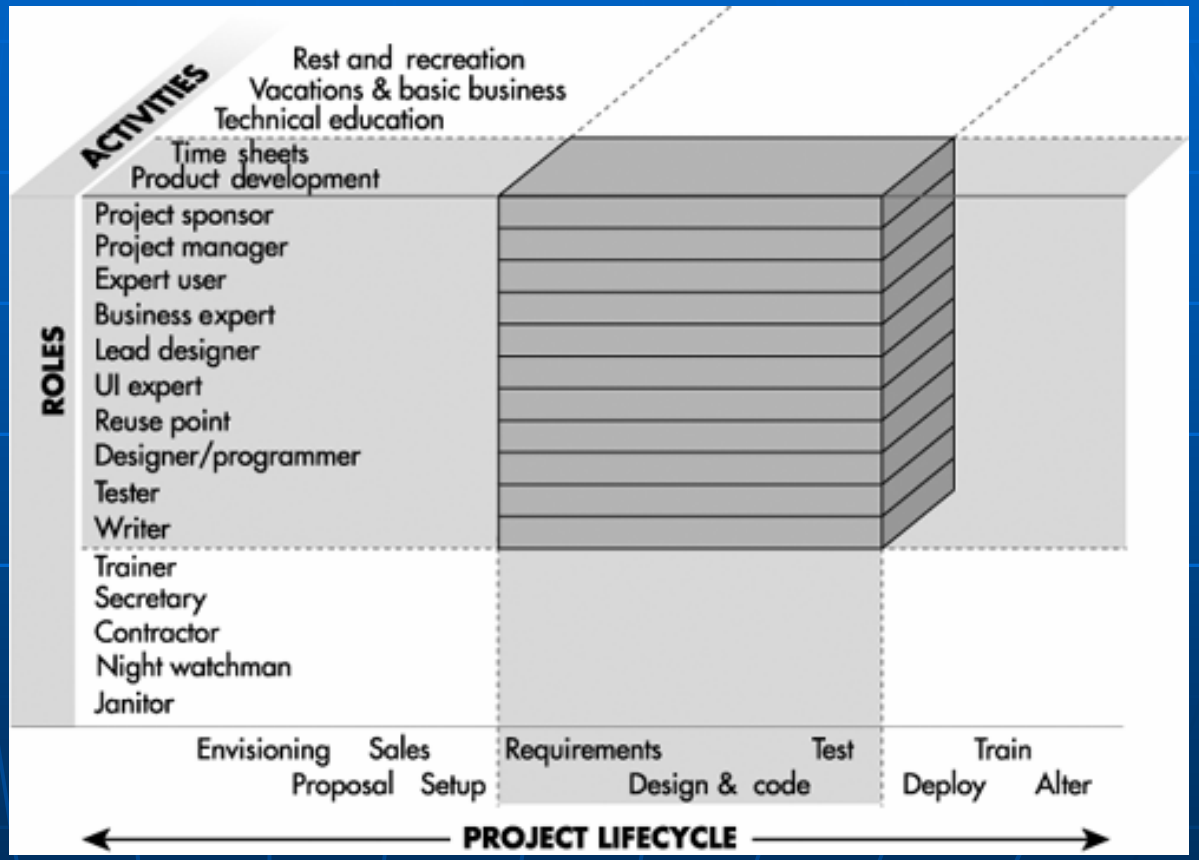


Organizija rada na softverskih projektima

- Razvoj i proizvodnja softvera zahteva visko obrazovane stručnjake raznih profila, najsavremenija sredstva rada kao i dobru organizaciju rada.
- Analiziraćemo aktivnsoti i uloge članova softverskog tima, organizaciju timova i potrebne uslove rada.

Aktivnosti i uloge softverskih timova

- Sledeća slika pokazuje kako se aktivnosti i uloge softverskih timova dovode u vezu sa životnim ciklusom razvoja softvera.
- Liste aktivnosti i uloga su samo ilustrativne i naravno ne iscrpljuju sve slučajeve.



Ativnosti

- **Razvoj proizvoda** (Product Development):
  Ovo je centralna aktivnsot u softverskim projektima. Obuhvata već pomenute procese analize, dizajna, kodiranja, testiranja, održavanja, pisanja tehničke dokumentacije i sl.

- **Tehnička edukacija** (Technical Education):
  Informacione tehnologije se veoma brzo razvijaju. Praćenje tako brzog razvoja zahteva poseban tretman u obuci i treningu članova softverskog tima.

  Specijalistički kursevi, seminari, konferencije, vorkšopovi su samo neki od oblika permanentnog obrazovanja neophodnog da softverski stručnjaci održe visok nivo svoje profesionalnosti.

Ativnosti

- **Odmor i rekreacija** (Rest and Recreation)

  Rad na softveru zahteva izuzetno naporno mentalno angažovanje. Borba sa vremenom za što raniji izlazak na tržište sa proizvodom koji se razvija može da bude veoma stresna.

  Zato softverske kuće pokušavaju da ove izuzetne napore svojih zaposlenih kompenziraju raznim oblicima komfornih radnih okruženja, sportskih i rekreacionih aktivnosti.

  Nije čudo što je u Kaliforniji sedište najvećih softverskih kompanija (Google, na primer) jer je Kalifornija poznata kao deo USA sa izuzetnom ponudom za rekreaciju i razonodu.

Uloge

- **Dizajneri/Programeri:**

  Ovo je suštinska uloga svakog razvoja softvera. To su kreatori sistema. Oni definišu arhitekturu sistema, vrše izbor platformi, alata, metoda, algoritama, proizvode kod (source code). Može postojati više nivoa prema iskustvu (seniori, juniori, početnici, itd.) ili prema oblastima (GUI, biznis logika, baze podataka, itd).

- **Testeri:**

  Testeri su veoma važna uloga kojom se postiže više efekata. Pored kontrole kvaliteta softvera koji se razvija, testeri mogu da pomognu i u podizanju ukupne produktivnsoti tima, da ukažu ne samo na reške u funkcionisanju već i da, svojim primedbama, pomognu dizajn korisničkog intefejsa, i/ili ukupne funkcionalnosti
- sistema.

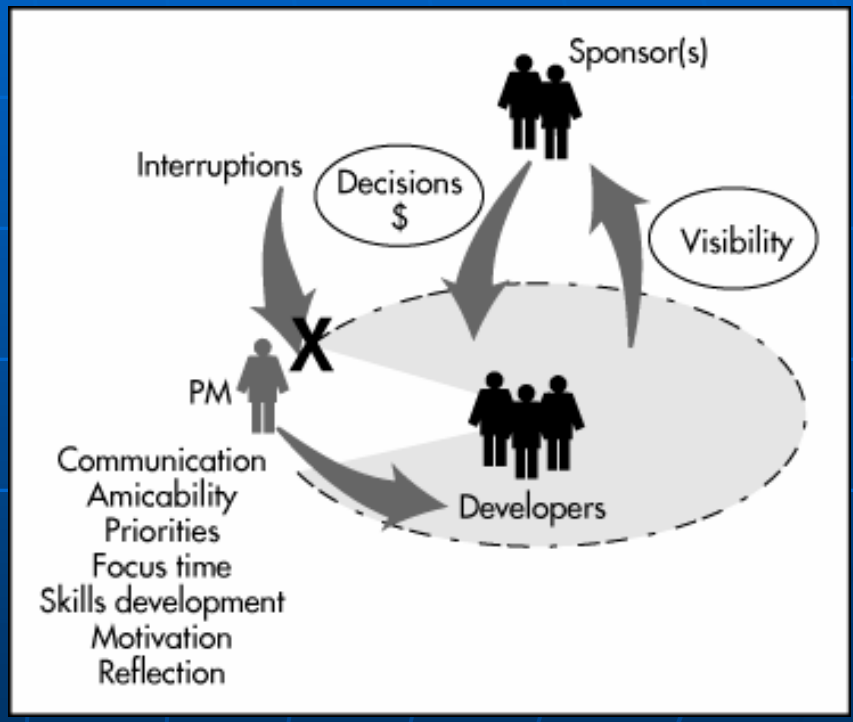
Uloge

- **Pisci:**

  Pisci su oni koji proizvode hiljade stranica teksta kojim se opisuje proizvod, od prvog dokumeta kojim se proizvod najavljuje (“white paper”), pa sve do opisa sistema, uputstava za korišćenje, marketinških prezentacija i reklamiranja.

- **Menadžer projekta** (Project Manager):

  To je osoba koja je najviše odgovorna za uspeh projekta. On planira aktivnosti, prati izvršenje plana, komunicira sa svim zaintersovanim stranama u projektu (članovima softverskog tima, menadžementu firme, korsnicima, itd.). Sldeća slika ilustruje rad menadžera projekta.



Uloge

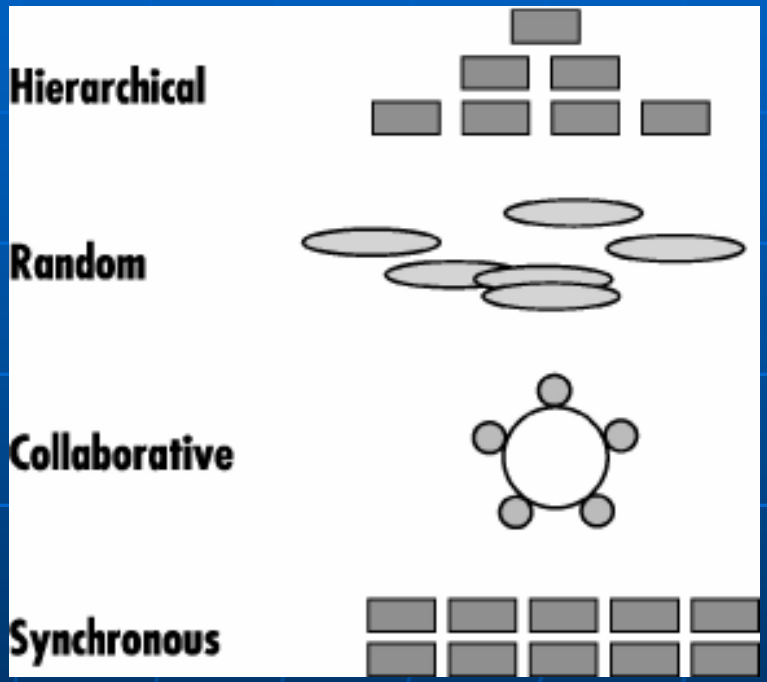
- **Sponzor projekta** (Project Sponsor)

  To je osoba koja predstavlja "spiritus movens" projekta. Projekat je "živ" sve dok su članovima tima na raspolaganju resursi (novac, ljudi i oprema) potrebni za njegovu realizaciju.

  Sponzor predstavlja sponu projektnog tima i finansijera projekta.

- **Timovi**

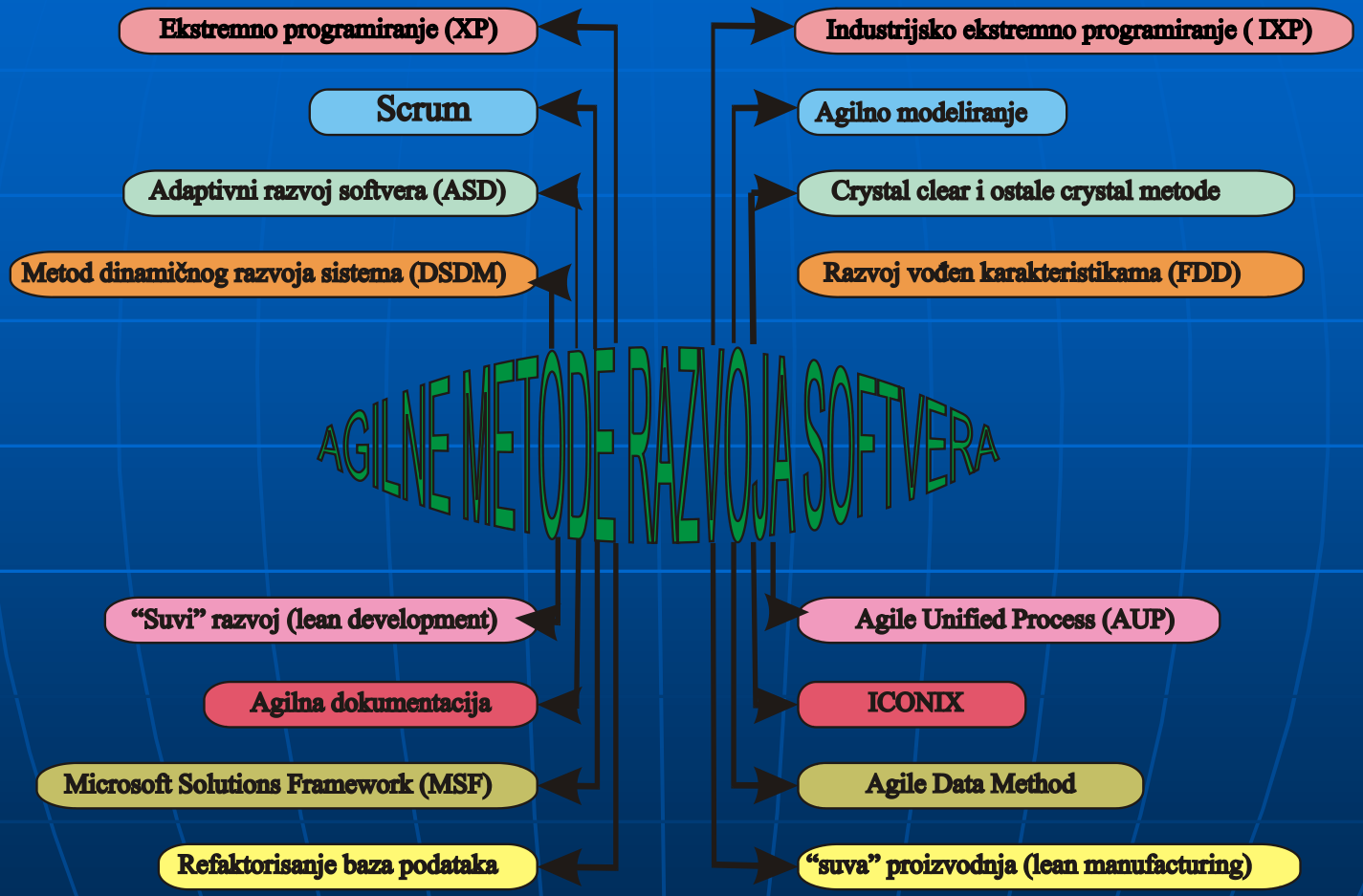
  Softverski timovi mogu biti organizovani na više načina. Sledeća slika prikazuje tipične organizacije:



METODE AGILNOG RAZVOJA SOFTVERA

- Kasnih devedesetih godina prošlog veka, grupa naučnika (Agile Alliance 2001) je pokušala da promeni do tada čvrsto ustaljeni način i disciplinu na koji se softver osmišljava, dokumentuje, razvija i testira. Oni su, naglašavajući ulogu koju fleksibilnost može da odigra u spretnom i brzom razvoju softvera, formulisali sopstvene principe ogledavajući ih u takozvanom ” agilnom manifestu ” :
  - Osobe i interakcije umesto procesa i alata
  - Softver koji radi umesto opširne dokumentacije
  - Zajednički rad s kupcem umesto pregovaranja preko ugovora
  - Reagovanje na promene umesto striktnog pridržavanja plana
- Na početku razvoja ove metode su se nazivale lakim metodama, da bi tek kasnije dobile ime koje i danas poseduju – AGILNE METODE RAZVOJA SOFTVERA

METODE AGILNOG RAZVOJA SOFTVERA



METODE AGILNOG RAZVOJA SOFTVERA

- Agilne metode u razvoju softvera javljaju se kao odgovor na opsežne (tradicionalne) metodologije. Tako iz sledeće tabele možemo utvrditi osnovne razlike između ”starog ” načina razvoja softvera, npr. poznatog tipa vodopada (Waterfall) i agilnih metoda.

Waterfall	Agilne metode
Puno dokumentacije	Verbalna komunikacija
Planiranje unapred	Postepeno planiranje Evolutivni dizajn
Mala fleksibilnost	Velika povratna sprega
Isporuka u jednom trenutku	Fleksibilnost isporuke
Kontrola	Automatizovano testiranje

METODE AGILNOG RAZVOJA SOFTVERA

- Agilne metode, označavaju spremnost na pokret, aktivnost, žestinu, brzinu razvoja softvera pokušavajući da ponude odgovor na želju klijenata za manje robusnim metodologijama razvoja softvera, koje sa sobom donose brže, žešće i kvalitetnije procese razvoja.
- Agilne metode razvoja softvera u softverskom inženjerstvu su manje opsežne metode prihvatajući činjenicu da je softver teško kontrolisati. Zato su, inženjeri koji razvijaju softver fokusirani na male jedinice posla, minimizirajući rizik za pravljenje grešaka. To je naročito važno sa današnjim rapidnim rastom industrije vezane za telekomunikacije, Internet i sa okolinom vezanom za mobilne aplikacije sa distribuiranim razvojem.

METODE AGILNOG RAZVOJA SOFTVERA - karakteristike:

- Mogu se definisati sledeće karakteristike agilnih softverskih procesa sa aspekta brze isporuke, koje omogućuju skraćivanje životnog ciklusa softverskih projekata:

  - ✓ modularnost stepena razvojnog procesa
  - ✓ kratke iteracije koje omogućavaju brze verifikacije i korekcije
  - ✓ vremenski okvir iteracija je od jedne do šest nedelja
  - ✓ uklanjanje svih nepotrebnih aktivnosti
  - ✓ prilagodljivost s mogućim nenadanim rizicima
  - ✓ inkrementalni pristup procesu koji omogućava funkcionalnu izgradnju softverskog produkta u malim koracima
  - ✓ konvergentni i inkrementalni pristup minimizira rizike
  - ✓ ljudski-orijentisani agilni proces favorizuje ljude iznad procesa i tehnologija
  - ✓ kolaborativni i komunikativni radni stil.

Ekstremno programiranje (XP)

- Ekstremno programiranje se počelo koristiti u prvoj polovini devedesetih godina prošlog veka sa vrlo disciplinovanim pristupom razvoju softvera.
- Ekstremno programiranje je metodologija softverskog inženjerstva, zasnovana na vrednostima povratne sprege, komunikacije, jednostavnosti, odvažnosti i poslovanja.
- Pokretač ove nove metodologije je bio *Kent Beck,* koristeći nove koncepte razvoja softvera. Rezultat je bio metodologija ekstremnog programiranja razvoja softvera koja primenjuje skup 12 veština koje se mogu izdvojiti u 4 oblasti:

Ekstremno programiranje (XP)

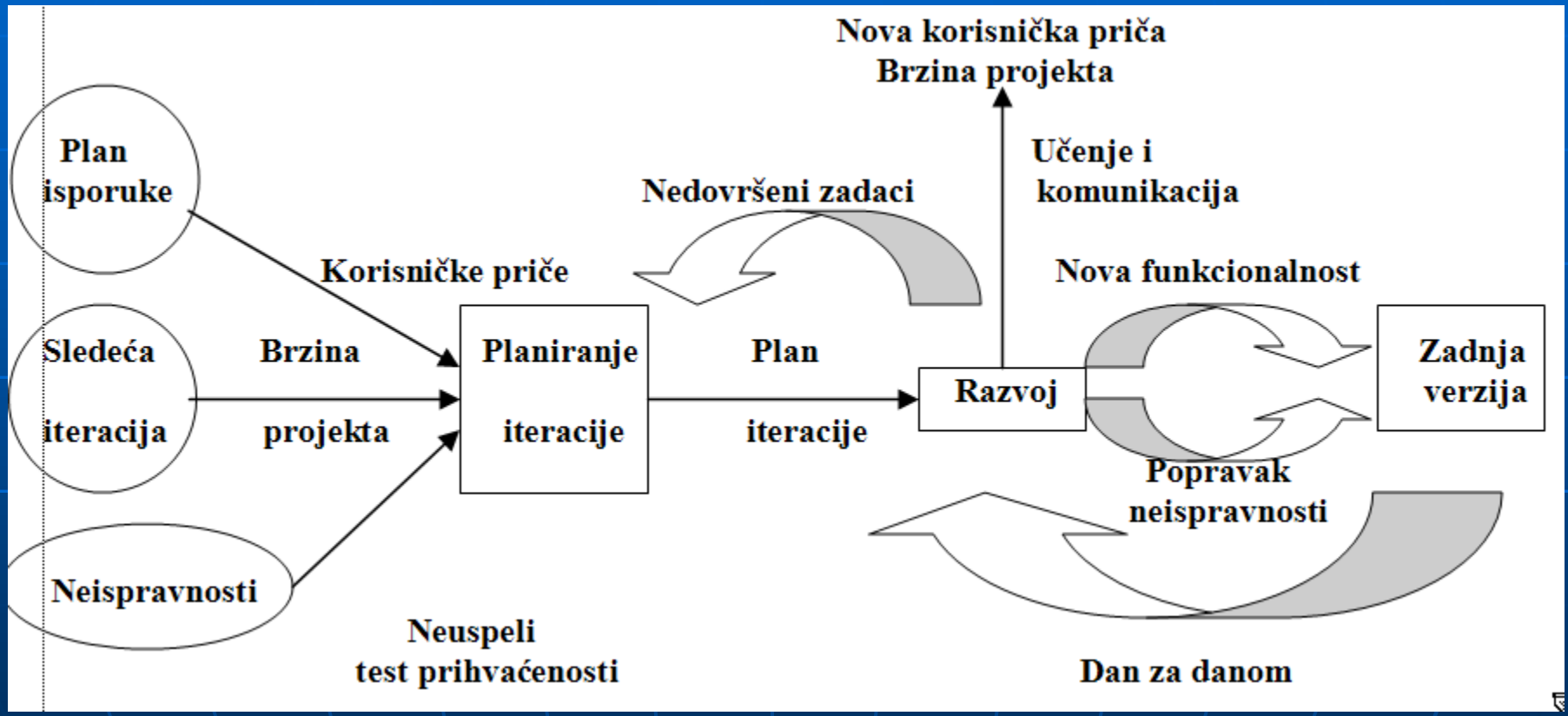
- **POVRATNA SPREGA**
  - ✓ Programiranje u paru (engl. *Pair Programming*). Ovaj princip znači da uvek dva čoveka pišu određeni kod.
  - ✓ Igra planiranja (engl. *Planning game*). Korisnička interakcija u programerskom (implementacionom) timu između programera i korisnika oko procena implementacije pojedinih funkcionalnosti projekta.
  - ✓ Testiranje (engl. *Testing*). Kontinualno, često ponavljajuće automatizovano testiranje jedinice (engl. *unit*) i regresiono (engl. *regression*) testiranje.

- **KONTINUALNOST PROCESA**
  - ✓ Kontinualna integracija (engl. *Continuous integration*). Novi kod se integriše u sistem čim je spreman (implementiran i testiran).
  - ✓ Korišćenje tehnike refaktorisanja (engl. *Refactoring*). Uklanjanje dvostrukog (redundantnog) koda i održavanje koda jednostavnim.
  - ✓ Male česte isporuke (engl. *Small/short releases*). Sistem se brzo i često isporučuje, najmanje svaka 2 do 3 meseca. Ovaj pristup se temelji na praksi iterativnog i inkrementalnog razvoja.
  - ✓ Povratna informacija od kupca (korisnika) (engl. *On-site customer*).Korisnik je stalno na raspolaganju programerima.

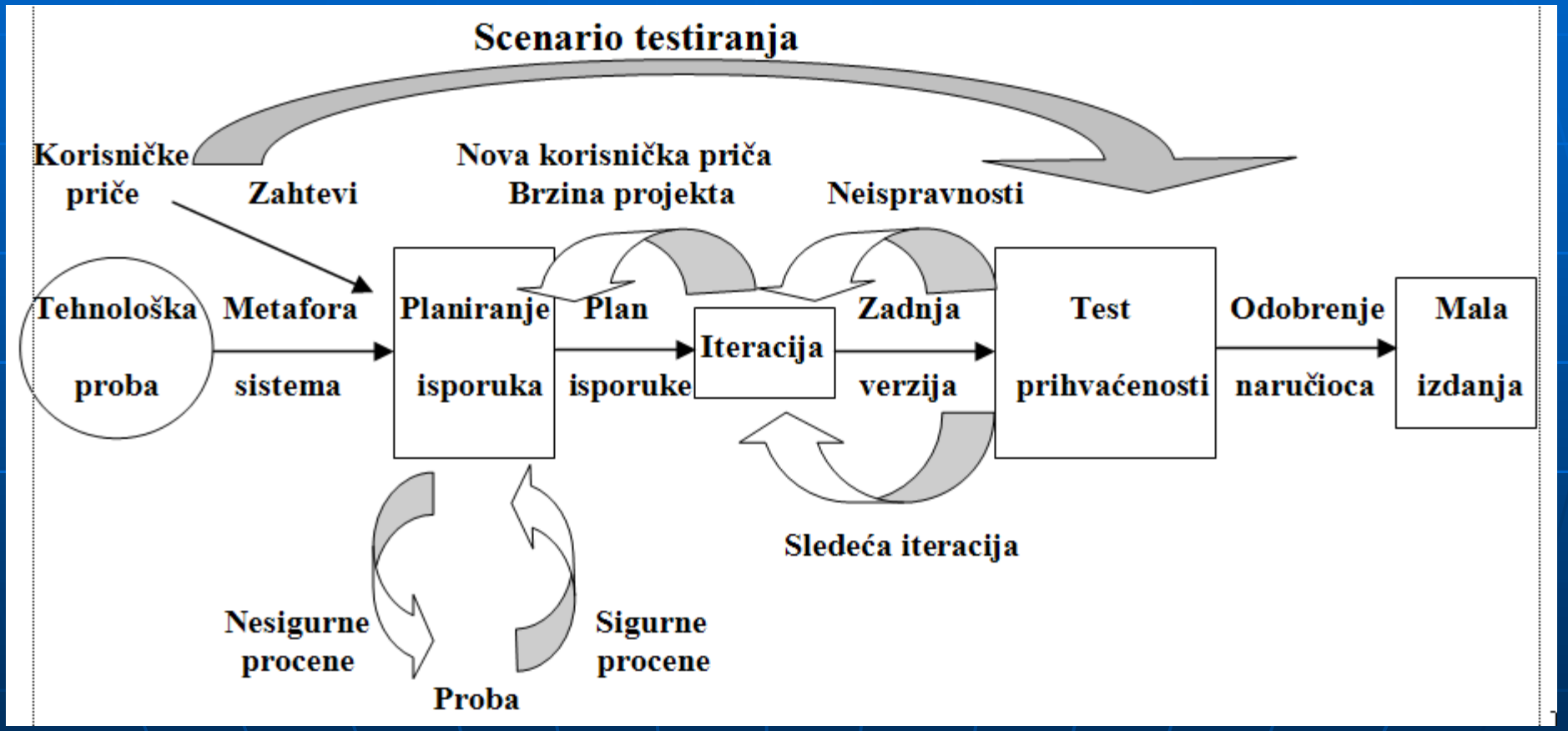
Ekstremno programiranje (XP)

- **DELJENO RAZUMEVANJE**
  - ✓ Standardi kodiranja (engl. *Coding Standards*). Postoje standardi kodiranja i programeri ih slede kako bi kod na kojem se trebaju napraviti bilo kakve izmene, a koga je pisao neko drugi u timu, bio razumljiviji.
  - ✓ Zajedničko deljenje koda (pristup kodu) (engl. *Collective Ownership*). Bilo ko iz tima sme mijenjati bilo čiji kod.
  - ✓ Jednostavan dizajn (engl. *Simple Design*). Naglasak je na dizajnu najjednostavnijeg rešenja koji je zahtevan u tom trenutku, bez dodatnog koda i viška funkcionalnosti.
  - ✓ Organizacija sistema sa metaforama (engl. *Methaphor*). Metafora je pojednostavljena slika sistema u razvoju.
- **DOBROBIT PROGRAMERA**
  - ✓ Održivi korak (engl. Sustainable Pace). Četrdeset - satna radna nedelja (engl. *40-hours week*). Maksimum je 40-satna radna nedelja. Nisu poželjni uzastopni prekovremeni dani zbog slamanja timskog duha.

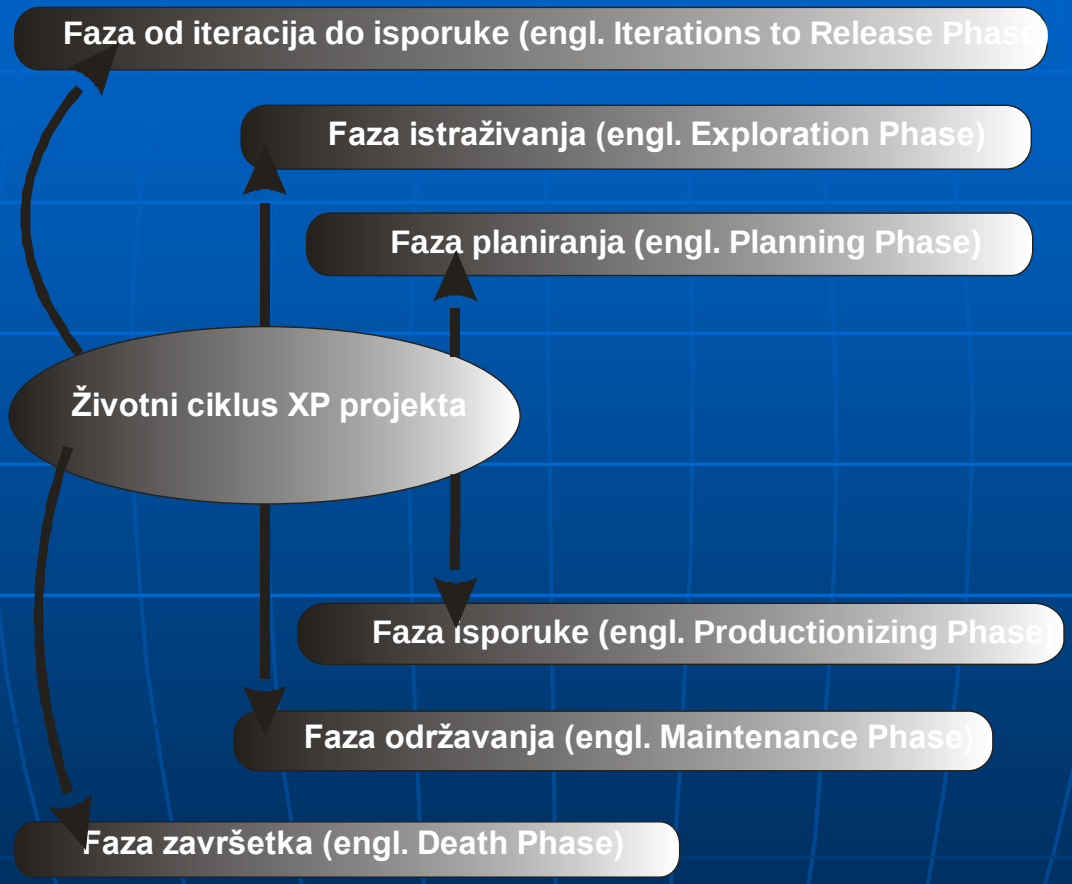
Iterativni postupak u procesu razvoja projekta



Proces razvoja projekta



Životni ciklus XP projekta



Scrum

- Termin *Scrum* (*Schwaber* 1995. godine, *Schwaber* i *Beedle* 2002. godine) se u kontekstu novih principa i teorije programiranja prvi put opisuje, (1986. Hirotaka Takeuchi i Ikujiro Nonaka), kao pristup koji povećava adaptivnost, fleksibilnost, brzinu i samoorganizaciju procesa razvoja softvera.

- Odgovornosti i uloge u procesu razvoja softvera mogu se definisati, slika

Scrum

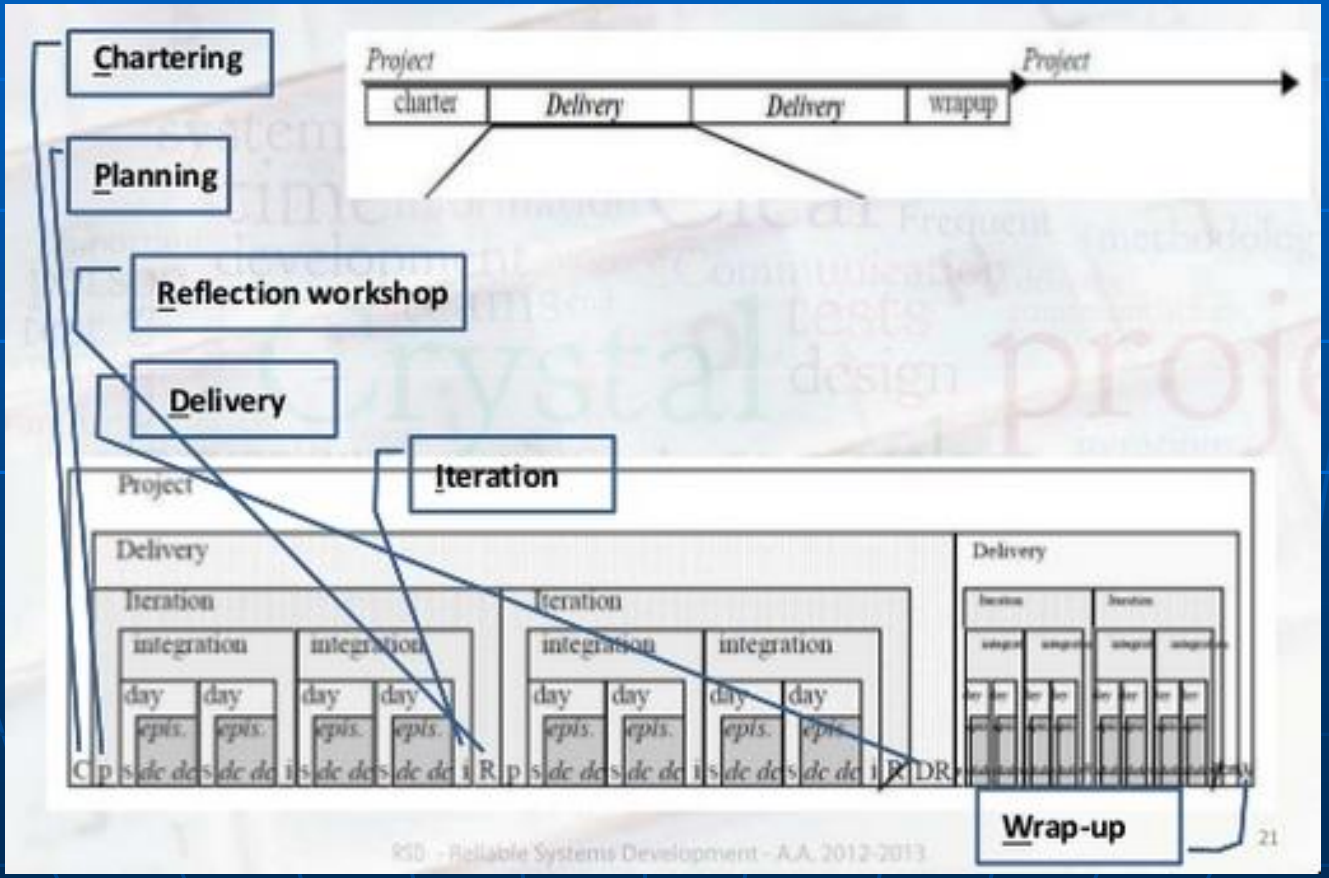
- Može se reći da se Scrum sastoji od:
  - tehnika i alata u cilju izbegavanja haosa koji nastaje zbog kompleksnosti i neizvesnosti u dosadašnjim metodama razvoja softvera,
  - uloga i procedura koje u tri faze:
    - ✓ Pre-game - Pre igre *(planiranje, dizajn / arhitektura visok nivo apstrakcije)*
    - ✓ Development – Razvojna *igra (razvoj, sprintovi – iterativni ciklusi, poboljšanja, nove verzije)*
    - ✓ Post-game - *Posle igre (nema novih zaheva, sistem spreman za produkciju).*

Scrum metodologija

- Omogućava bolje uočavanje i menadžersku kontrolu svih potencijalnih defekata ili poteškoca u razvojnom procesu.

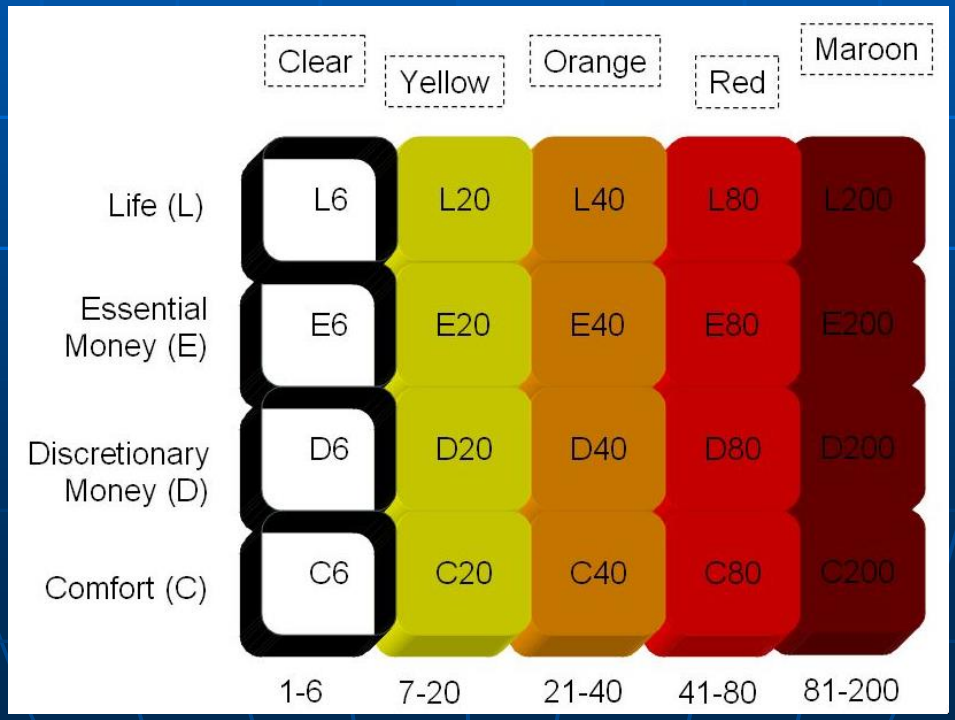
Crystal metodologija

- Crystal familija za razvoj softvera se uvodi kao skup metoda koje su više ili manje primjenjive na neke konkretne razvojne situacije i koja na osnovu povratnih veza najviše odgovara, a onda se i dodatno adaptira da odgovori izazovima konkretne situacije.



Crystal metodologija

- Metodologije u kojima centralno mesto predstavljaju ljudi - *"people-centric methodologies", su* bolje od *"process-centric methodologies",* u kojima centralno mesto zauzimaju procesi.
- One, u okviru Cockburn koncepta, pridružuju različite boje pojedinim metodama, da bi naglasile njihovu adekvatnost pojedinim tipičnim razvojnim situacijama, po principu: što tamnija boja, to "teža" metoda.
- Metodologije imaju svoja specifična imena, nazvana prema geološkim kristalima: čisti (clear), žuti, narandžasti, crveni, kestenjasti (maroon). Najpoznatije metodologije su cystal clear, crystal orange i crystal orange web.



Crystal metodologija

- Učestala isporuka – svakih nekoliko meseci, korisnici upoznati sa međuverzijama, daju povratne informacije.
- Kontinualne povratne informacije – projektni tim raspravlja o projektnim aktivnostima, vrši se validacija projekata sa korisnicima, rasprava o mogućim problemima.
- Stalna komunikacija – mali tim (svi u jednoj sobi), veći timovi lokacijski povezani
- Sigurnost – predstavljena na dva načina
  - sigurna zona - članovi tima komuniciraju i rade bez represije;
  - svi projekti nisu kritično isti
- Fokus – članove tima upoznati sa prioritetnim ciljevima, dati im mogućnost da ih ostvare
- Raspoloživost korisnika – članovima tima omogućiti saradnju sa korisnicima tokom celog projekta
- Automatski testovi i integracija – različiti oblici verifikacije funkcionalnosti projekata.

ALGORITMI
I
PSEUDO KOD

ALGORITAM

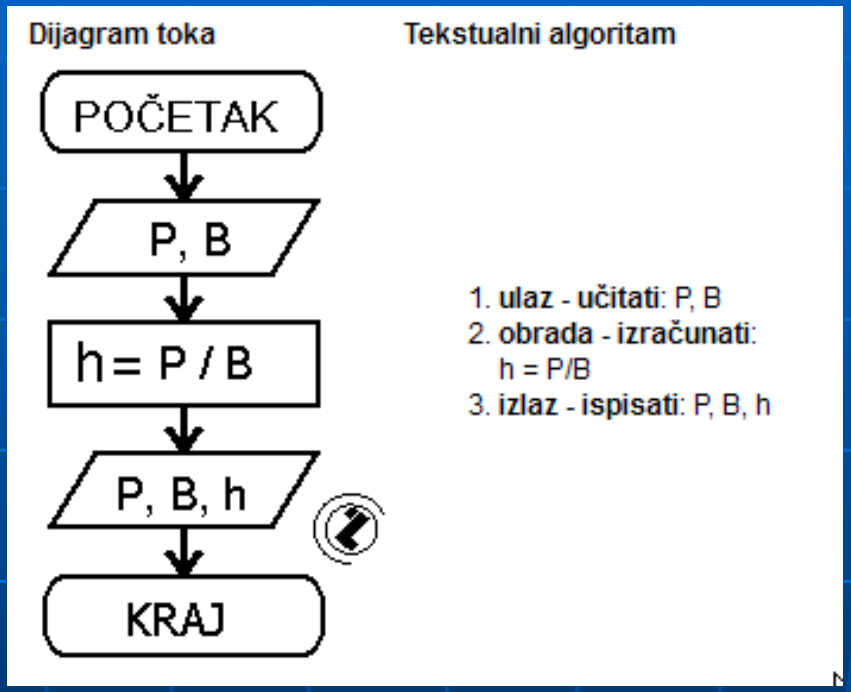
- Riječ "*algoritam*" dolazi od latinskog prevoda imena iranskog matematičara
- Al-Hvarizmija koji se bavio trigonometrijom, astronomijom, zemljopisom, kartografijom, a smatra se ocem algebre jer je definisao osnovna pravila rešavanja linearnih i kvadratnih jednačina. Njegovi radovi su osnova razvoja mnogih matematičkih i prirodnih disciplina, među njima i računarstva.(Daffa,1977)

- *Algoritam* kao postupak za dobijanje rešenja, sastoji se od konačnog niza koraka – tzv. instrukcija, naredbi ili operacija, koje treba izvršiti (izvesti) da bi se dobilo rešenje.

- Svaki algoritam ima sledećih pet bitnih osnovnih svojstava:

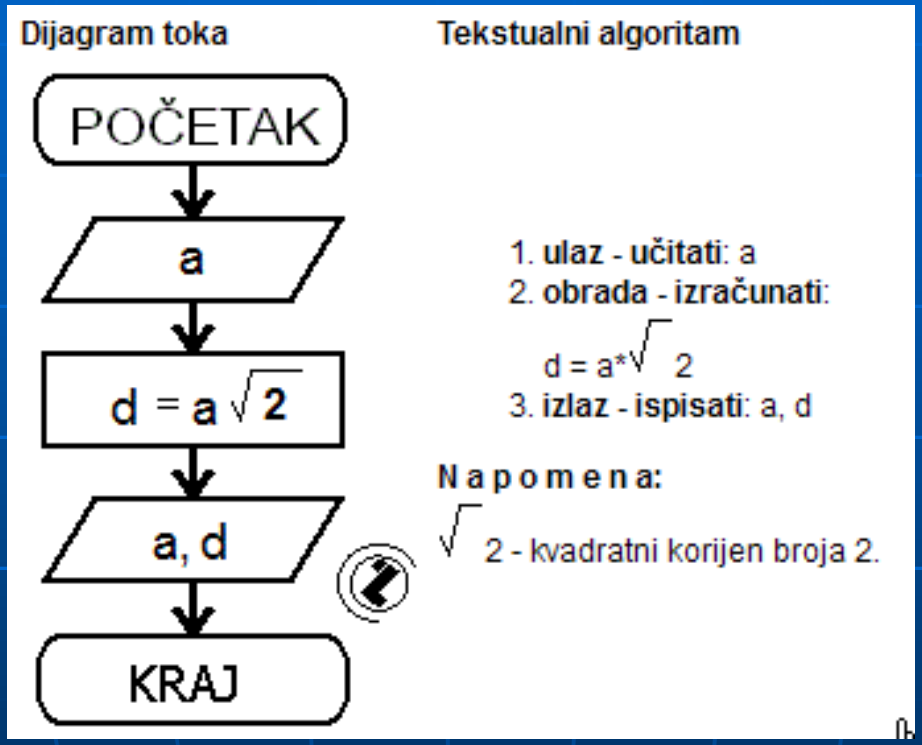
- Ulaz,
- Izlaz,
- Konačnost,
- Definiranost i nedvosmislenost (određenost),
- Efikasnost (efektivnost).

Linijski algoritmi i pseudo kod



Za poznate vrednosti površine i baze valjka izračunati odgovarajuću visinu.

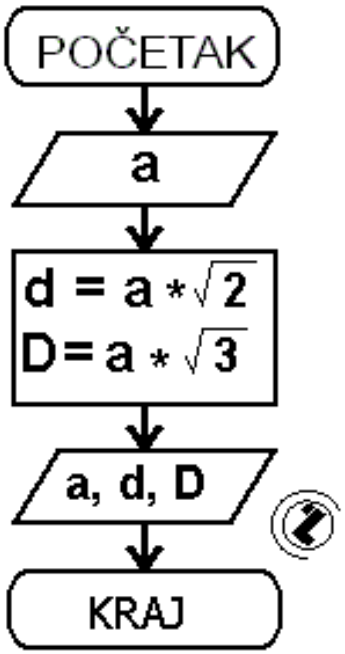
Linijski algoritmi i pseudo kod



Izračunati dijagonalu kvadrata.

Linijski algoritmi i pseudo kod

Dijagram toka



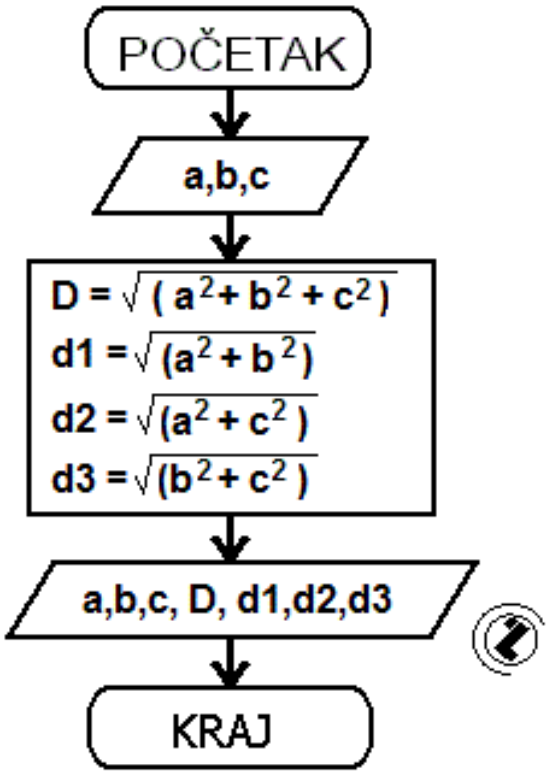
Tekstualni algoritam

1. **ulaz** - **učitati**: a
2. **obrada** - **izračunati**:
   d = a * SQR(2)
   D = a * SQR(3)
3. **izlaz** - **ispisati**: "Dijagonala d je: ",d, "Dijagonala D je: ",D

Izračunati dijalgonale d i D kocke (za poznatu stranicu a).

Linijski algoritmi i pseudo kod

Dijagram toka



Tekstualni algoritam

1. **ulaz - učitati**: "Stranice a,b,c = "; a,b,c
2. **obrada - izračunati**:

   $D = \sqrt{}(a^2 + b^2 + c^2)$

   $d_1 = \sqrt{}(a^2 + b^2)$

   $d_2 = \sqrt{}(a^2 + c^2)$

   $d_3 = \sqrt{}(b^2 + c^2)$
3. **izlaz - ispisati**: "Za stranice a,b,c "; a,b,c
   "Velika dijagonala je "; D ,
   "Dijagonala koju obrazuju stranice a i b je = "; d_1
   "Dijagonala koju obrazuju stranice a i c je = "; d_2
   "Dijagonala koju obrazuju stranice b i c je = "; d_3

N a p o m e n a:

$\sqrt{}(a^2 + b^2 + c^2)$ - kvadratni korijen ($a^2 + b^2 + c^2$).

$\sqrt{}(a^2 + b^2)$ - kvadratni korijen ($a^2 + b^2$).

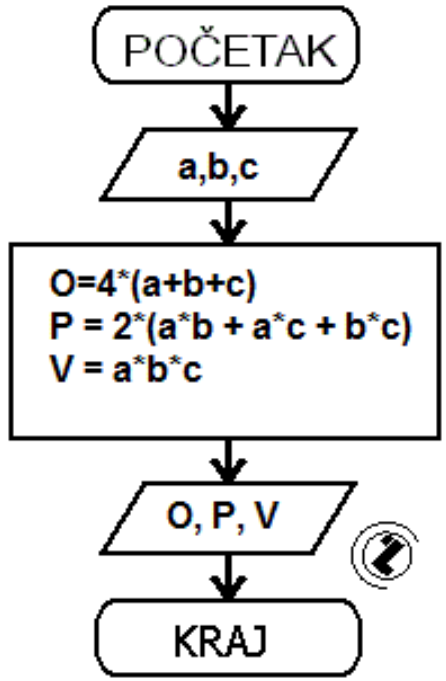
$\sqrt{}(a^2 + c^2)$ - kvadratni korijen ($a^2 + c^2$).

$\sqrt{}(b^2 + c^2)$ - kvadratni korijen ($b^2 + c^2$).

Izračunati dijagonale d i D kvadra (za poznate stranice a, b, c).

Linijski algoritmi i pseudo kod

Dijagram toka

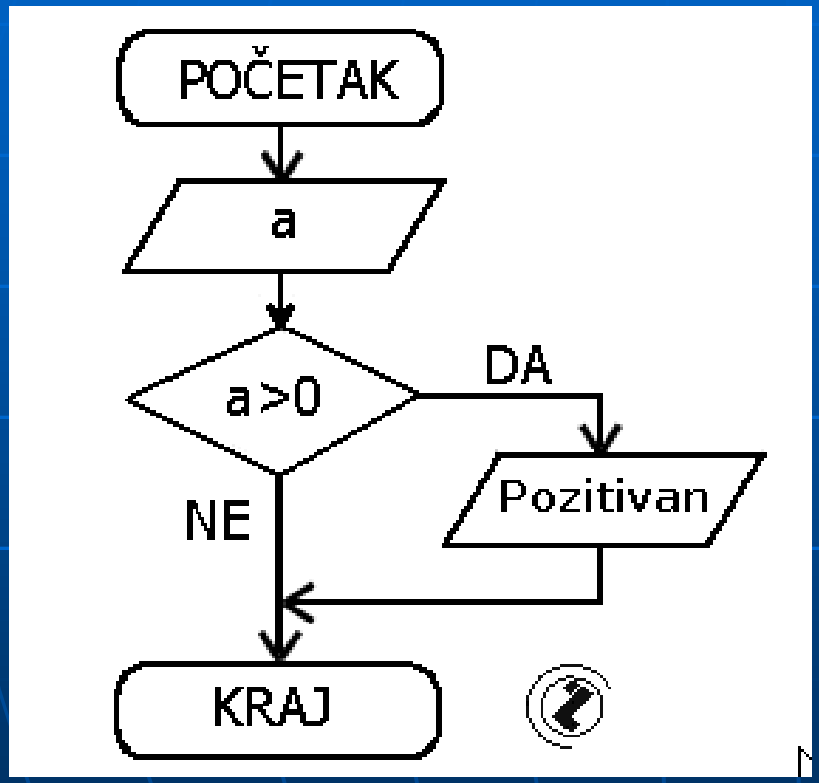


Tekstualni algoritam

1. **ulaz** - **učitati**: "UNESI a, b, c: ", a,b,c
2. **obrada** - **izračunati**:
   O=4*(a+b+c)
   P = 2*(a*b + a*c + b*c)
   V = a*b*c
3. **izlaz** - **ispisati**: "O = ";O, "P = ";P, "V = ";V

Izračunati O, P i V kvadra (za poznate stranice a, b, c)

Razgranata algoritamska šema - IF THEN i pseudo kod

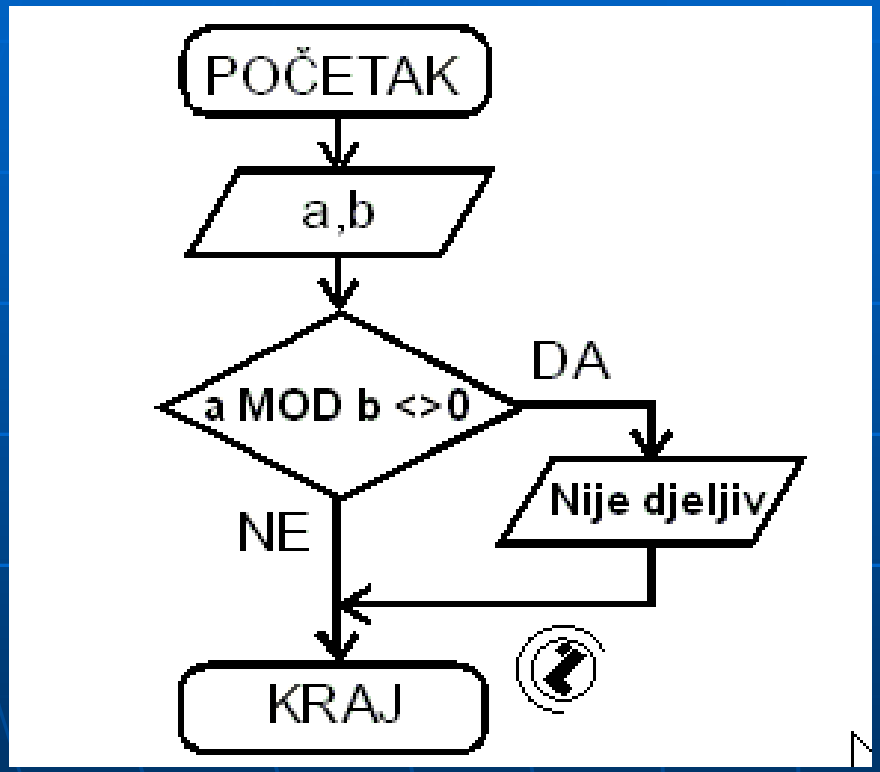


Opis rešenja:

- Sa tastature se upisuje vrednost varijable a.
- Logičkim izrazom a>0 u naredbi IF izvodi se poredjenje da li je upisana vrednost pozitivna.
- Ako je logički izraz istinit na ekranu se ispisuje "POZITIVAN",
- inače za ostale slučajeve nema ispisa.
- Ovo je provera da li je broj pozitivan tj. veći od nule.

Učitati broj a. Ako je učitani broj a veći od nule napisati "POZITIVAN"

Razgranata algoritamska šema - IF THEN i pseudo kod

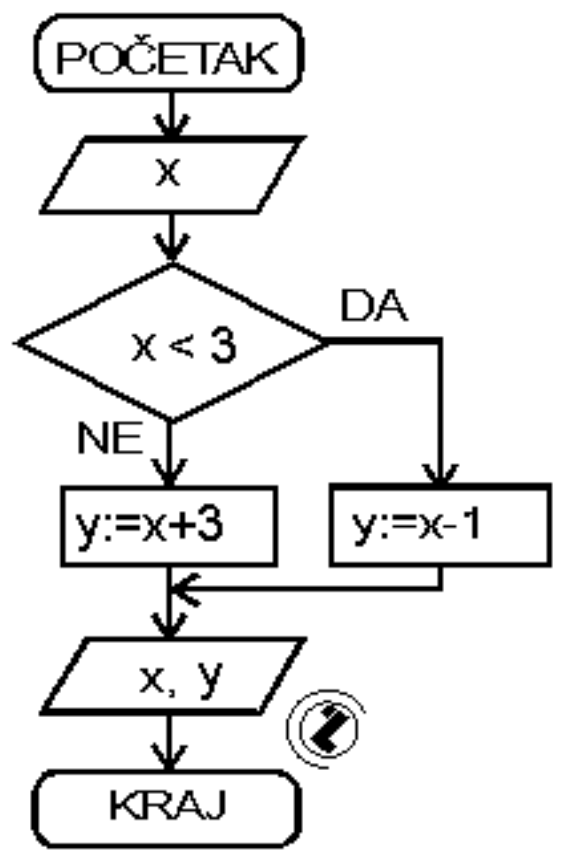


Opis rešenja:

- Upisati dve vrednosti (a, b).
- Provera da li je prvo upisani broj (a) deljiv sa drugim se izvodi sa logičkim izrazom
  a MOD b =0 u naredbi IF.
- Ako je logički izraz istinit na ekranu se ispisuje DELJIV, inače nema ispisa.
- Logički izraz za proveru deljivosti (a MOD b = 0) je istinit ako je a djeljivo sa b i tada se na ekranu ispisuje DJELJIV. Inače za ostale slućajeve nema ispisa.
- Naredbom a MOD b se izračunava ostatak dijeljenja broja a sa brojem b. Ostatak dijeljenja je jednak 0 ako je broj a djeljiv sa brojem b.

Proveriti da li je od dva učitana broja prvi deljiv sa drugim.

Razgranata algoritamska šema - IF THEN i pseudo kod



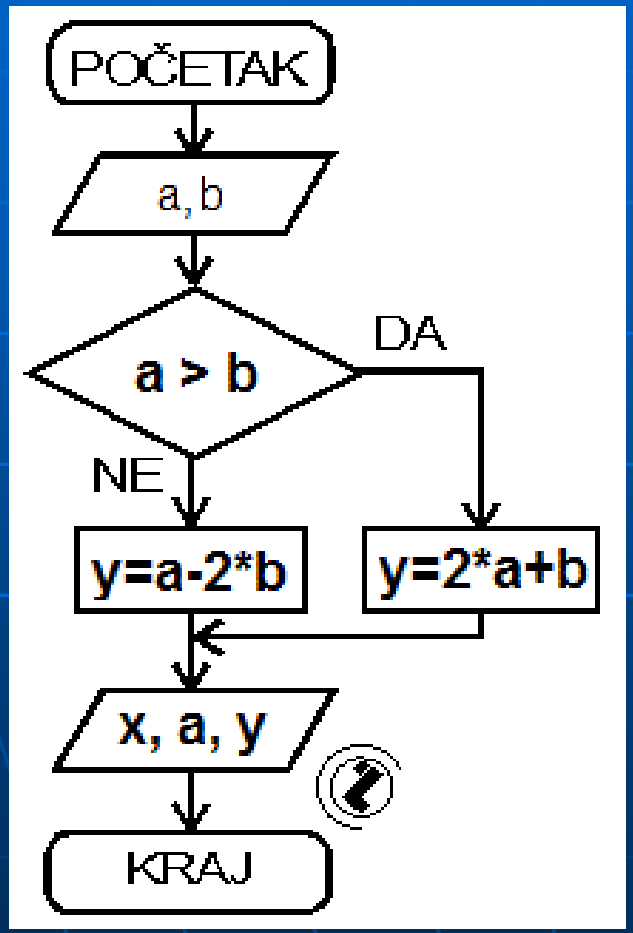
Napisati program za odredjivanje y po formuli:

$$y = \begin{cases} x - 1, & x<3 \\ x + 3, & \text{inače} \end{cases}$$

1. **Ulaz**: x
2. **Logički uslov**: X < 3
   Istinit /TRUE/: Y = X - 1
   Lažan /FALSE/: Y = X + 3
3. **Izlaz**: x, y

Učitati x, Ako je x manje od 3, tada je y=x-1, inače je y=x+3.

Razgranata algoritamska šema - IF THEN i pseudo kod



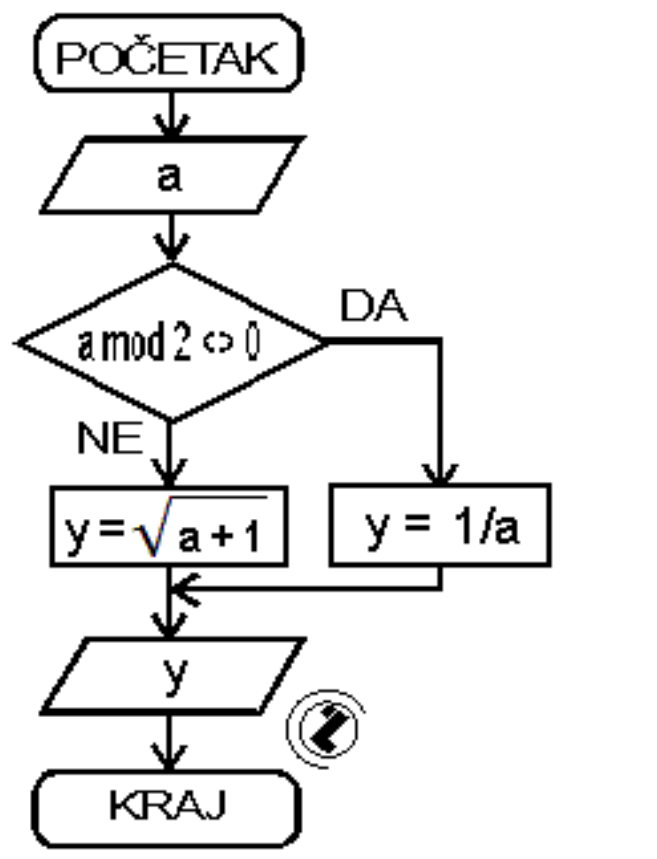
- Napisati program za odredjivanje y po formuli:

$$y = \begin{cases} 2*a+b, & a>b \\ a-2*b, & a<=b \end{cases}$$

- **Ulaz**: a, b
- **Logički uslov**: a > b
- **Istinit /TRUE/:**
- y = 2 * a + b
- **Lažan /FALSE/:**
- y = a - 2 * b
- **Izlaz**: a, b, y

Učitati a i b, ako je a veće od b, tada je y=2*a+b, inače je y=a-2b.

Razgranata algoritamska šema - IF THEN i pseudo kod



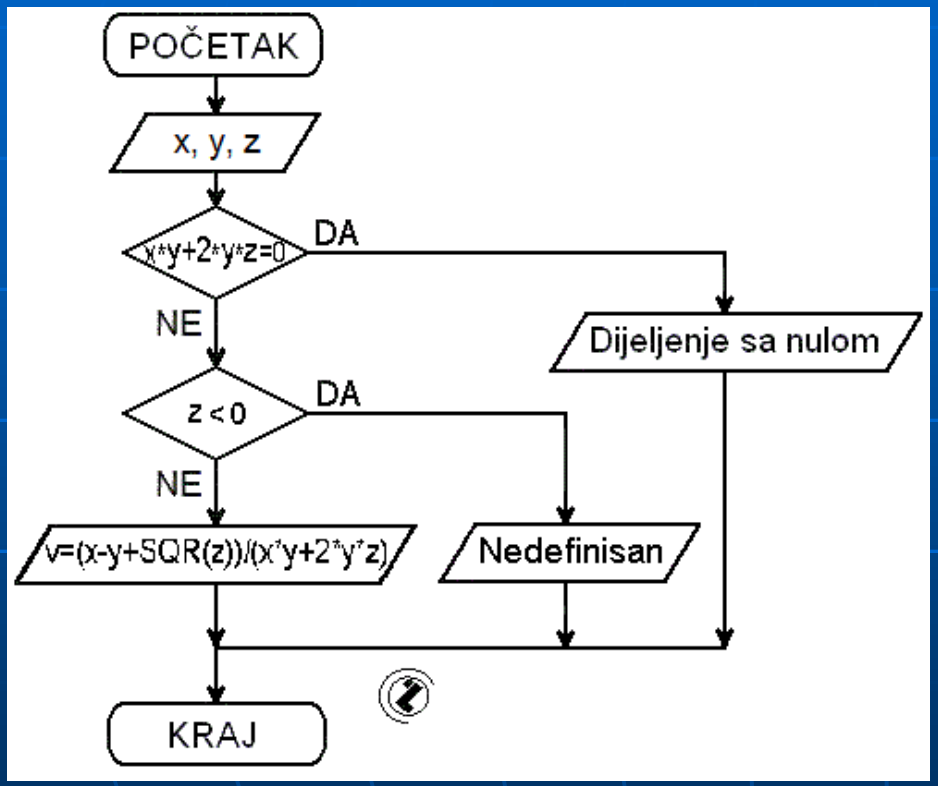
- Napisati program za odredjivanje y po formuli:

$$y = \begin{cases} 1/a, & a \text{ neparno} \\ \sqrt{a+1}, & \text{za ostale slučajeve} \end{cases}$$

- **Ulaz**: a
- **Logički uslov**:
  a / 2 = INT(a / 2)
  Istinit /TRUE/:
  y = SQR(a + 1)
  Lažan /FALSE/:
  y = 1 / a
- **Izlaz**: a, y

Učitati prirodan broj. Ako je neparan ispisati njegovu recipročnu vrednost, a ako je paran ispisati kvadratni koren njegovog sledbenika.

Razgranata algoritamska šema - IF THEN i pseudo kod

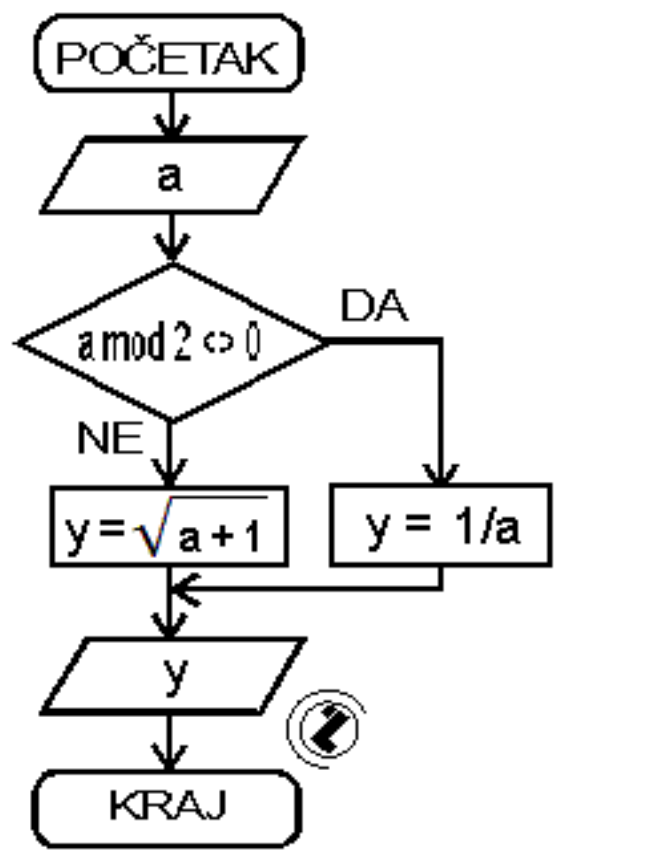


- Napiši program za rešavanje sledećeg izraza:

$$k = \frac{\left(x - y + 2 * y * \sqrt{z}\right)}{x * y + 2 * y * z}$$

- **Ulaz**: x, y, z
- **Logički uslov**:
- x * y + 2 * y * z <> 0
- **Istinit /TRUE/:**
- k = (x - y + SQR(z) / (x * y + 2 * y * z),
- **Izlaz**: x, y, z, k
  Lažan /FALSE/: "Deljenje sa 0 nije definisano"

Razgranata algoritamska šema - IF THEN ELSE i pseudo kod

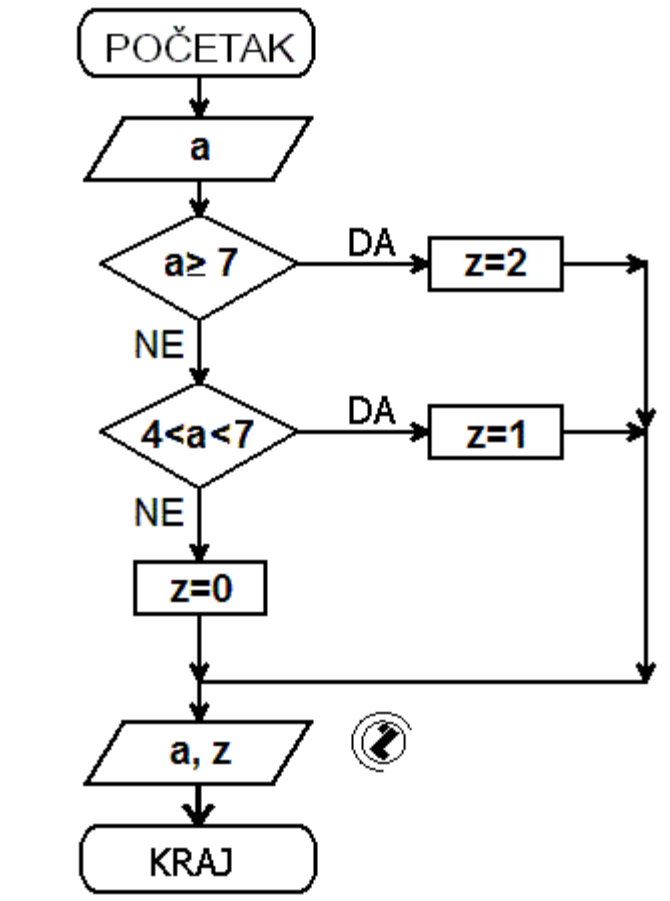


- Učitati prirodan broj. Ako je neparan ispisati njegovu recipročnu vrednost, a ako je paran ispisati kvadratni koren njegovog sledbenika.

$$y = \begin{cases} 1/a, & a \text{ neparno} \\ \sqrt{a+1}, & \text{za ostale slučajeve} \end{cases}$$

- **Ulaz**: a
- **Logički uslov**:
- a / 2 = INT(a / 2)
  Istinit /TRUE/:
- y = SQR(a + 1)
  Lažan /FALSE/:
- y = 1 / a
- **Izlaz**: a, y

Razgranata algoritamska šema - IF THEN ELSE i pseudo kod

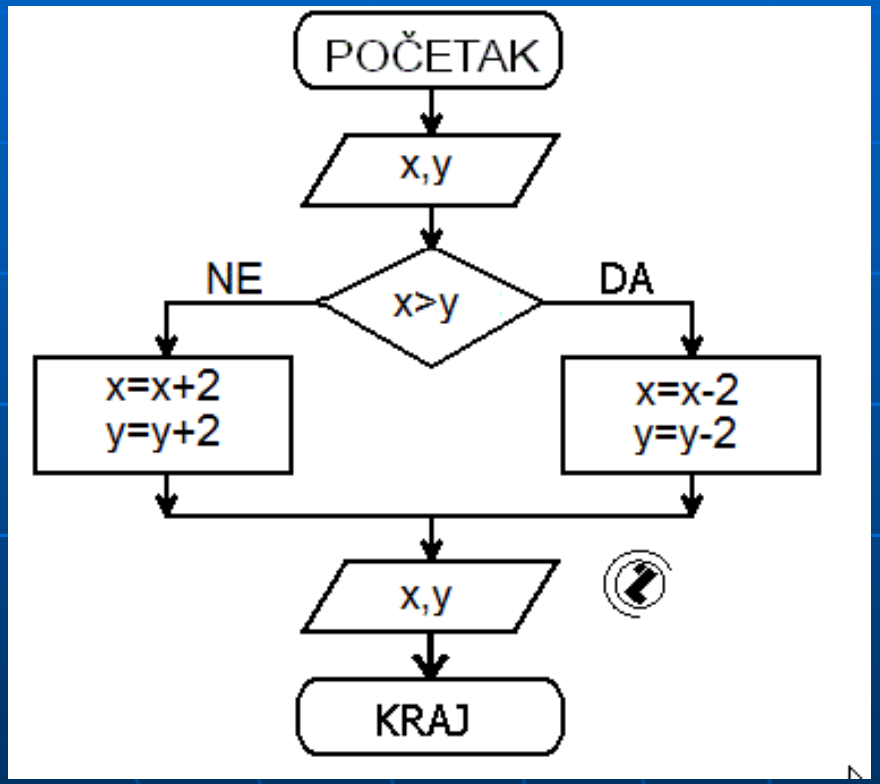


- Ako je :

$$z = \begin{cases} 2, & a \geq 7 \\ 1, & 4 < a < 7 \\ 0, & a \leq 0 \end{cases}$$

- napisati program u pseudo kodu za izračunavanje z.

Razgranata algoritamska šema - IF THEN ELSE i pseudo kod



- Ako je x>y umanjuju se vrednosti obe varijable

x := x - 2;

y := y - 2,

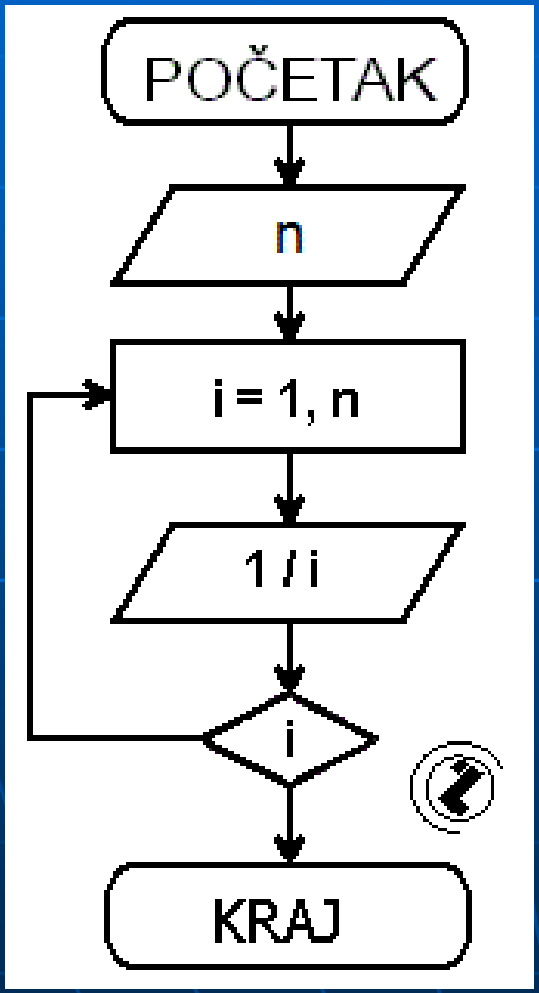
inače

- uvećavaju se vrednosti obe varijable

x := x + 2;

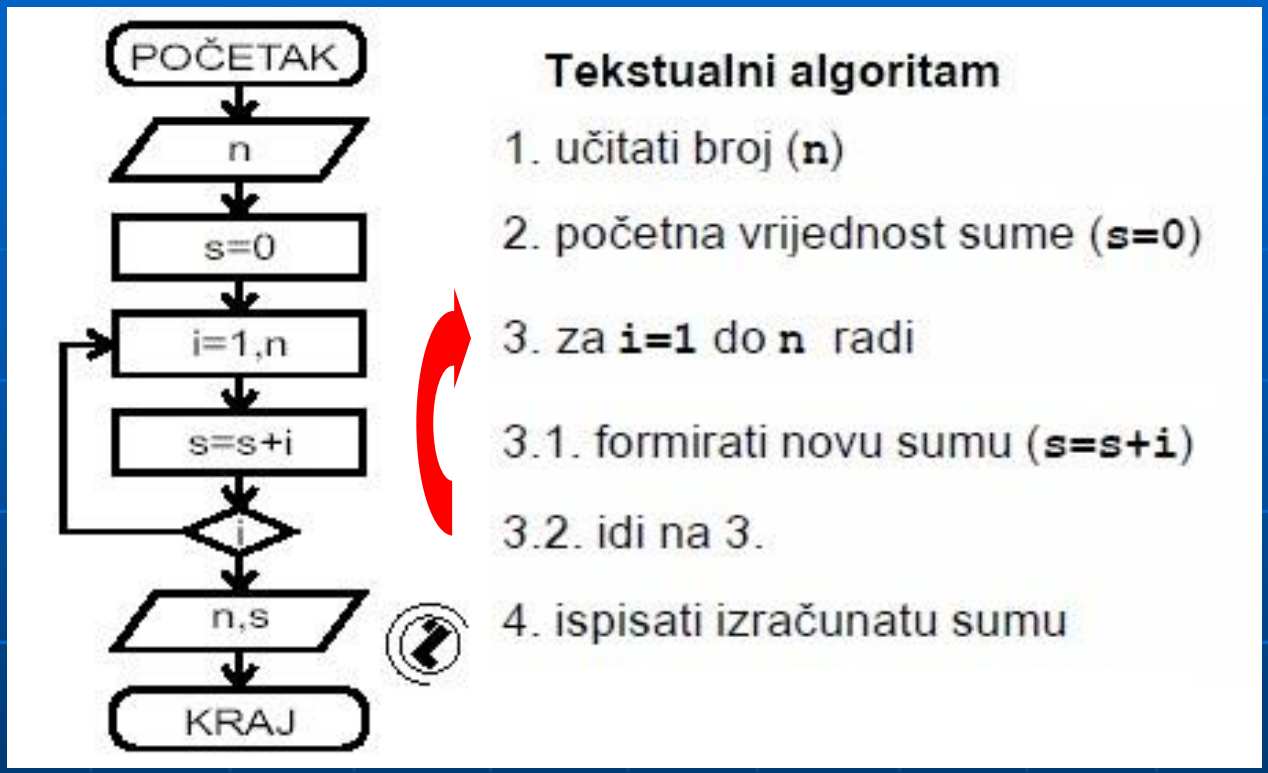
y := y + 2.

Ciklična struktura - FOR petlja



- Napisati program za ispis recipročnih vrednosti prvih n prirodnih brojeva.
- Učitati do kog broja se izvodi ispis (n)
- Za i = 1 do n radi ispis recipročne vrednosti promjenljive i.

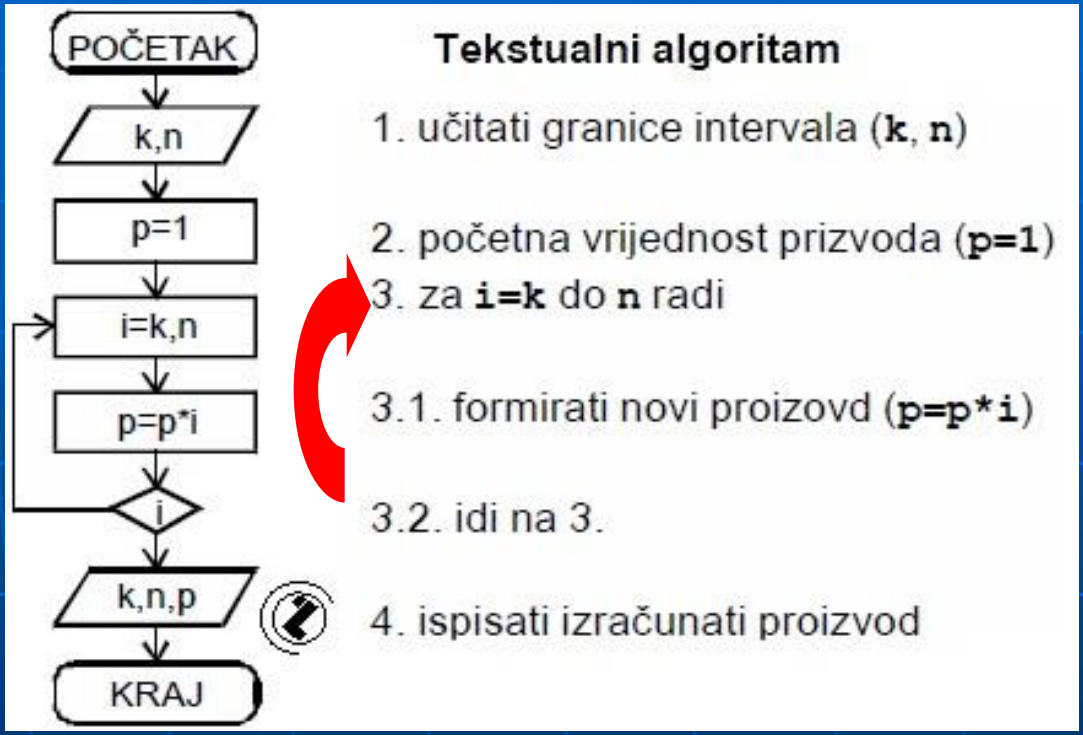
Ciklična struktura - FOR petlja



Izračunati sumu prvih n prirodnih brojeva. Koristiti FOR petlju.

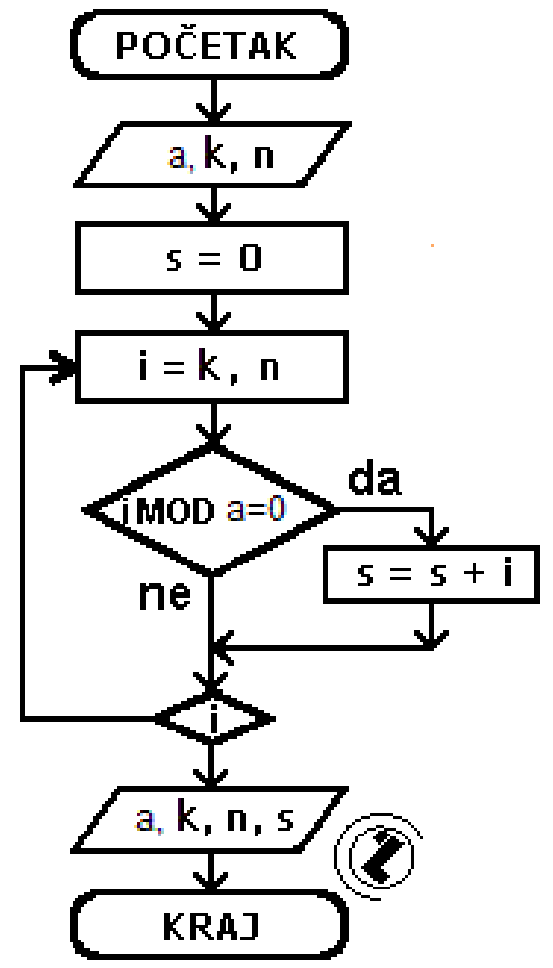
- Izraz s=s+i je računarski, a ne matematički izraz. Njim se predstavlja dinamika promene vrednosti varijable s. Desni deo izraza (s+i) predstavlja uvećanje vrednosti varijable s za vrednost kontrolne varijable i. Izračunata vrednost se pridružuje varijabli s. Zato se izraz s=s+i čita s postaje s+i tj. s prima vrednost s+i. Zatim se uvećava kontrolna varijabla i. Postupak uvećanja i i s se ponavlja sve dok vrednost kontrolne varijable i ne postane n. Ispis je na kraju programa.

Ciklična struktura - FOR petlja



- Izračunati proizvod prirodnih brojeva u intervalu od k do n. Koristiti FOR petlju.
- Izvodjenje programa počinje učitavanjem vrednosti n do koje se izvodi ispis (linije 10 i 20). For petlja omogućuje promenu vrednosti kontrolne varijable od početne n do krajnje vrednosti. Kontrolna varijabla za ovaj zadatak je i. Početna vrednost je k, a krajnja n. Vrednost varijable i se ispisuje na ekran pri svakom prolazu kroz petlju. Početna vrednost proizvoda p je 1. a novi proizvod se formira izrazom p := p * i.

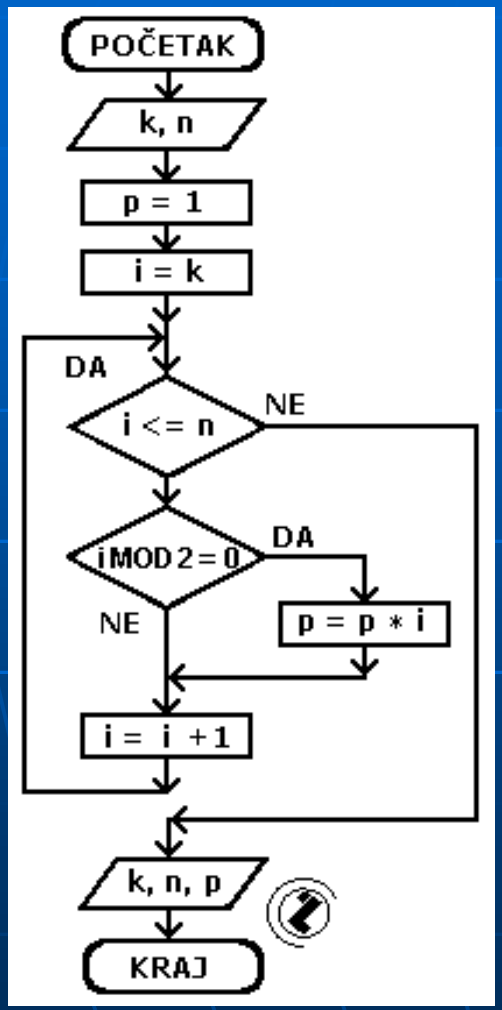
Ciklična struktura - FOR petlja



- učitati granice intervala (k, n)
- učitati broj a
- početna vrednost sume (s=0)
- za i = 1 do n radi ako je i deljivo sa a formirati novu sumu (s=s+i)
- ispisati izračunatu sumu s

Izračunati sumu prirodnih brojeva u intervalu od k do n koji su deljivi sa a.

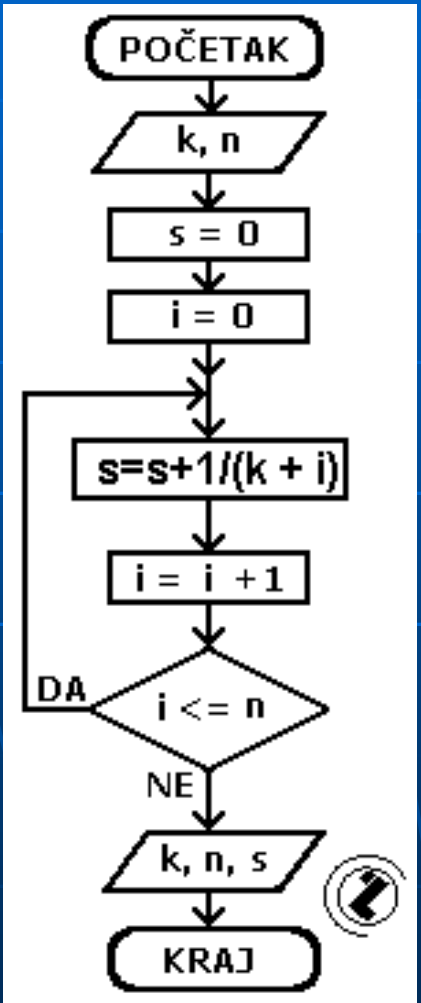
Ciklična struktura - FOR petlja



1. učitati granice intervala (k,n) za sabiranje
2. početna vrednost proizvoda (p=1)
3. početne vrednosti za i (i = k)
4. dok je i manje ili jednako n (i<=n) pređi na sledeće korake; inače idi na nastavak programa (korak 8)
5. ako je i deljivo sa 2 formirati novi proizvod (p=p*i)
6. uvećaj vrednost kontrolne promenljive (i=i+1)
7. idi na korak 4
8. ispisati izračunatu vrednost proizvoda p, granice (k,n)

Proizvod parnih prirodnih brojeva u intervalu od k do n.

Ciklična struktura - FOR petlja

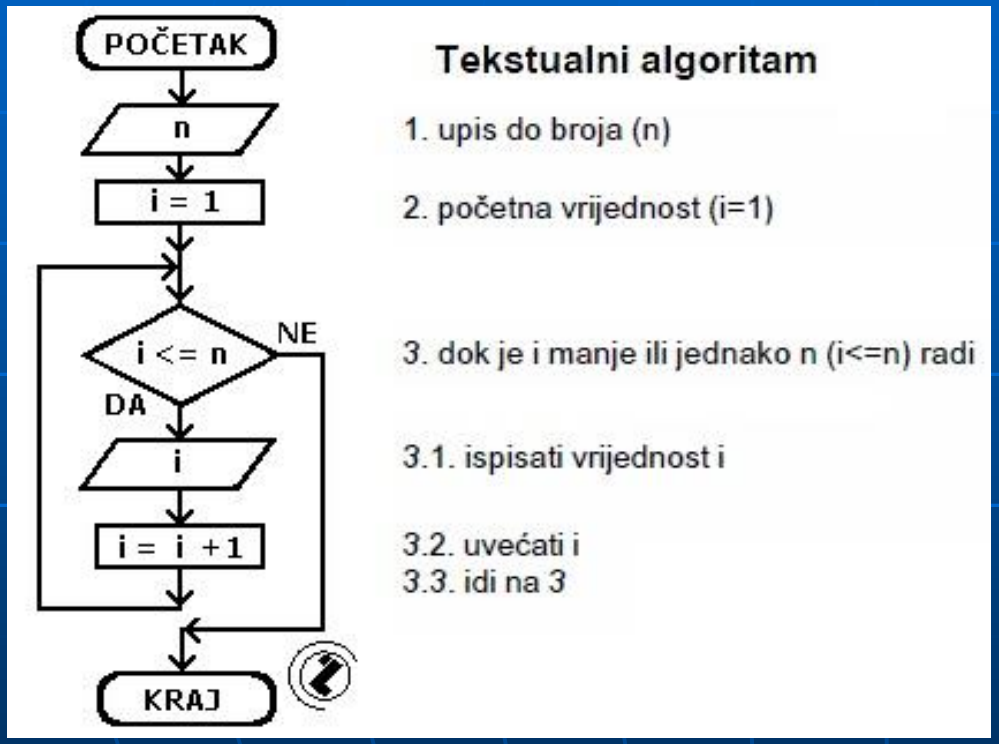


- Napisati program koji će za učitanu vrednost K i N izračunati vrednost izraza i ispisati:

$$s = \frac{1}{k} + \frac{1}{k+1} + \ldots + \frac{1}{k+N}$$

1. učitati granicu intervala (n), učitati koeficijent (k)
2. početna vrednost sume (s=0)
3. početne vrednosti za i (i = 0)
4. početak petlje
5. formirati novu sumu (s=s+1/(k+i))
6. uvećaj vrednost kontrolne promenljive (i=i+1)
7. ako je promenljiva i veća od n izađi iz petlje inače idi na korak 4
8. ispisati izračunatu vrijednost sumu s, granicu (n) i koeficijent (k)

Ciklična struktura - WHILE petlja



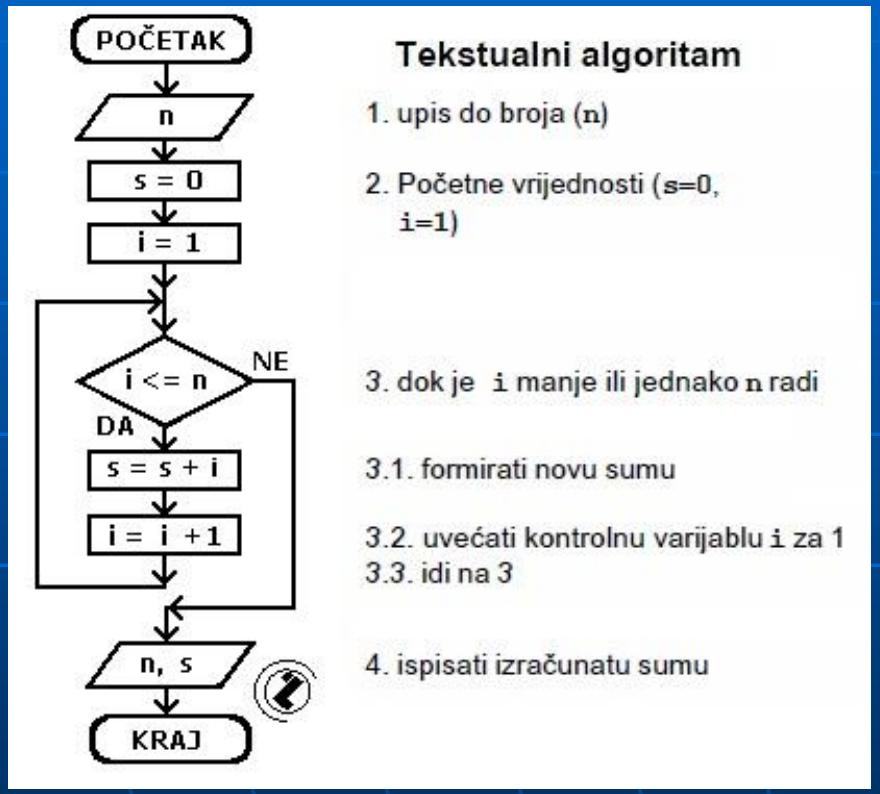
- Obezbediti ispis prvih n prirodnih brojeva sa WHILE petljom.

- **Opis rešenja:**

1. Na početku se upisuje do koje vrednosti (n) se izvodi ispis.
2. Kontrolnoj varijabli i se prideljuje vrednost jedan (1).
3. Zatim se prelazi na proveru logičkog izraza u WHILE petlji.
4. WHILE petlja se ponavalja sve dok je logički izraz (i<=n) istinit.
5. Prva naredba u petlji je ispis vrednosti kontrolne varijable i na ekran.
6. Naredba i=i+1 uvećava vrednost varijable i za jedan.
7. Prvo vrednosti varijable i dodaje 1, zatim izračunatu sumu pridružuje varijabli i.
8. Izvršenje programa prelazi na početak WHILE petlje, koja obezbeđuje ponavljanje.

WHILE petlja se neće nikada izvršiti ako je uslov neistinit !!!

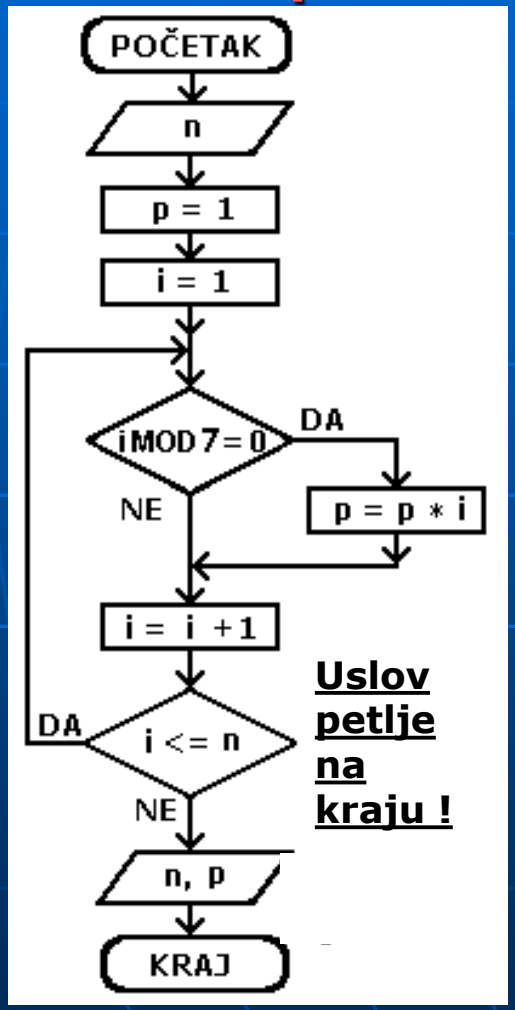
Ciklična struktura - WHILE petlja



- Obezbediti izračuanavanje sume prvih n prirodnih brojeva. Zadatak rešiti sa WHILE petljom.

- **Opis rešenja:**
- Početna vrednost varijable s (suma) je 0.
- Suma se formira po izrazu s=s+i. Računar pri izvodjenju prvo izvodi desni deo, sabira vrednost s i vrednost i.
- Izračunata vrednost se pridružuje varijabli s.
- Zatim se vrednost varijable i uvećava za jedan (i=i+1).
- Postupak uvećanja sume i vrednosti varijable i se ponavlja sve dok je vrednost varijable i<=n.

Ciklična struktura – DO WHILE petlja
(rešava WHILE problem)

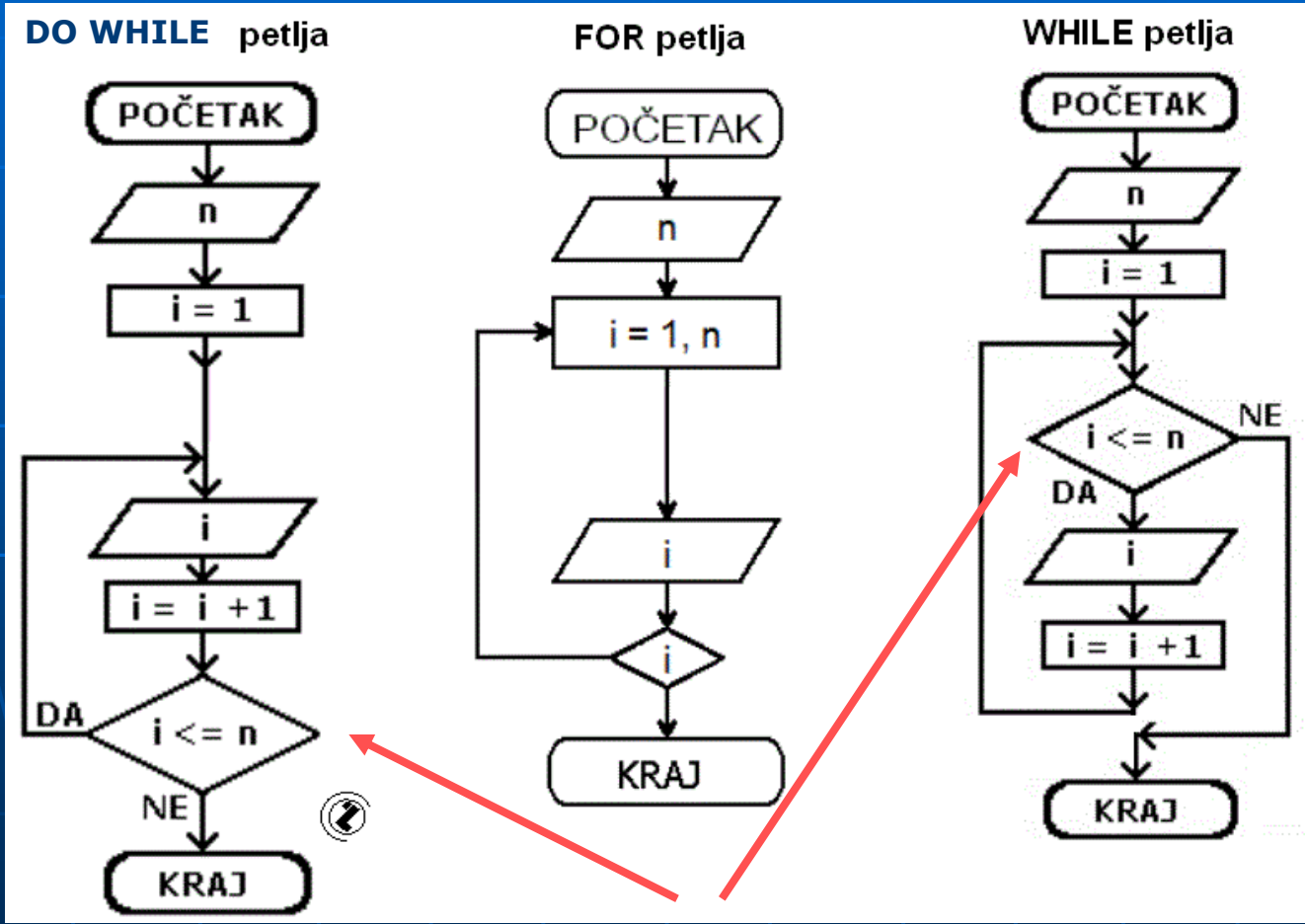


Pseudo kod:

1. Učitati do kog broja se izvodi množenje (n)
2. Početna vrednost proizvoda (p=1)
3. Početne vrednosti za i (i =1)
4. Početak petlje
5. Ako je i deljivo sa 7 formirati novi proizvod (p=p*i)
6. Uvećaj vrednost kontrolne promenljive (i=i+1)
7. Ako je promenljiva i veća od n izađi iz petlje, inače idi na korak 4
8. Ispisati izračunati proizvod p

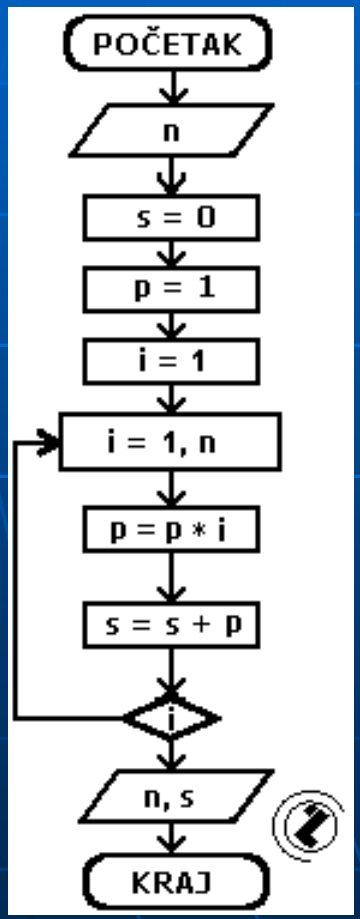
Napisati program za ispis proizvoda brojeva od 1 do n koji su djeljivi sa 7.

Ispis prirodnih brojeva od 1 do N - u tri petlje

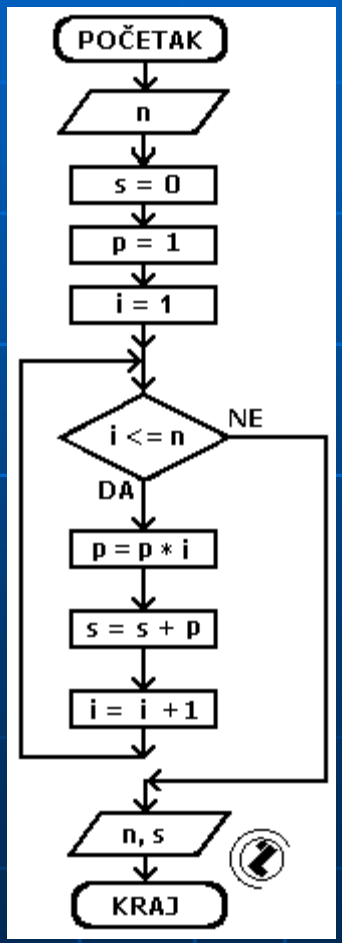


Izračunati sumu faktorijela prvih n prirodnih brojeva (S = 1! + 2! + 3! + … + N!).

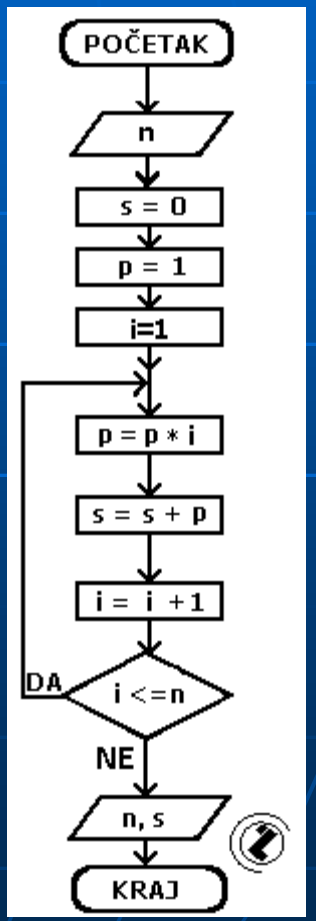
FOR petlja



WHILE petlja

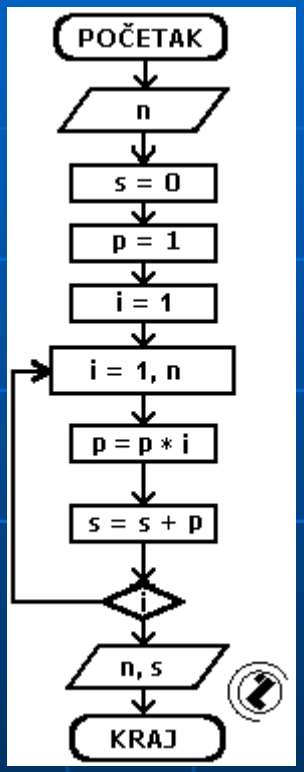


DO WHILE petlja

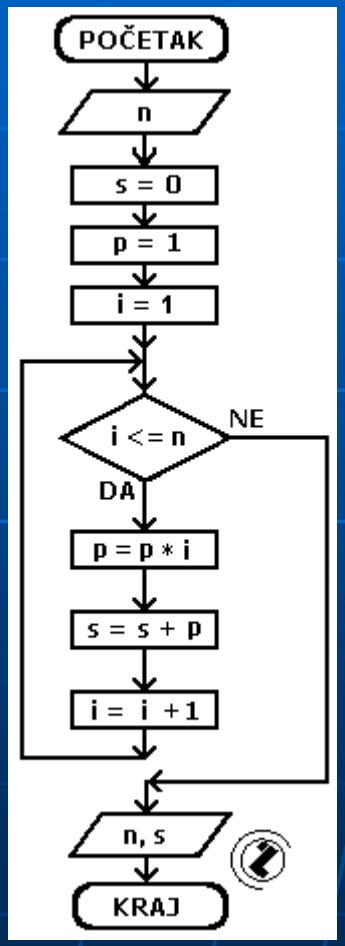


Izračunati sumu faktorijela prvih n prirodnih brojeva (S = 1! + 2! + 3! + … + N! (FOR, WHILE i DO WHILE petlja)

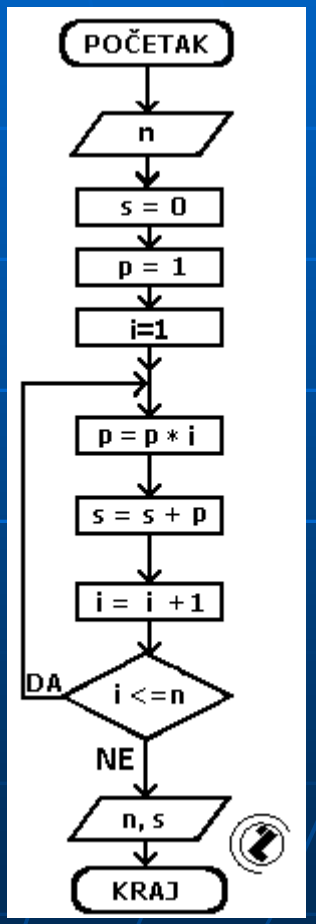
FOR petlja



WHILE petlja



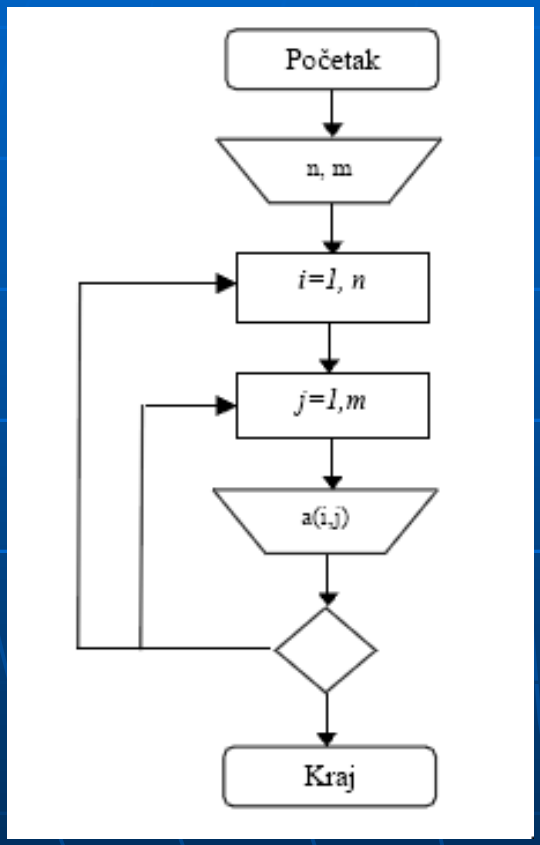
DO WHILE petlja



Dvodimenzionalna polja

Nacrtati algoritam za učitavanje svih elemenata matice A reda *m x n*.

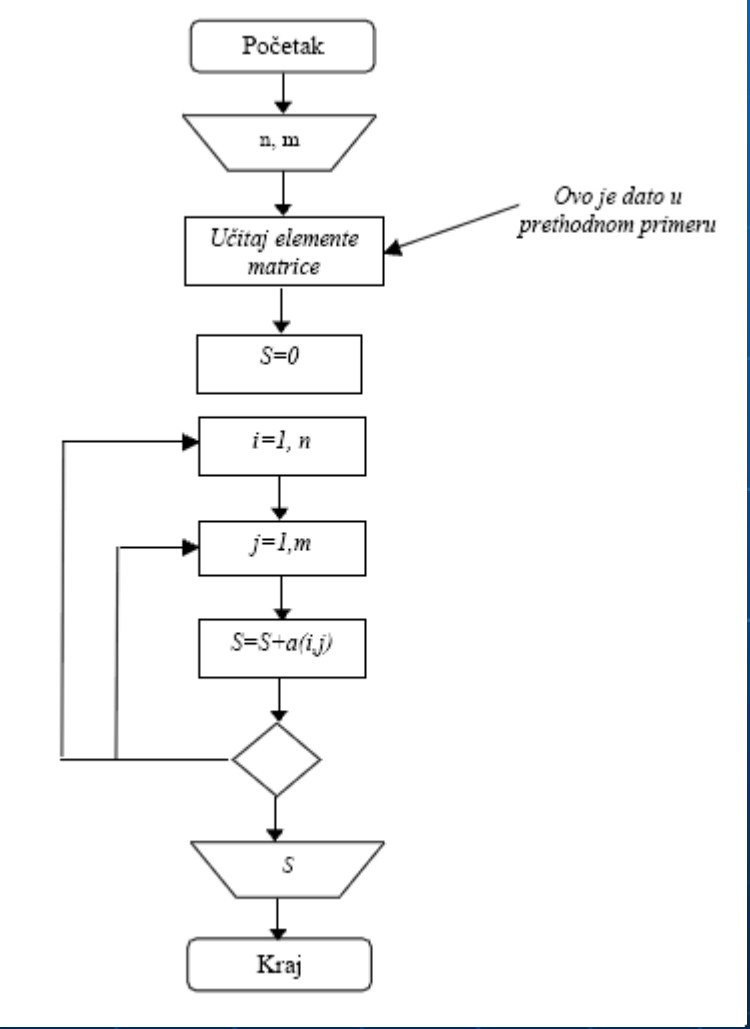
$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{pmatrix}$$



1. Učitati vrednosti za m i n.
2. Uvesti novu pomoćnu promenljivu i koja je u dijapazonu od 1 do n za vodeću petlju
3. Uvesti novu pomoćnu promenljivu j koja je u dijapazonu od 1 do m za ugnježdenu petlju
4. Učitavanje vrednosti članova: vodeća petlja jednom a ugnježdena do kraja, pa opet ponovo u isti ciklus
5. ispunjenje uslova vodeće petlje završava ciklus
6. Kraj

Dvodimenzionalna polja

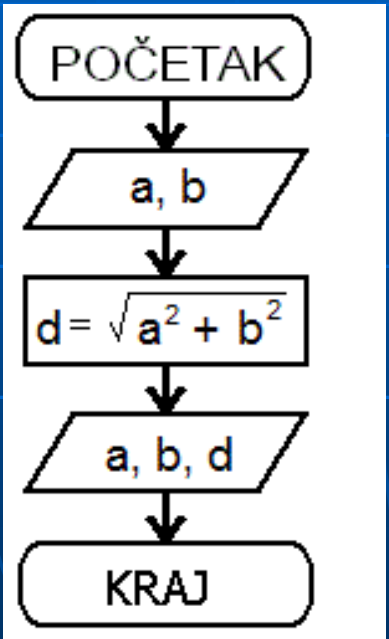
- Nacrtati algoritam za sabiranje svih elemenata matice A reda *m x n* kao u predhodnom primeru



1. Učitati vrednosti za m i n.
2. Uvesti novu pomoćnu promenljivu i koja je u dijapazonu od 1 do n za vodeću petlju
3. Uvesti novu pomoćnu promenljivu j koja je u dijapazonu od 1 do m za ugnježdenu petlju
4. Postaviti vrednost sume na nulu
5. Učitavanje vrednosti članova: vodeća petlja jednom a ugnježdena do kraja, pa opet ponovo u isti ciklus
6. Izračunavanje sledeće sume članova
7. Ispunjenje uslova vodeće petlje završava ciklus
8. Kraj

Primeri u programskim jezicima

- Napisati program za izračunavanje dijagonale pravougaonika. Zadatak obrazložiti algoritmom, pseudo kodom i programom.



Pseudo kod:

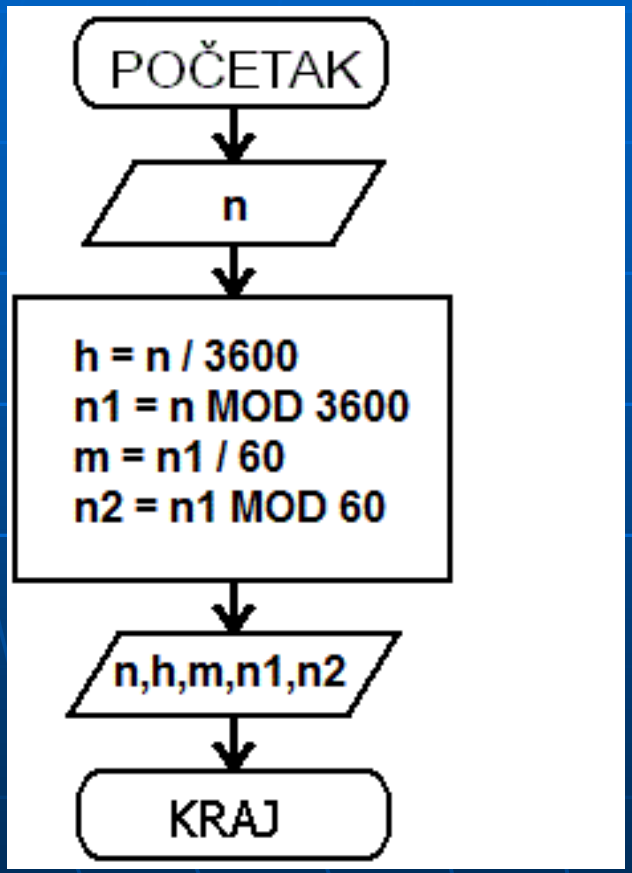
ulaz - učitati: a, b **(format)**
obrada - izračunati: d = SQRT($a^2 + b^2$)
izlaz - ispisati: a, b, d
N a p o m e n a:
SQRT($a^2 + b^2$) - kvadratni koren ($a^2 + b^2$).

Listing programa C:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main ()
{     float a,b,d;
printf("unesi stranicu a:");
scanf("%f",&a);
printf("unesi stranicu b:");
scanf("%f",&b);
d=sqrt((a*a)+(b*b));
printf("dijagonala pravokutnika stranice je %.1f",d);
getch();     }
```

Primeri u programskim jezicima

•Putovanje traje n sekundi. Izračunaj koliko je to sati, minuta i sekundi. (Sat ima 3600 sekundi. Deljenjem vremena u sekundama sa 3600 dobijamo sate sat = n DIV 3600. Ostatak sekundi (sek1) se deli sa 60 i dobijamo minute, a ostatak deljenja sek1 sa 60 su sekunde).



Pseudo kod:

- ulaz - učitati: "Unesi broj sekundi.", n (format)
- obrada - izračunati:
  h = n / 3600
  n1 = n MOD 3600
  m = n1 / 60
  n2 = n1 MOD 60
- izlaz - ispisati: "Broj sati je: ";h; ", broj minuta je: ";m; ", a broj sekundi je:";n2

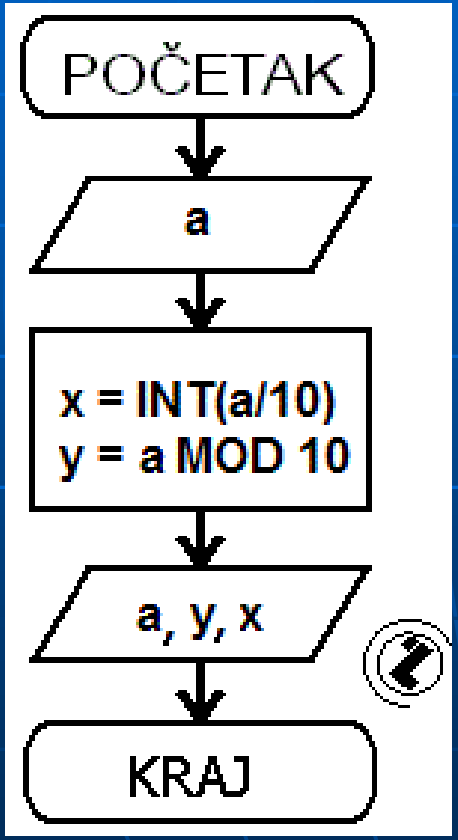
Listing programa C:

- #include<stdio.h>
- #include<conio.h>main()
- { int n,sat,min,sek, sek1;
- printf("Ucitati sekunde\n");
- scanf("%d",&n);
- sat=n/3600;
- sek1=n%3600;
- min=sek1/60;
- sek=sek1%60;
- printf("proteklo %d sati,%d minuta i %d sekundi",sat,min, sek);
- getch(); }

Primeri u programskim jezicima

•Izdvojiti cifre dvocifrenog broja i ispisati unazad.

(Pri pretvaranju se koristi osobina pozicionog sistema. Dvocifren broj broj se može predstaviti kao = $x*10^1 + y*10^0$. Ostatak deljenja dvocifrenog broja sa 10 broj % 10 daje jedinice broja (u ovom slučaju dvocifrenog). Celobrojnim deljenom sa 10 (broj / 10) dvocifrenog broja se dobija cifra desetica.)



Pseudo kod:

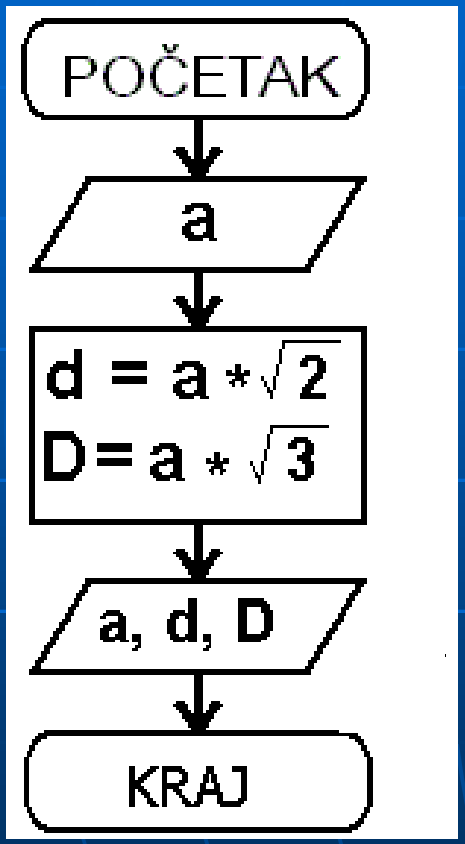
- **ulaz - učitati**: "Dvocifren broj "; broj (format)
- **obrada - izračunati**:
  y = broj MOD 10 // cifra jedinica
  x = INT (broj/10) // cifra desetica
- **izlaz - ispisati**: "broj=", broj, " cifra jedinica=", y, " cifra desetica=", x

Listing programa C:

```
#include<stdio.h>
#include<math.h>
main()
{ int x,y,broj;
printf("unesi dvocifren broj:");
scanf("%d",&broj);
y=broj %10;
x=broj/10;
printf("jedinica ima %d a desetica %d",y,x);
getch();      }
```

Primeri u programskim jezicima

Izračunati dijalgonale d i D kocke (za poznatu stranicu a).



Pseudo kod:

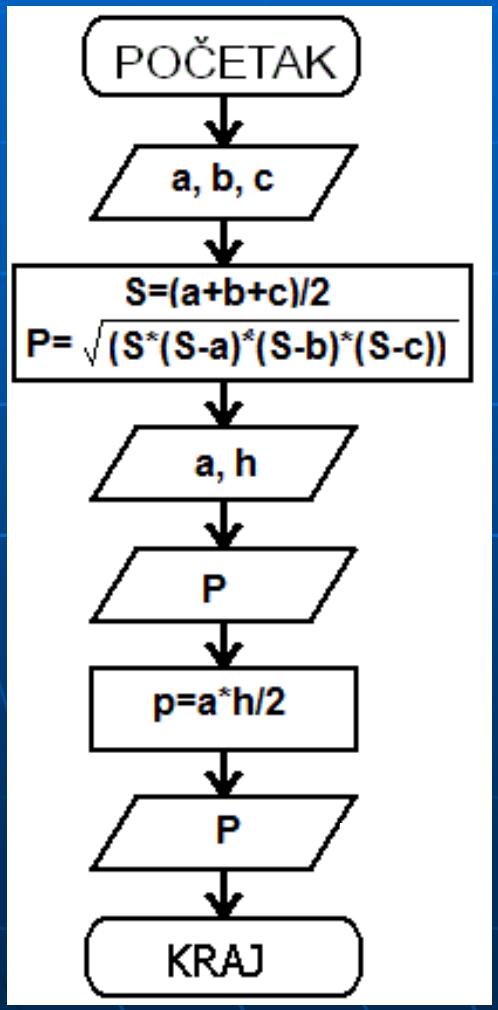
ulaz - učitati: a
obrada - izračunati:
d = a * SQR(2)
D = a * SQR(3)
izlaz - ispisati: "Dijagonala d je: ",d, "Dijagonala D je: ",D

Listing programa C:

```
#include<stdio.h>
#include<math.h>
main(){
float a,d,D;
printf("unesi stranicu kocke a;");
      scanf("%f",&a);
d=a*sqrt(2);
D=a*sqrt(3);
printf("dijagonala d je %.1f\n",d);
      printf("dijagonala D je %.1f\n",D);
      getch();     }
```

Primeri u programskim jezicima

•Napisati program za izračunavanje površine trougla: koristeći Heronovu formulu: S=(a+b+c)/2, P=SQRT(S(S-a)(S-b)(S-c)) pa zatim i po formuli P=a*h/2. Uporediti dobijene rezultate.



Pseudo kod:

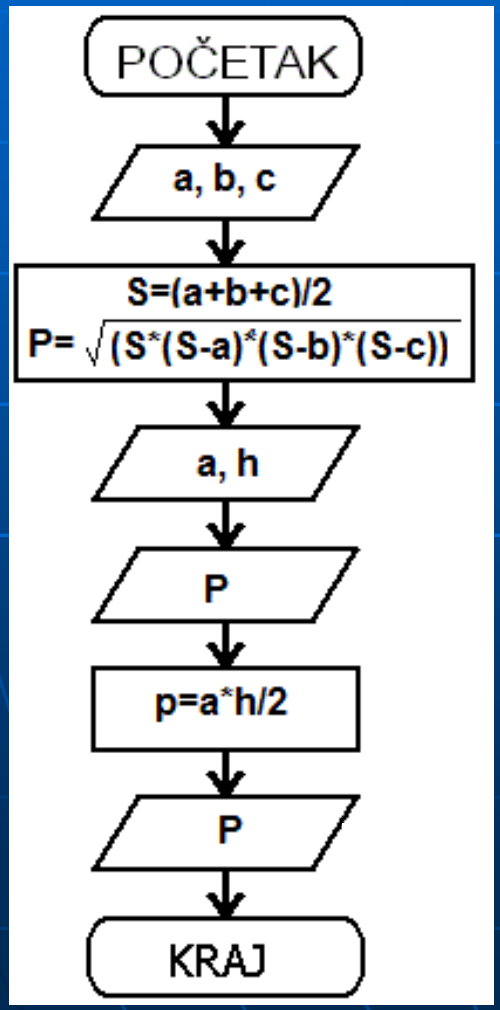
- **ulaz - učitati**: "stranice trougla"; a, b, c (format)
- **obrada - izračunati**:
  S=(a+b+c)/2
  P1 = SQRT(S*(S-a)*(S-b)*(S-c))
  izlaz - ispisati: "povrsina trougla po Heronovoj formuli iznosi:"; P1
- **ulaz - učitati**: "stranica a i visina h"; P2
- **obrada - izračunati**: p2=a*h/2
- **izlaz - ispisati**: "Povrsina trougla po formuli P2=a*h/2 iznosi:"; P2

Listing programa C:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
main()
{    float a,b,c,S,h,P1,P2;
printf("Unesi duzine stranica trougla: \n");
scanf("%f %f %f",&a,&b,&c);
S=(a+b+c)/2;
P1=sqrt(S*(S-a)*(S-b)*(S-c));
printf("Povrsina trougla iznosi: %.2f",P1);
printf("Unesi visinu trougla: \n");
scanf("%f ",&h);
P2=a*h/2;
printf("Nova povrsina trougla iznosi: %.2f",P2);
return 0;
```

Primeri u programskim jezicima

•Napisati program za izračunavanje površine trougla: koristeći Heronovu formulu: S=(a+b+c)/2, P=SQRT(S(S-a)(S-b)(S-c)) pa zatim i po formuli P=a*h/2. Uporediti dobijene rezultate.



Pseudo kod:

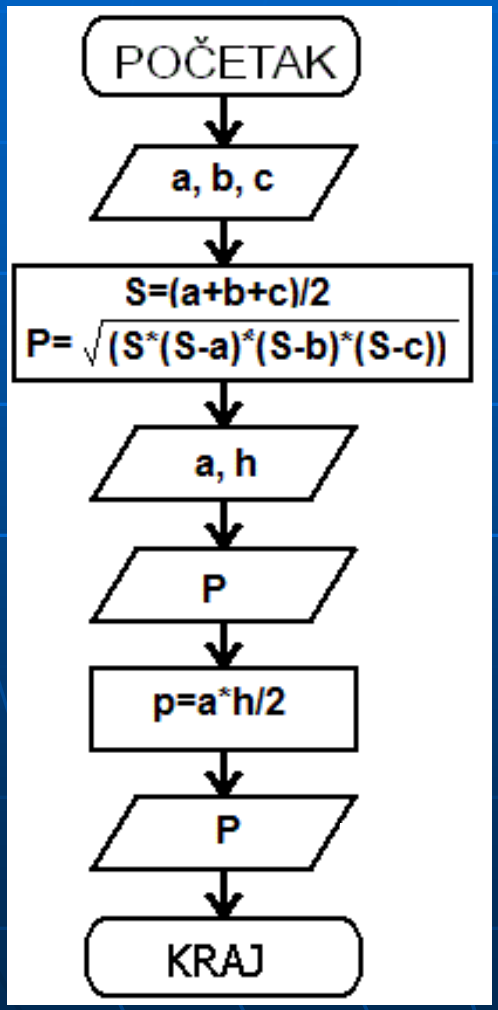
- **ulaz - učitati**: "stranice trougla"; a, b, c (format)
- **obrada - izračunati**:
  S=(a+b+c)/2
  P1 = SQRT(S*(S-a)*(S-b)*(S-c))
  izlaz - ispisati: "povrsina trougla po Heronovoj formuli iznosi:"; P1
- **ulaz - učitati**: "stranica a i visina h"; P2
- **obrada - izračunati**: p2=a*h/2
- **izlaz - ispisati**: "Povrsina trougla po formuli P2=a*h/2 iznosi:"; P2

Listing programa PASCAL:

```
PROGRAM P06411081;
USES    WinCRT;
VAR  a, b, c, s, p: Real;
BEGIN
Write('Stranice trougla ');
Readln(a, b, c);
s  := (a + b + c)/2;
p  := S*(S-a)*(S-b)*(S-c)/2;
Writeln('a ', a, ' b ', b, ' c ', c);
Writeln('Povrsina trougla je: ', p);
END.
```

Primeri u programskim jezicima

•Napisati program za izračunavanje površine trougla: koristeći Heronovu formulu: S=(a+b+c)/2, P=SQRT(S(S-a)(S-b)(S-c)) pa zatim i po formuli P=a*h/2. Uporediti dobijene rezultate.



Pseudo kod:

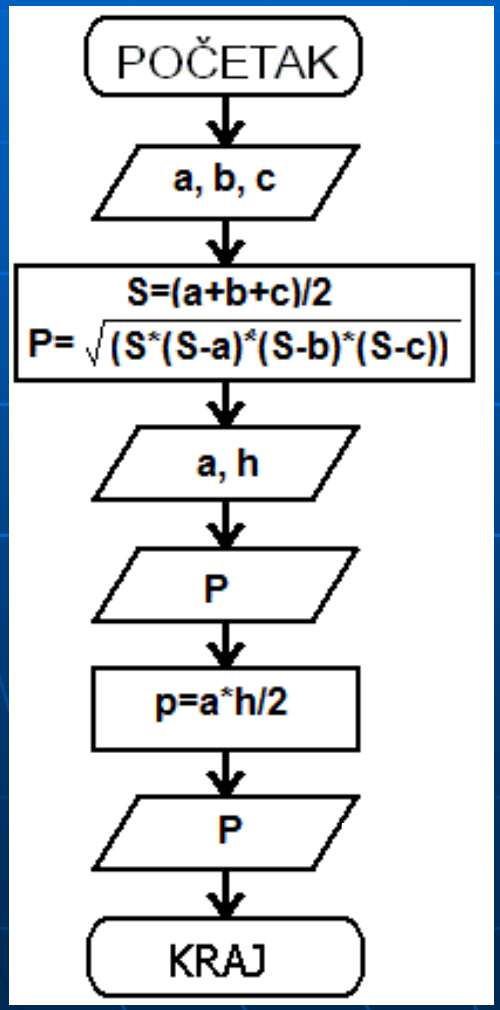
- **ulaz - učitati**: "stranice trougla"; a, b, c (format)
- **obrada - izračunati**:
  S=(a+b+c)/2
  P1 = SQRT(S*(S-a)*(S-b)*(S-c))
  izlaz - ispisati: "povrsina trougla po Heronovoj formuli iznosi:"; P1
- **ulaz - učitati**: "stranica a i visina h"; P2
- **obrada - izračunati**: p2=a*h/2
- **izlaz - ispisati**: "Povrsina trougla po formuli P2=a*h/2 iznosi:"; P2

Listing programa JAVA:

```
package zadatak;
import java.util.Scanner;
public class P06411081 {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("Unesi stranice trougla:");
        double a = input.nextDouble();
        double b = input.nextDouble();
        double c = input.nextDouble();
        double s = (a + b + c) / 2;
        double P = Math.sqrt((s * (s - a) * (s - b) * (s - c)));
        System.out.println("Povrsina trougla je" + P);
    }
```

Primeri u programskim jezicima

•Napisati program za izračunavanje površine trougla: koristeći Heronovu formulu: S=(a+b+c)/2, P=SQRT(S(S-a)(S-b)(S-c)) pa zatim i po formuli P=a*h/2. Uporediti dobijene rezultate.



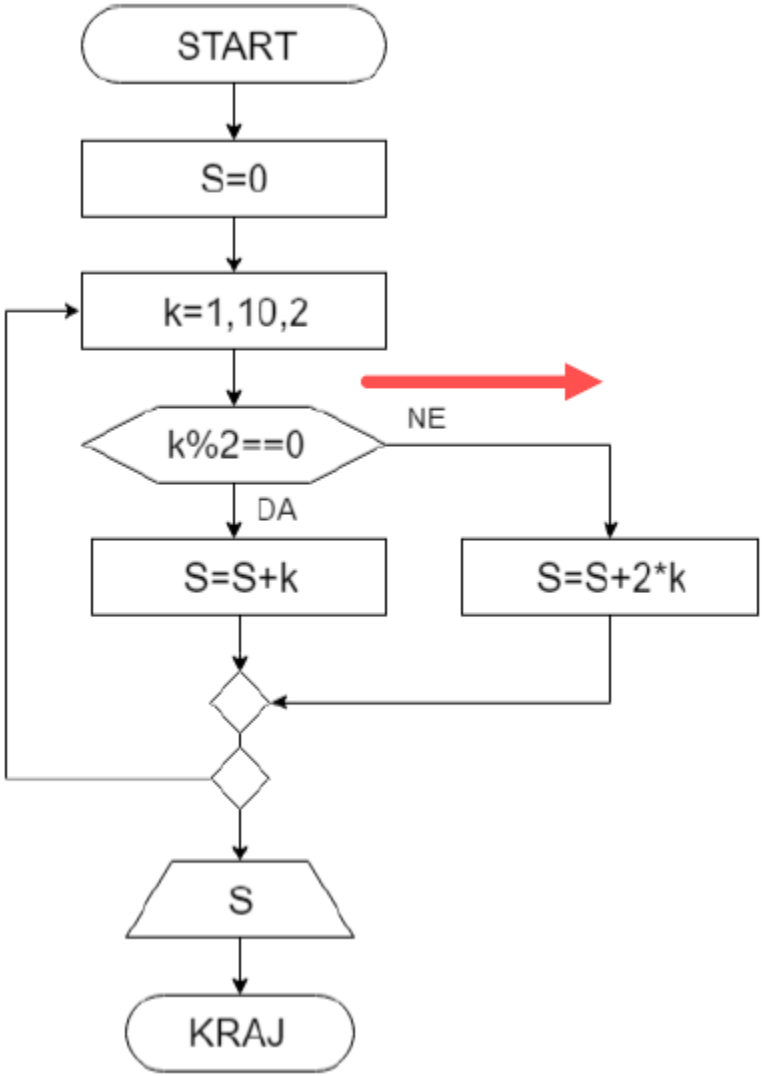
Pseudo kod:

- **ulaz - učitati**: "stranice trougla"; a, b, c (format)
- **obrada - izračunati**:
  S=(a+b+c)/2
  P1 = SQRT(S*(S-a)*(S-b)*(S-c))
  izlaz - ispisati: "povrsina trougla po Heroneru iznosi:"; P1
- **ulaz - učitati**: "stranica a i visina h"; P2
- **obrada - izračunati**: p2=a*h/2
- **izlaz - ispisati**: "Povrsina trougla po formuli P2=a*h/2 iznosi:"; P2

Listing programa BASIC:

- REM
- INPUT "Unesi stranice trougla"; a,b,c
- S=(a+b+c)/2
- P=SQRT(S*(S-a)*(S-b)*(S-c))
- PRINT "Povrsina trougla po Heronovoj formuli iznosi:";
- P INPUT "Unesi stranicu a i visinu ha"; a,h
- p=a*h/2
- PRINT "Povrsina trougla po formuli P=a*h/2 iznosi:"; p
- END

Analiza algoritama

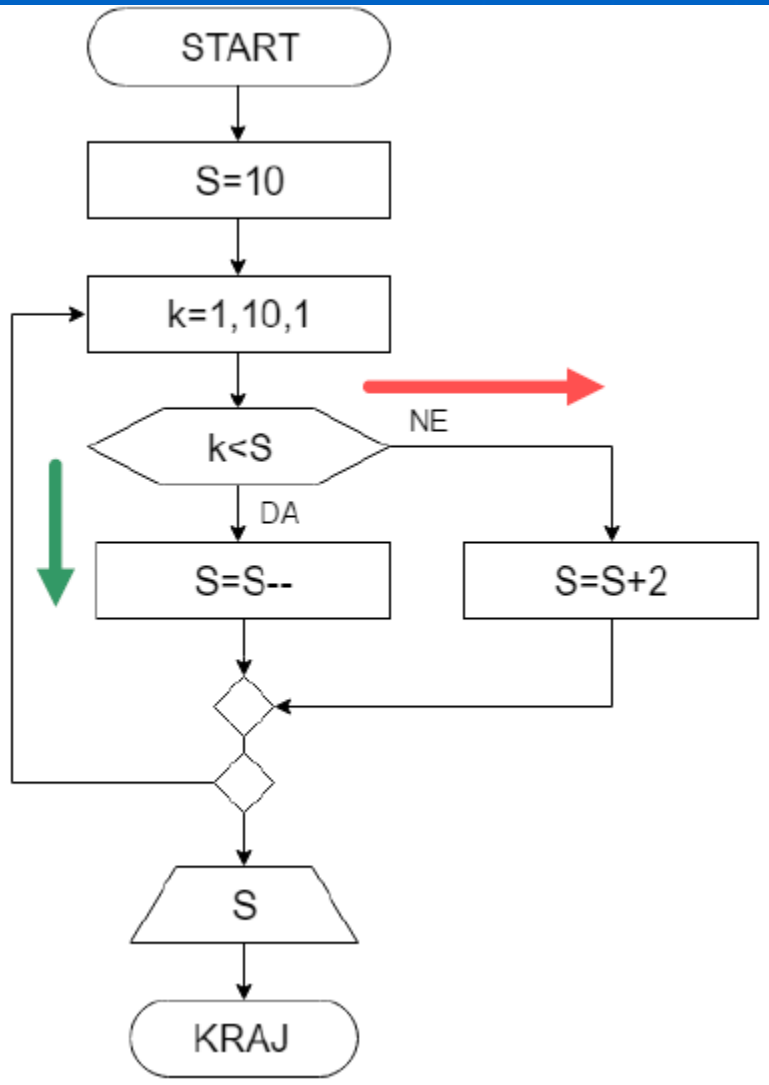


- Odrediti šta se dobija kao izlaz algoritma prikazanog na slici.
- ANALIZA:

k = 1	S=0+2*1	S=2
k = 3	S=2+2*3	S=8
k = 5	S=8+2*5	S=18
k = 7	S=18+2*7	S=32
k = 9	S=32+2*9	S=50

- IZLAZ JE: 50

Analiza algoritama



- Odrediti šta se dobija kao izlaz algoritma prikazanog na slici.
- ANALIZA:

k = 1	S=10-1	S=9
k = 2	S=9-1	S=8
k = 3	S=8-1	S=7
k = 4	S=7-1	S=6
k = 5	S=6-1	S=5
k = 6	S=5+2	S=7
k = 7	S=7+2	S=9
k = 8	S=9-1	S=8
k = 9	S=8+2	S=10
k = 10	S=10+2	S=12

- IZLAZ JE: 12

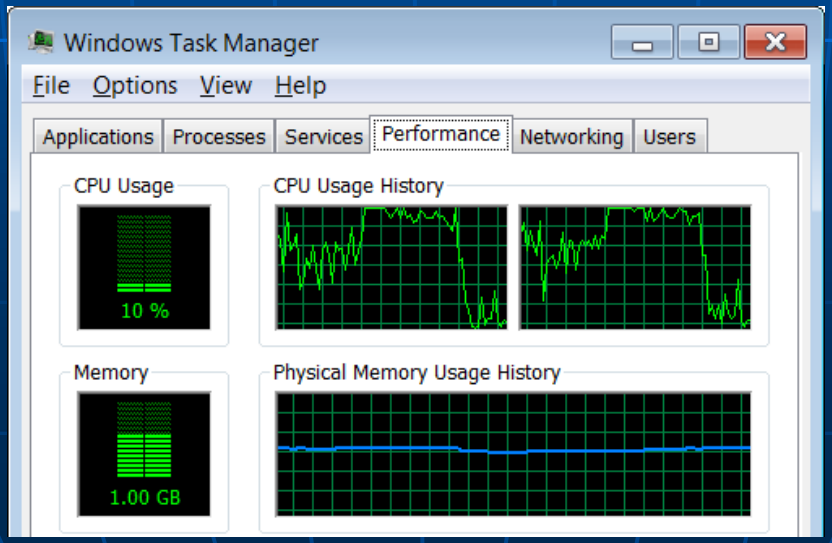
SLOŽENOST ALGORITMA

- Uteoriji složenosti(što nije isto što iteorija izračunljivosti)se izučava problematika složenosti, tj. Kompleksnosti algoritma, u smislu zauzimanja resursa.

Pod pojmom resursi se podrazumeva:

- **PROSTOR**-količina zauzete memorije
- **VREME**-količina potrošenog vremena procesora za izvršenje

- Složenost zavisi od veličine ulaznih podataka.
- Algoritmi se prave za rešenje opšteg problema, bez obzira na veličinu ulaza,ali sa druge strane razne ulazne veličine izazivaju da programi pisani na osnovu algoritmatroše razne količine resursa.



SLOŽENOST ALGORITMA

- Algoritam (A) za uređivanje niza od *n* elemenata je vremenske složenosti n^2.To znači da dvostruko veći broj elemenata zahteva četiri puta više vremena za uređivanje.

- Ako je, drugi algoritam (B) malo pažljivije napisan i brži je dvostruko, on će raditi dvostruko brže za bilo koju veličinu niza.
- Međutim, ako se programer namuči i osmisli suštinski drugačiji algoritam (C) za uređivanje, on stvarno može biti reda složenosti $n \cdot log(n)$.

- Algoritmi (A) i (B) su iste složenosti, jer se u notaciji sa velikim O obeležavaju sa $O(n^2)$, a u govoru se zovu 'algoritmi kvadratne složenosti', dok je algoritam (C) 'algoritam složenosti $n \cdot log(n)$'.

- Zaključak: algoritam (C) je najbolji sa stanovišta korišćenja vremena za velike setove ulaznih podataka.

SLOŽENOST ALGORITMA

- Ocena (vremenske) složenosti algoritma sastoji se u brojaču računskih koraka koje treba izvršiti.
- Međutim, termin računska operacija može da podrazumeva različite operacije, na primer sabiranje i množenje, čije izvršavanje traje različito vreme.
- Različite operacije se mogu posebno brojati, ali je to obično komplikovano. Pored toga, vreme izvršavanja zavisi i od konkretnog računara, izabranog programskog jezika, odnosno prevodioca.
- Zato se obično u okviru algoritma izdvaja neki osnovni korak, onaj koji se najčešće ponavlja.

SLOŽENOST ALGORITMA

- Postoje 3 mere na osnovu kojih se porede efikasnosti algoritama:
  - 1. najgori mogući slučaj
  - 2. prosečan slučaj
  - 3. brzina izvršavanja na unapred odredjenom skupu instanci (eng. benchmarks)
- Za merenje u druga 2 slučaja potrebno je imati više informacija o problemu:
  - koji je to prosečan ulaz i koliko se često javlja,
  - kakva je priroda problema i kakve instance problema se najčešće susreću.
- U prvom slučaju prednost je što se pomoću matematičkog aparatačesto lako odredi efikasnost nekog algoritma; prvi način se pokazao dobar i zato je opšte prihvaćen.

SLOŽENOST ALGORITMA

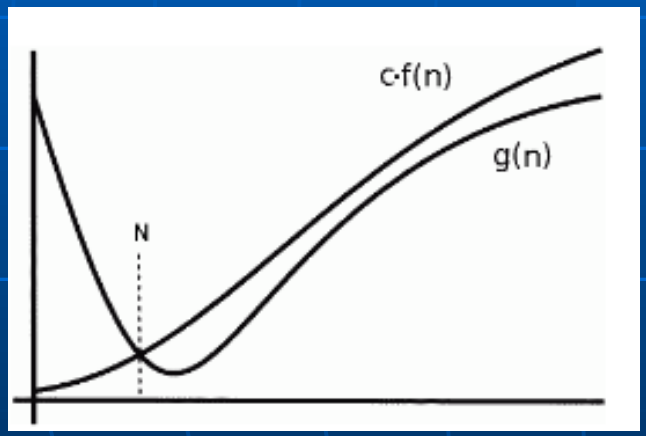
- **Složenost algoritma** (vremenska ili prostorna) je obično **neka funkcija** koja povezuje veličinu problema (ulaza) sa brojem koraka izvršavanja algoritma **(vremenska složenost)** ili brojem potrebnih memorijskih lokacija **(prostorna složenost)**. Uobičajeno je da se složenost algoritama procenjuje u asimptotskom smislu, tj. da se funkcija složenosti procenjuje za dosta velike dužine ulaza. Za to se koriste "veliko O" notacija **(O())**, "omega notacija" **(Ω())**, i "tetanotacija" **(Θ())**.

- "Veliko O" notacija, poznata kao Landau ili Bahman-Landau notacija, **opisuje granično ponašanje funkcije** kada argument teži nekoj specifičnoj vrednosti ili beskonačnosti, obično u terminima jednostavnijih funkcija. "Veliko O" notacija - dobila je ime od "order of" ili "red veličine" pa ćemo za vremensku složenost koja je reda O(f(n)) govoriti da je "proporcionalna" sa f(n). Analiza vremenske složenosti algoritma **ne mora biti precizna** (da prebroji svaki korak u izvršavanju algoritma), već je dovoljno da odredi najveće elemente takvih

Oznake O,Ω,Θ

- Veliko O notacija koristi se za procenu efikasnosti algoritama.

  - Definicija 1:Neka su $f:N\rightarrow N$ i $g:N\rightarrow N$ dve proizvoljne funkcije argumenta n. Kažemo da je $g(n)=O(f(n))$ ako postoje pozitivne konstante $c\in R$ i $N\in N$, takve da $\forall n\geq N$ važi:$g(n)\leq c\cdot f(n)$.



Takođe važi:

$$O(f(n)) + O(g(n)) = O(f(n) + g(n))$$
$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

Primer: $5n^2+ 8=O(n^2)$ jer je $5n^2+ 8\leq 11/2n^2$ za $n\geq 4$, tj.$c=11/2$, za $N= 4$.

Slika pokazuje odnos funkcija g(n) i f(n) pa sledi da g(n) ne raste brže od c·f(n). Oznaka O(f(n)) u stvari se odnosi na klasu funkcija, tako da je $g(n)=O(f(n))$ drugi zapis za $g(n)\in O(f(n))$. Uočimo i da je O(1) oznaka za klasu ograničenih funkcija.

Oznake O,Ω,Θ

- Dakle, kada je potrebno dokazati da jedna funkcija pripada nekoj klasi funkcija, traže se konstante c i N da bude zadovoljena nejednakost. Prvo se traži c, a potom dovoljno veliko N. Obično postoji veliki (neograničen) broj kombinacija c i N pomoću kojih se može dokazati tvrdjenje.
- Funkcije se smeštaju u različite klase u zavisnosti od njihove brzine rasta.
- U prethodnom primeru se pokazuje da funkcija $5n^2+ 8$ ne raste brže od $6 \cdot n^2$. Vidimo da nam multimplikativne konstante ne igraju bitnu ulogu, paćemo uvek umesto $O(a \cdot n+b)$ koristiti $O(n)$, gde su a i b proizvoljne konstante (isto važ i za $O(n^2)$,$O(\log n)$ itd).
- Osnov alogartima nam nije bitna, što sledi iz jednakosti $\log_a n=\log_a(b^{\log_b n}) =\log_b n \cdot \log_a b=\log_b n \cdot c$, gde $\log_a b$ menjamo oznakom c za konstantu.

Broj koraka

- Svaku od osnovnih operacija (sabiranje, oduzimanje, množenje, deljenje, povećanje za jedan, dodela, poredenje, šifrovanje) ćemo radi jednostavnosti smatrati jednim korakom.Veliko O notacija je nastala za potrebe matematike, ali se danas dosta koristi i u algoritmici da bi se izrazila procena zauze ća memorije ili vreme izvršavanja algoritma.

- Postavlja se pitanje da li je opravdano koristiti je zbog zanemarivanja multiplikativnih konstanti ali sledeći razlozi govore u prilog njenom korišćenju:

  - 1. jednostavna je za korišćenje.
  - 2. konstantni faktori se mogu zanemarivati jer se prilikom konkretne implementacije pseudokoda brojevi koraka menjaju u zavisnosti od toga koji se programski jezik koristi, koji programer piše kod, itd.
  - 3. iako ne oponaša efikasnost na malim dimenzijama, ona sa velikom preciznošću pokazuje efikasnost algoritma kada veličina ulaza (broj n) postane velika. To i jeste interesantno jer se na malim ulazima svi algoritmi brzo izvršavaju.

Klase složenosti

- Cilj nam je da za svaki problem napišemo što efikasniji alogritam i procenimo njegovu složenost u O notaciji. Algoritmi se grubo mogu podeliti u sledeće klase složenosti (n je veličina ulaza):
  - **konstantna složenost (O(1)).** Ovo znači da algoritam uopšte ne zavisi od dimenzije ulaza n. Idealan slučaj koji se retko javlja u praksi (npr. pristup prvom elementu liste).
  - **sublinearna složenost** (npr.O(logn),O(√n)). Ovo je klasa vrlo efikasnih algoritama kod kojih je vreme izvršavanja manje od vremena potrebnog da se pročita ulaz (O(n)). Primer je binarna pretraga.
  - **linearna složenost (O(n)).** Ovo su efikasni algoritmi. Vreme izvršavanja je u rangu vremena potrebnog da se pročita ulaz. Primer je linearna pretraga.
  - **superlinearna složenost** (npr.O(nlogn),O(n^2)). Ovi algoritmi se smatraju dovoljno dobrim za implementaciju na računaru.
  - **eksponencijalna složenost** (npr.O(2^n),O(n^n)). Njihova implementacija je nepraktična za veće dimenzije ulaza n.Pogledati grafike prethodnih funkcija.

Uobičajeni tipovi složenosti algoritama.

´Većina interesantnih algoritama imaće složenost proporcionalnu nekoj od sledećih funkcija:

- 1 Ovo označava konstantan broj operacija, nezavisno od veličine niza ulaznih podataka

- logn Ova se složenost javlja kod algoritama koji problem mogu rešiti tako da ga transformišu u jednostavniji smanjujući broj elemenata za fiksan odnos. Na primer, potraga za zadatim elementom u nekoj sortiranoj listi. Algoritmi s ovim rastom su vrlo prihvatljivi, jer se složenost pove ćava samo za malu konstantu u slučaju udvostručavanja broja ulaznih podataka:log(2n) =logn+log 2.

- n Složenost je proporcionalna broju ulaznih podataka kod algoritama u kojima se odredjena količina posla mora napraviti nad svakim elementom ulaznog niza. To je tipična situacija kad algoritam zahteva unos ili ispis n podataka.

- nlogn Ova se složenost javlja u algoritmima koji za svaki ulazni podatak definišu podalgoritam rešiv u logaritamskom vremenu. Mnogi algoritmi sortiranjai mat će upravo takvo ponašanje. Za umerene vrednosti od n ovakav se algoritam ponaša gotovo kao linearan, logn sporo raste u odnosu prema veličini broja n. Nazvaćemo ovakve algoritme loglinearnim.

Uobičajeni tipovi složenosti algoritama

- n^2 Kvadratna zavisnost sa ulaznim podatcima daje algoritam koji je efikasan samo za umerene vrednosti broja n. Dvostruko veći niz ulaznih podataka zahteva četverostruko više vremena za rešavanje. Ovi se algoritmi javljaju, naprimer, prolazom kroz dvostruku petlju.

- n^α Razni algoritmi imaće ovu složenost za neku vrednost od α. Parametar α ne mora biti prirodan broj. Kažemo da algoritmi imaju potencijalni rast. Ako je α <1, algoritmi su brži od linearnog. Za α >1 oni su sporiji od linearnog, ali i od loglinearnog, za iole veći broj n. Ako je n celobrojan,govorimo još da su ovi algoritmi sa polinomijalnim rastom.

- a^n Ovde se pretpostavlja da je a>1, jer složenost mora rasti sa veličinom.Najčešće je a=2. Ovi algoritmi uporabljivi su samo za početne vrednosti broja n.

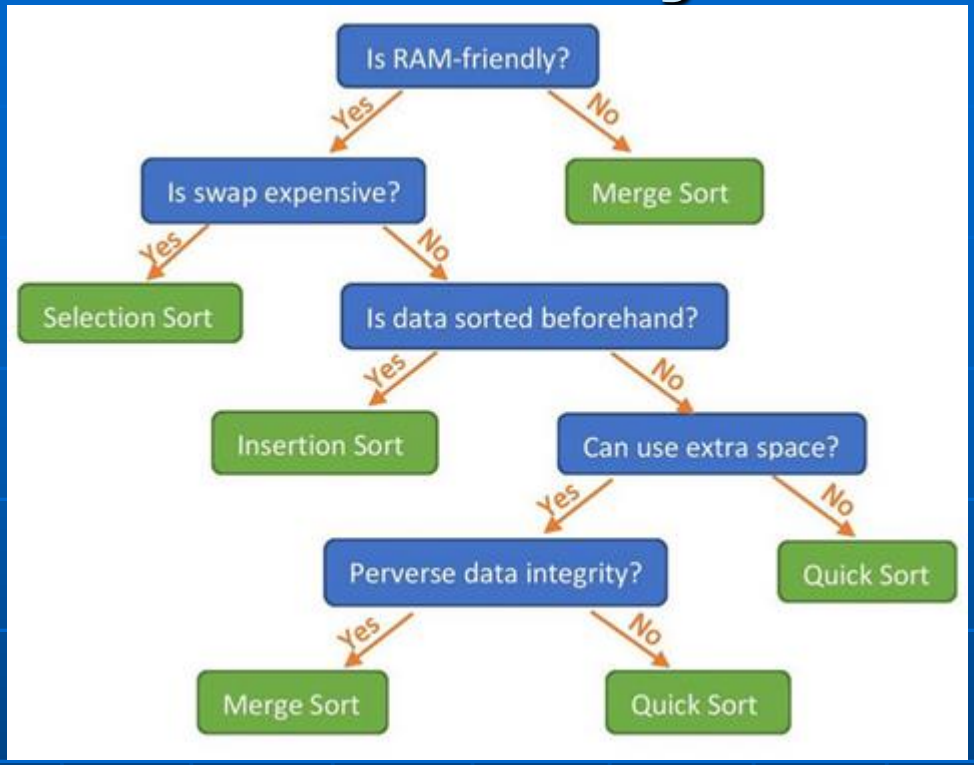
Sortiranje

- **Sortiranje je jedan od fundamentalnih zadataka u računarstvu.** Sortiranje podrazumeva uređivanje niza u odnosu na neko linearno uređenje (npr. uređenje niza brojeva po veličini — rastuće ili opadajuće, uređivanje niza niski leksikografski ili po dužini, uređivanje niza struktura na osnovu vrednosti nekog polja i slično).

- Mnogi zadaci nad nizovima se mogu jednostavnije rešiti u slučaju da je niz sortiran (npr. pretraživanje se može vršiti binarnom pretragom).

- Postoji više različitih algoritama za sortiranje nizova. Neki algoritmi su jednostavni i intuitivni, dok su neki kompleksniji, ali izuzetno efikasni. Najčešće korišćeni algoritmi za sortiranje su:

Sortiranje



- Postoji puno algoritama za sortiranje, ali ovde ću razmotriti najčešće korišćene, jednostavne za implementaciju algoritme, naime sortiranje odabira, sortiranje umetanjem, sortiranje spajanjem i brzo sortiranje.
- Dakle, na slici je prikazano stablo odluka kako bi rešavali problem izbora pravog algoritma za sortiranje sa obzirom na podatke i situacije. Sortiranje je osnovni gradivni element na kojem se grade mnogi drugi algoritmi. Razumevanje kako algoritmi za sortiranje u Python, C, Java... jeziku funkcionišu iza scene predstavlja osnovni korak ka primeni ispravnih i efikasnih algoritama koji rešavaju problema iz stvarnog sveta.

Sortiranje

U ovom delu biće govora o:

- Kako funkcionišu različiti algoritmi za sortiranje u Python's, C i Java jeziku i kako se oni upoređuju pod različitim okolnostima
- Kako odredjena jezički ugrađena funkcionalnost sortiranja radi iza kulisa
- Kako se različiti koncepti računarstva poput rekurzije i podele i osvajanja primenjuju na sortiranje
- Kako izmeriti efikasnost algoritma koristeći Big O notaciju i neke druge modue jezika

Sortiranje je jedan od najtemeljitijih algoritama u računarstvu.

Sortiranje

- Sortiranjem možemo rešiti širok spektar problema:
  - ✓ Pretraživanje: Traženje predmeta na listi deluje mnogo brže ako je lista sortirana.
  - ✓ Izbor: Izbor stavki sa liste na osnovu njihovog odnosa sa ostatkom stavki lakše je pomoću sortiranih podataka. Na primer, pronalaženje k-te najveće ili najmanje vrednosti ili pronalaženje medijane vrednosti liste je mnogo lakše kada su vrednosti u rastućem ili silaznom redosledu.
  - ✓ Duplikati: Pronalaženje duplikata vrednosti na listi može se izvršiti vrlo brzo kada se lista sortira.
  - ✓ Distribucija: Analiza frekvencijske raspodele stavki na listi je vrlo brza ako se lista sortira. Na primer, pronalaženje elementa koji se pojavljuje najčešće ili najmanje često relativno je jednostavno sa sortiranom listom.
- Izlaz bilo kojeg algoritma za sortiranje mora zadovoljavati dva uslova:
- 1. Izlaz nije u opadajućem redosledu (svaki element nije manji od prethodnog elementa prema željenoj ukupnoj narudžbi);
- 2. Izlaz je permutacija (preuredjivanje, ali zadržavajući sve izvorne elemente) ulaza.

Klasifikacija algoritama

Algoritmi sortiranja često su klasificirani prema:

- Kompjuterska složenost (najgore, prosečno i najbolje ponašanje) u smislu veličine liste (n). Za tipično serijske algoritme sortiranja dobro ponašanje je O(nlog n), sa paralelnim sortiranjem $O(\log^2 n)$, a loše ponašanje $O(n^2)$. Optimalno paralelno sortiranje je O(logn). Algoritmi sortiranja utemeljeni na usporedjivanju trebaju najmanje popredjenja Ω (n log n) za većinu ulaznih podataka.
- Računarska složenost (za algoritme "na mestu").
- Korišćenje memorije (i korišćenje ostalih računarskih resursa). Konkretno, neki algoritmi za sortiranje nalaze se "na mestu".
- Rekurzija. Neki su algoritmi rekurzivni ili nerekurzivni, dok drugi mogu biti I jedno I drugo (npr. Sortiranje spajanjem).
- Stabilnost: stabilni algoritmi za sortiranje održavaju relativni redosled zapisa sa jednakim ključevima (tj. Vrednostima).

Klasifikacija algoritama

- Bez obzira jesu li ili ne, oni su u okviru soritiranja poredjenjem. Sortiranje poredjenjem ispituje podatke samo upoređujući dva elementa sa operatorom poredjenja.

- *Opšta metoda:* umetanje - insertion, razmena - exchange, odabir - selection, spajanje - merging, itd. Razmena - exchange sort uključuje Bubble sort i Quick sort. Odabir – selection sort uključuje Shaker sort i Heapsort.

- *Prilagodljivost:* Bez obzira dali predodređenost ulaza ili ne utiče na vreme rada. Algoritmi koji uzimaju ovo u obzir se kaže da su prilagodljivi.

- *Stabilni algoritmi* za sortiranje ponavljaju elemente istim redosledom kojim se pojavljuju na ulazu. Kada sortiratmo neke vrste podataka, samo se deo podataka ispituje pri određivanju redosleda sortiranja. Stabilnost je važna iz sledećeg razloga: recimo da se na web stranici dinamički razvrstavaju studentski zapisi koji se sastoje od imena i odjeljenja, prvo po imenu, a zatim po odeljku klase u drugoj operaciji. Ako se koristi stabilni algoritam sortiranja u oba slučaja, operacija sort-by-class-section operation neće promeniti redosled imena; sa nestabilnom razvrstavanjem, to bi moglo biti sortiranje po klasi pa onda po imenu.

Algoritamska Analiza

- Efikasnost algoritma može se analizirati u dve različite faze, pre primene i nakon primene. One su sledeće:

- **A Priori Analysis** - Ovo je teorijska analiza algoritma. Efikasnost algoritma meri se pretpostavljanjem da su svi drugi faktori, na primer, brzina procesora, konstantni i nemaju uticaja na primenu.

- **A Posterior Analysis** - Ovo je empirijska analiza algoritma. Odabrani algoritam je implementiran pomoću programskog jezika. Ovo se zatim izvršava na ciljanoj računarskoj mašini. U ovoj analizi prikupljaju se stvarni statistički podaci poput potrebnog vremena i prostora.

Algoritamska i vremenska kompleksnost

Pretpostavimo da je X algoritam, a n veličina ulaznih podataka, vreme i prostor koje algoritam X koristi su dva glavna faktora koja odlučuju o efikasnosti X.

- **Faktor vremena** - vreme se meri brojanjem broja ključnih operacija kao što su poređenja u algoritmu za sortiranje.
- **Faktor prostora** - Prostor se meri brojanjem maksimalnog memorijskog prostora potrebnog algoritmu.
- **Složenost algoritma** f(n) data je vremenom rada i/ili prostora za skladištenje potrebnih algoritmu u termu n kao veličini ulaznih podataka.
- **Vremenska složenost algoritma** predstavlja količinu vremena potrebnog algoritmu da se izvrši do kraja. Vremenski zahtevi mogu se definisati kao numerička funkcija T (n), pri čemu se T (n) može meriti brojem koraka, pod uslovom da svaki korak troši konstantno vreme.
- **Efikasnost i tačnost algoritama** moraju se analizirati kako bi se uporedili i izabrali odredjeni algoritmi za određene scenarije. Proces izrade ove analize naziva se asimptotska analiza. Odnosi se na izračunavanje vremena izvođenja bilo koje operacije u matematičkim jedinicama računanja.

Složenost prostora

- **Složenost prostora algoritma** predstavlja količinu memorijskog prostora koju algoritam zahteva u svom životnom ciklusu. Prostor potreban algoritmu jednak je zbiru sledeće dve komponente :

  - **Fiksni deo** koji je prostor potreban za čuvanje određenih podataka i promenljivih, koji su nezavisni od veličine problema. Na primer, jednostavne promenljive i konstante koje se koriste, veličina programa itd.

  - **Promenljivi deo** je prostor potreban promenljivim, čija veličina zavisi od veličine problema. Na primer, dinamičko dodeljivanje memorije, prostor steka rekurzije itd.

Efikasnost i tačnost algoritama

- *Efikasnost i tačnost algoritama* moraju se analizirati kako bi se uporedili i izabrali određeni algoritam za određene scenarije. Proces izrade ove analize naziva se Asimptotska analiza. Odnosi se na izračunavanje vremena izvođenja bilo koje operacije u matematičkim jedinicama računanja.

- Na primer, vreme izvođenja jedne operacije izračunava se kao f(n), a možda se za drugu operaciju računa kao $g(n^2)$. To znači da će se vreme izvođenja prve operacije linearno povećavati sa povećanjem n, a vreme izvođenja druge operacije će se eksponencijalno povećavati kada se n poveća. Slično tome, vreme izvođenja obe operacije biće približno isto ako je n značajno malo.

- Obično vreme potrebno algoritmu spada u tri vrste:

  - Best Case – Minimalno vreme potrebno za izvršavanje
  - Average Case – Srednje vreme potrebno za izvršavanje
  - Worst Case – Maksimalno vreme potrebno za izvršavanje.

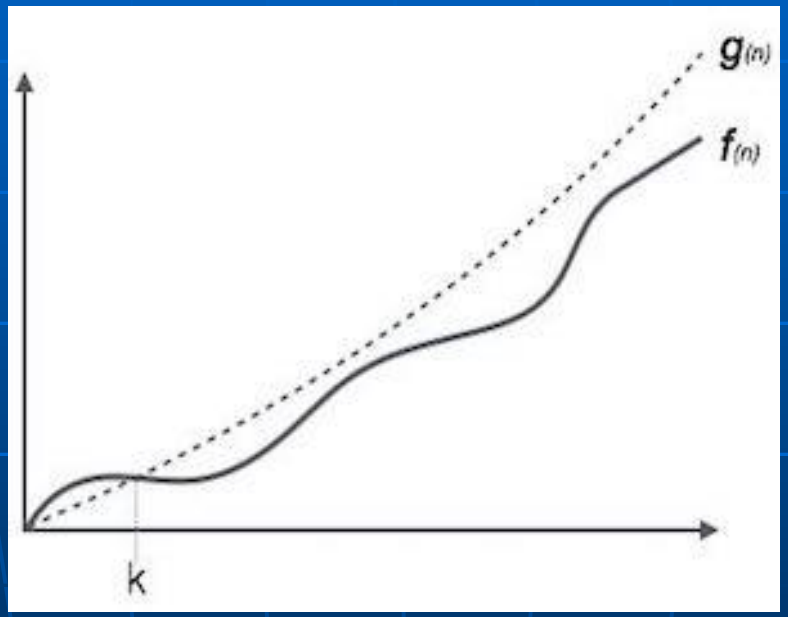
Asimptotska notacija

- Slede najčešće korišćeni asimptotski zapisi za izračunavanje složenosti radnog vremena algoritma.

  - O Notation
  - Ω Notation
  - θ Notation

Notacija veliko O

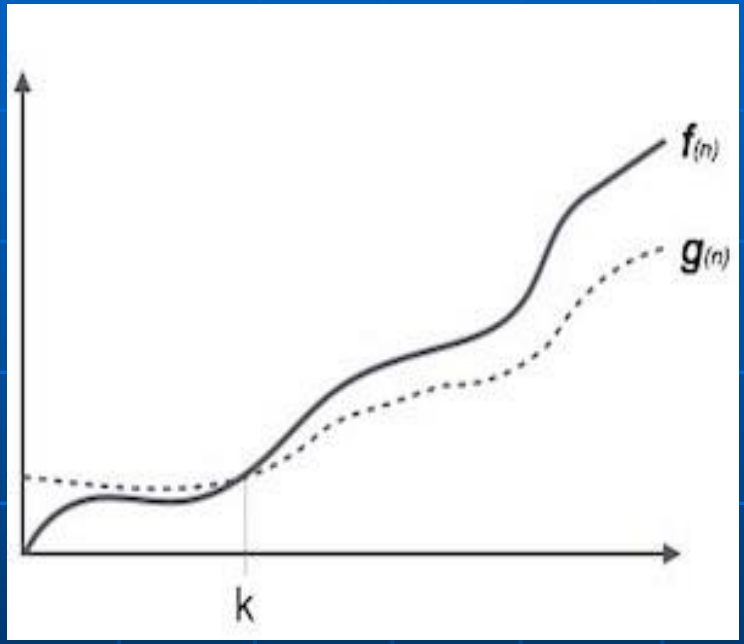
- Notacija O (n) je formalni način da se izrazi gornja granica vremena rada algoritma. Ona meri najgoru vremensku složenost ili najduže vreme koje može potrajati da algoritam završi sa radom.



- Na primer za funkciju ***f*(n)**
- **O(f(n)) = { g(n) : there exists c > 0 and n_0 such that f(n) ≤ c.g(n) for all n > n_0. }**

Omega Notacija, Ω

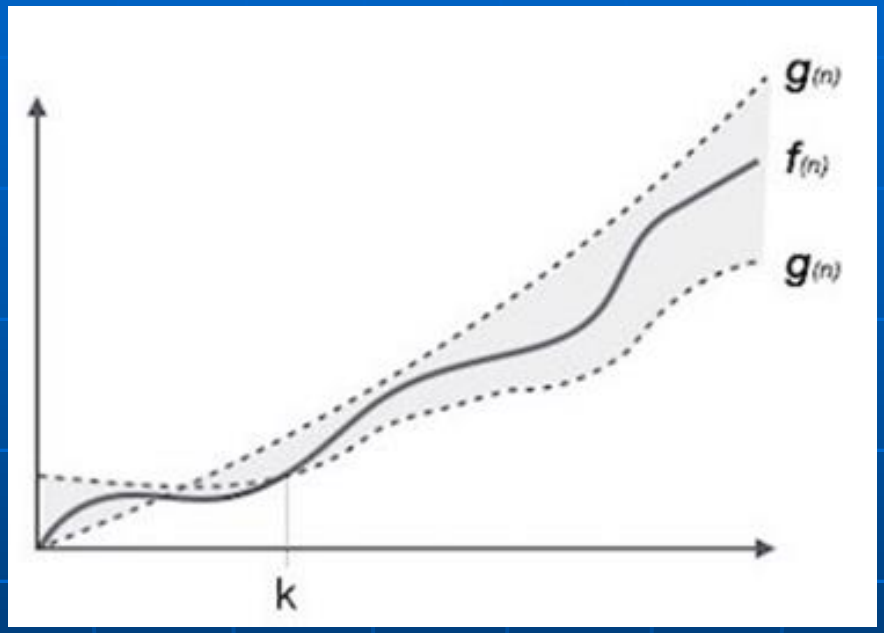
- Notacija Ω(n) je formalni način da se izrazi donja granica vremena rada algoritma. Ona meri najbolju vremensku složenost ili najbolje vreme koje može potrajati da algoritam završi sa radom.



- Na primer za funkciju ***f*(n):**
- **Ω(f(n)) ≥ { g(n) : there exists c > 0 and n_0 such that g(n) ≤ c.f(n) for all n > n_0. }**

Theta Notacija, θ

- Notacija θ(n) je formalni način da se brzo obrade i *lower bound* i *upper bound* vremena algoritamskog izvršavanja. Reprezentuje se kao sledeće:



- Oznaka θ(n) je formalni način da se izraze i donja i gornja granica vremena rada algoritma. Predstavljen je na sledeći način –

- **θ(f(n)) = { g(n) if and only if g(n) = O(f(n)) and g(n) = Ω(f(n)) for all n > n0. }**

Lista komparacije

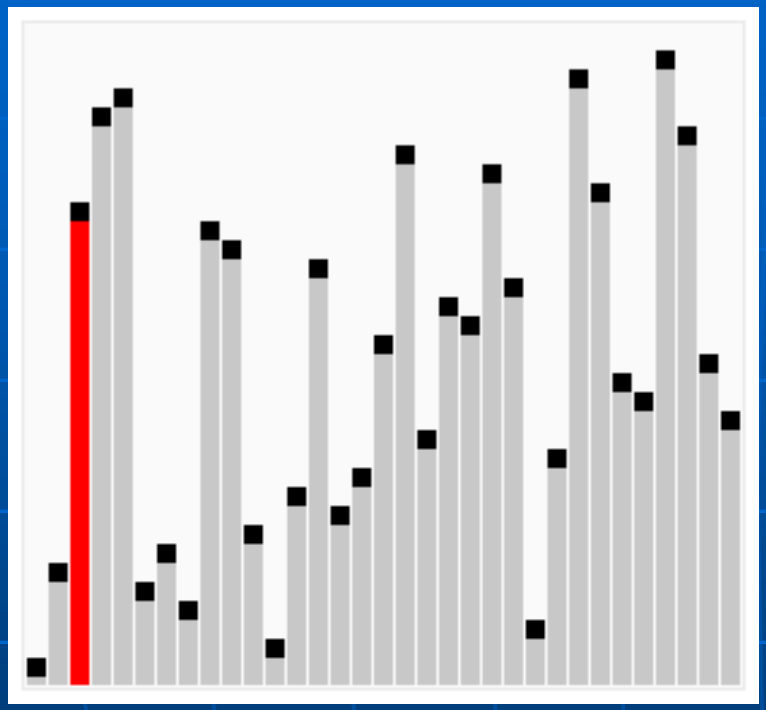
Comparison sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[7]
In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[8]
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have d inversions.
Block sort	n	$n \log n$	$n \log n$	1	Yes	Insertion & Merging	Combine a block-based $O(n)$ in-place merge algorithm[9] with a bottom-up merge sort.
Quadsort	n	$n \log n$	$n \log n$	n	Yes	Merging	Uses a 4-input sorting network.[10]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes n comparisons when the data is already sorted or reverse sorted.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space or when using linked lists.[11]
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes n comparisons when the data is already sorted or reverse sorted.
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes.

Sortiranje

- Redjanje mehurom (*bubble sort*)
- Redjanje izborom (*selection sort*)
- Redjanje umetanjem (*insertion sort*)
- Redjanje spajanjem (*merge sort*)
- Brzo redjanje (*quicksort*)
- Šelovo redjanje (*Shell sort*)
- Redjanje hrpom (*heapsort*)

Sortiranje-Redjanje mehurom (*bubble sort*)



Ovo je jedna od najjednostavnijih metoda sortiranja koja je efikasna samo za relativno mali broj elemenata. Za veći broj elemenata ova metoda je prespora. Stoga se vrlo retko upotrebljava osim za edukacijske svrhe.

Bubble sort algoritam u svakom prolazu kroz niz poredi uzastopne elemente, i razmenjuje im mesta ukoliko su u pogrešnom poretku. Prolasci kroz niz se ponavljaju sve dok se ne napravi prolaz u kome nije bilo razmena, što znači da je niz sortiran.

$\underline{4},\underline{2},6,3,1,5$	$2,3,1,\underline{4},\underline{5},6$
$2,\underline{4},\underline{6},3,1,5$	$\underline{2},\underline{3},1,4,5,6$
$2,4,\underline{6},\underline{3},1,5$	$2,\underline{3},\underline{1},4,5,6$
$2,4,3,\underline{6},\underline{1},5$	$2,1,\underline{3},\underline{4},5,6$
$2,4,3,1,\underline{6},\underline{5}$	$\underline{2},\underline{1},3,4,5,6$
$\underline{2},\underline{4},3,1,5,6$	$1,\underline{2},\underline{3},4,5,6$
$2,\underline{4},\underline{3},1,5,6$	$\underline{1},\underline{2},3,4,5,6$
$2,3,\underline{4},\underline{1},5,6$	$1,2,3,4,5,6$

Recimo da tim algoritmom želimo poredati elemente u listi A {4,2,6,1,3,5}. Onda su nam potrebni sledeći koraci (rezultati su zapisani po stupcima zbog uštede u prostoru)

Sortiranje-Redjanje mehurom (*bubble sort*)

Prvi prolaz:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Algoritam uporedjuje prva dva elementa, i procenjuje 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), da li je 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), da li je 5 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Pošto su elementi već u redu order (8 > 5), algoritam ih ne svapuje.

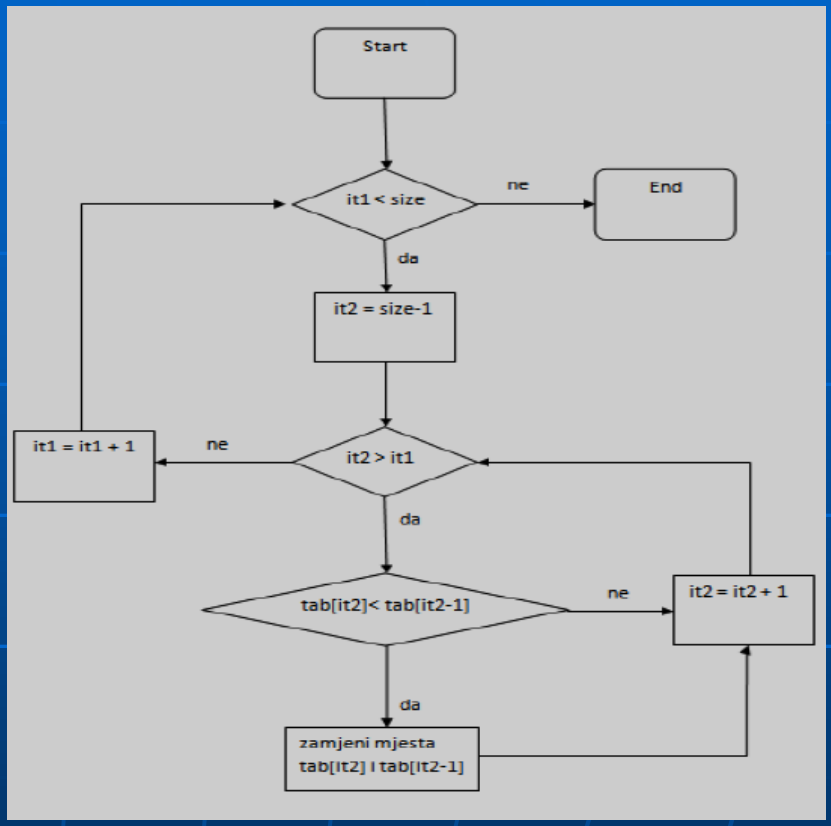
Drugi prolaz:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), da li je 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Polje je već sortirano, ali algoritam nezna da li je kompletan? Zbog toga algoritam prolazi još kroz jedan prolaz bez svapovanja da bi bio siguran da je sad OK.

Treći prolaz:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Sortiranje-Redjanje mehurom (*bubble sort*)

- Sortiranje mehurom (engl. Bubble sort), ponekad pogrešno nazivan sinking sort, je jednostavan algoritam za sortiranje koji radi tako što više puta prolazi kroz niz koji treba biti sortiran i uspoređuje se svaki par susjednih elemenata. Elementi zamjenjuju mjesta ako su napisani pogrešnim redoslijedom. Prolaz kroz niz se ponavlja sve dok se ne izvrše sve potrebne zamjene, što ukazuje da je niz sortiran.
- Algoritam je dobio ime zbog načina na koji najmanji element „bubble" dolazi na početak niza. Pošto se samo uspoređuju elementi, ovo je komparativno sortiranje.



Složenost Bubble sorta

- Bubble sort vrsta sortiranja ima najgoru složenost koja je data sa $O(n^2)$, gde je n broj elemenata koji se sortiraju. Postoji puno drugih algoritama za sortiranje koji imaju znatno bolju složenost O(n log n) od samog Bubble sort-a. Čak i drugi algoritimi sortiranja složenosti $O(n^2)$, kao što je insertion sort, imaju tendenciju da
- budu što bolji i da imaju što bolje performanse od bubble sort-a. Dakle, bubble sort nije toliko praktičan algoritam za sortiranje ako je n veliki, međutim u našem slučaju taj n je 5 tako da ne predstavlja veliku razliku korisiti bubble ili insertion sort.

- Jedina značajna prednost koju ima bubble sort za razliku od drugih vrsta sortiranja, čak i quicksorta, ali ne i insertion sorta, je sposobnost otkirvanja da je sortirani niz efikasno ugrađen u algoritam. Složenost bubble sorta kad imamo na ulazu sortirani niz (što je ujedno i najbolji slučaj) je O(n) .

- Pozicije elemenata u bubble sortu igraju veliku ulogu u određivanju složenosti samog algoritma. Veliki elementi na početku niza ne predstavljaju problem jer se brzo zamene. Mali elementi pri kraju niza se kreću na početak veoma sporo. Zbog toga se ove vrste elemenata respektivno nazivaju zečevi i kornjače.

Implementacije bubble sort algoritma u Python_u

```
def bubble_sort(array):
    n = len(array)

    for i in range(n):
        # Create a flag that will allow the function to
        # terminate early if there's nothing left to sort
        already_sorted = True

        # Start looking at each item of the list one by one,
        # comparing it with its adjacent value. With each
        # iteration, the portion of the array that you look at
        # shrinks because the remaining items have already
been
        # sorted.
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                # If the item you're looking at is greater than its
                # adjacent value, then swap them
                array[j], array[j + 1] = array[j + 1], array[j]

                # Since you had to swap two elements,
                # set the `already_sorted` flag to `False` so the
                # algorithm doesn't finish prematurely
                already_sorted = False

        # If there were no swaps during the last iteration,
        # the array is already sorted, and you can terminate
        if already_sorted:
            break
    return array
```

Pošto ova implementacija sortira niz u rastućem redosledu, svaki korak „ bubbles“ stavlja najveći element na kraj niza. To znači da svaka iteracija preduzima manje koraka od prethodne iteracije, jer se sortira kontinuirano veći deo niza.

Petlje u redovima 4 i 10 određuju način na koji algoritam prolazi kroz listu. Treba obratiti pažnju na to kako *j* u početku ide od prvog elementa na listi do elementa neposredno pre poslednjeg.

Tokom druge iteracije *j* egzekutira od druge stavke od poslednje, zatim treće stavke od poslednje itd. Na kraju svake iteracije, krajnji deo liste će biti sortiran. Kako petlje napreduju, red 15 upoređuje svaki element sa susednom vrednošću, a red 18 ih zamenjuje ako su u pogrešnom redosledu. Ovo osigurava sortiranu listu na kraju funkcije.

Bubble Sort proces

Sada pogledajmo korak po korak šta se događa sa naprimer nizom [8, 2, 6, 4, 5] i kako algoritam napreduje:

Kod započinje upoređivanjem prvog elementa 8 sa susednim elementom 2. Sa obzirom da je 8> 2, vrednosti se zamenjuju, što rezultuje sledećim redosledom:

[2, 8, 6, 4, 5].

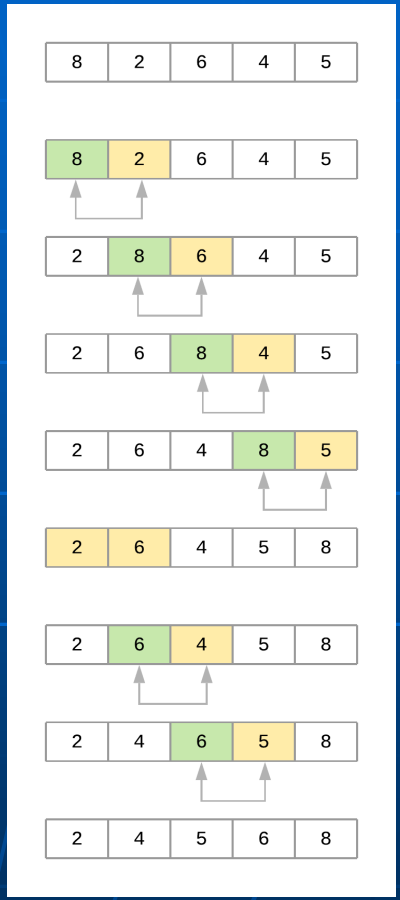
Algoritam zatim upoređuje drugi element 8 sa susednim elementom 6. Sa obzirom da je 8> 6, vrednosti se zamenjuju, što rezultuje sledećim redosledom:

[2, 6, 8, 4, 5].

Dalje, algoritam upoređuje treći element, 8, sa susednim elementom 4. Sa obzirom da je 8> 4, on takođe zamenjuje vrednosti, što rezultuje sledećim redosledom: [2, 6, 4, 8, 5].

Na kraju, algoritam upoređuje četvrti element, 8, sa susednim elementom 5, i takođe ih zamenjuje, što rezultira u [2, 6, 4, 5, 8]. U ovom trenutku, algoritam je izvršio prvi prolazak kroz listu (i = 0). Primetićemo kako se vrednost 8 pomerala od početne lokacije do tačnog položaja na kraju liste.

Drugi prolaz (i = 1) uzima u obzir da je poslednji element liste već pozicioniran i fokusira se na preostala četiri elementa, [2, 6, 4, 5]. Na kraju ovog prolaza, vrednost 6 pronalazi svoj tačan položaj. Treći prolazak kroz listu pozicionira vrednost 5 i tako dalje dok se lista ne sortira.



Vremena Bubble Sort implementacije

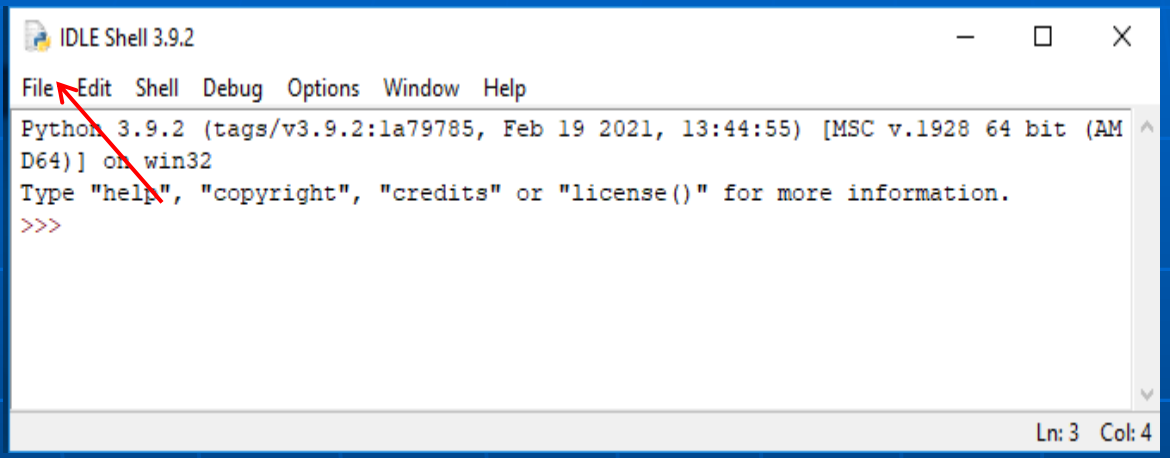
- Koristeći funkciju *run_sorting_algorithm(),* evo vremena koje je potrebno sortiranju bubble_sort da obradi niz sa deset hiljada elemenata. Red 8 zamenjuje ime algoritma, a sve ostalo ostaje isto: t se sortira.

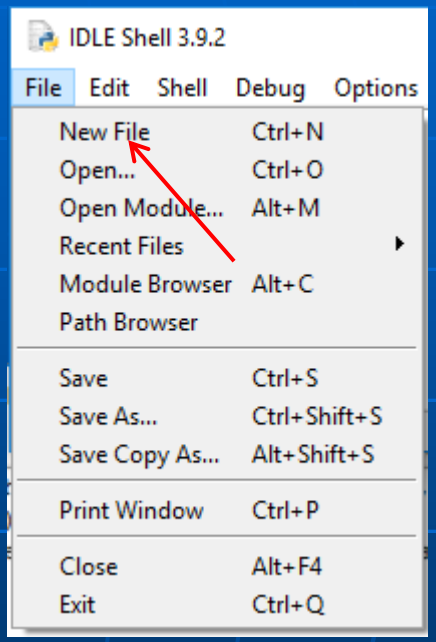
- if __name__ == "__main__":
- # Generate an array of `ARRAY_LENGTH` items consisting
- # of random integer values between 0 and 999
- array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
-
- # Call the function using the name of the sorting algorithm
- # and the array you just created
- run_sorting_algorithm(algorithm="bubble_sort", array=array)

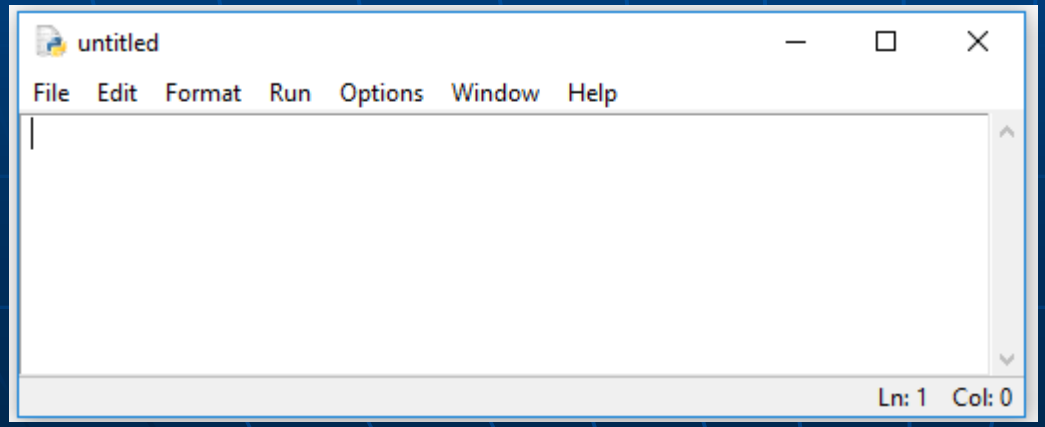
- IZLAZ:
- $ python sorting.py
- Algorithm: bubble_sort. Minimum execution time: 73.21720498399998

PRIMER 1. BS u Python_u:

- Za optimizaciju problema možemo implementirati Bubble Sort u Python programskom jeziku:

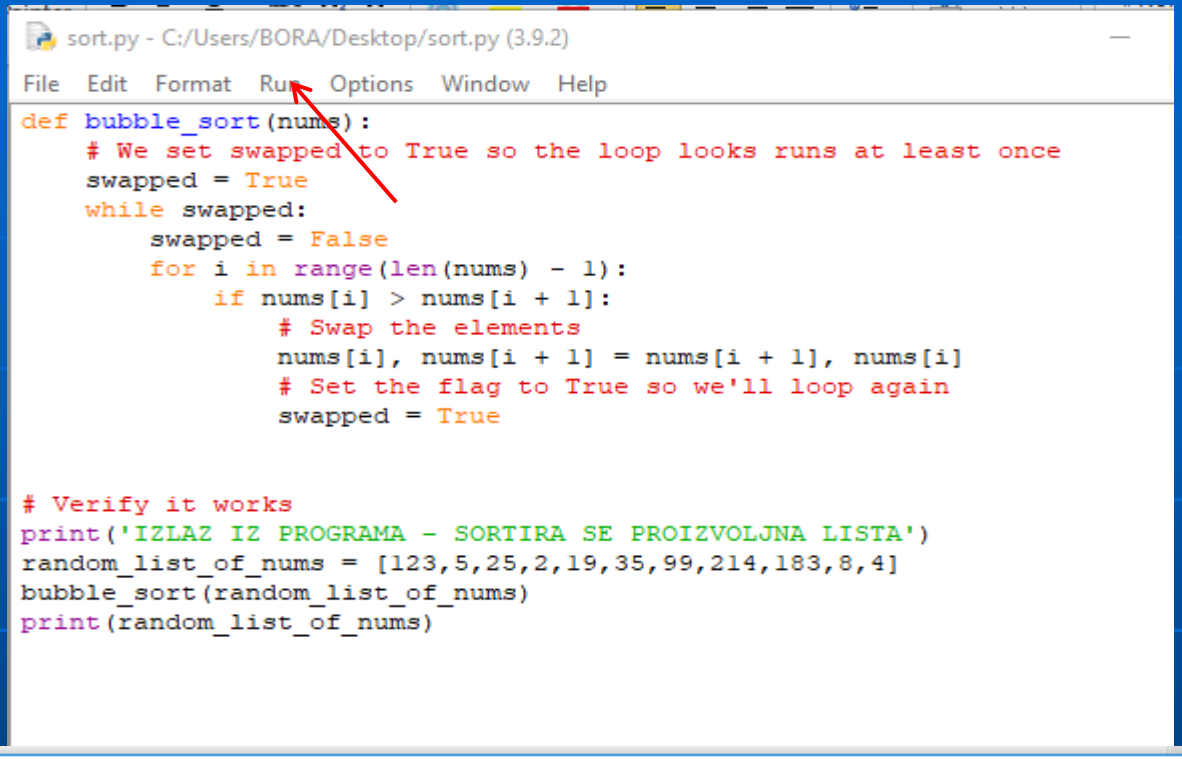


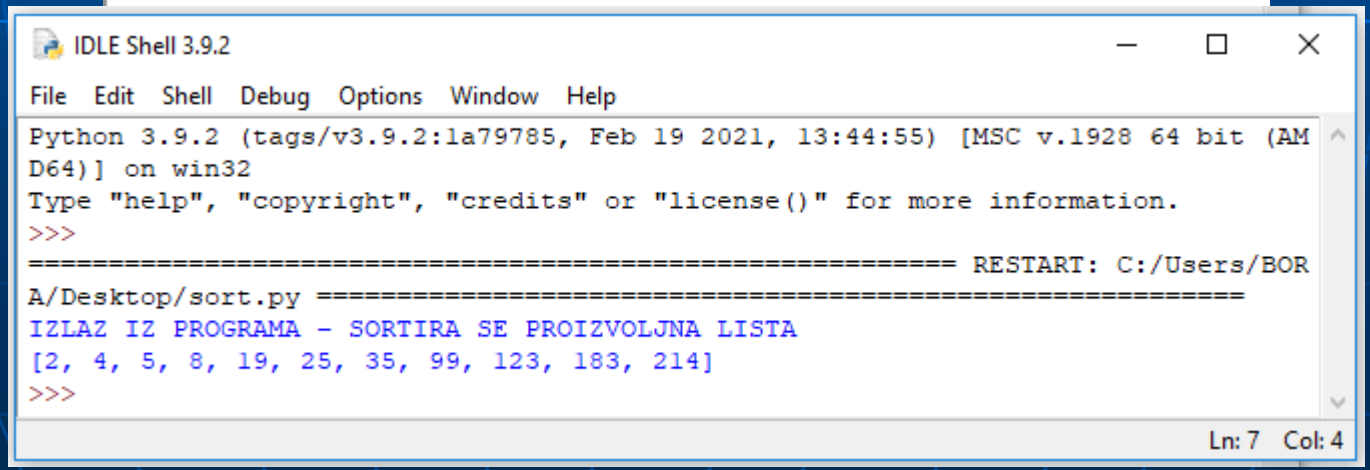




Obavezno treba formirati direktorijum na kome će se nalaziti fajlovi Python jezika: izvorni program i izvršna verzija. Naprimer PYTHON PROBA

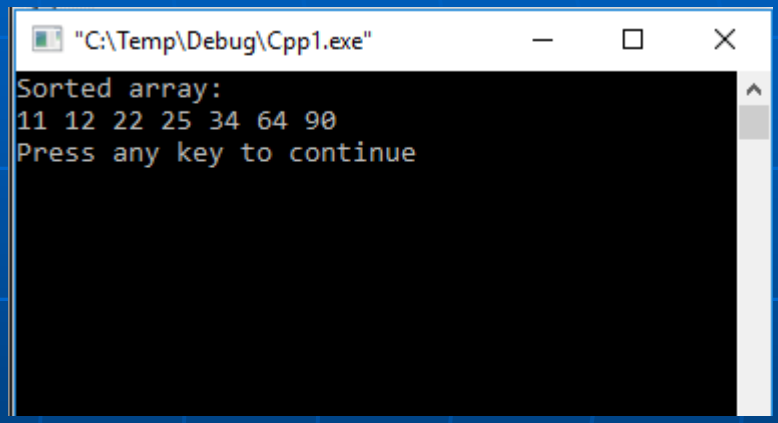
PRIMER 1. BS u Python_u:





PRIMER 1. BS u C_u:

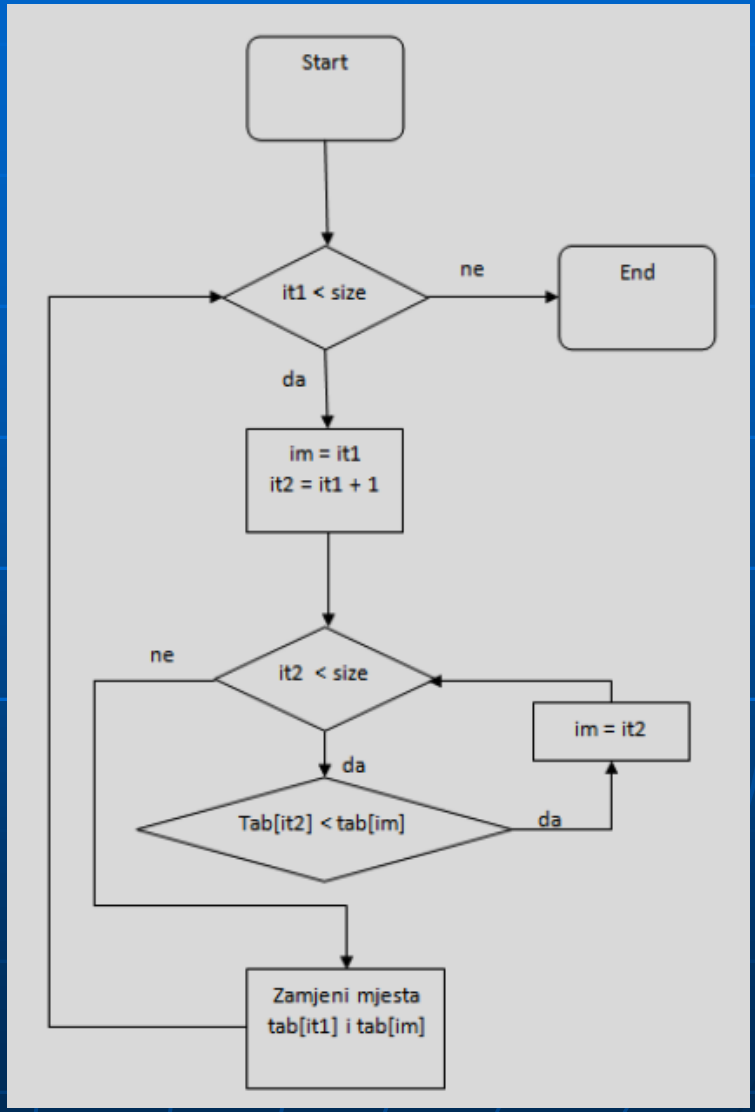
```
// C program for implementation of Bubble sort
#include <stdio.h>
  void swap(int *xp, int *yp)
{    int temp = *xp;
    *xp = *yp;
    *yp = temp;}
  // A function to implement bubble sort
void bubbleSort(int arr[], int n)
{   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);}
  /* Function to print an array */
void printArray(int arr[], int size)
{    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");}
  // Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```



Selection sort

- Selection sort algoritam deli uneseni niz u dva dela: podniz predmeta koji su već sortirani, koji je izgrađen sa leva na desno iz niza, a podniz preostalih predmeta će biti rešen tako da zauzme ostatak niza. U početku, sortirani podniz je prazan, a nesortiran niz je zapravo celi niz kojeg dovodimo na ulaz.

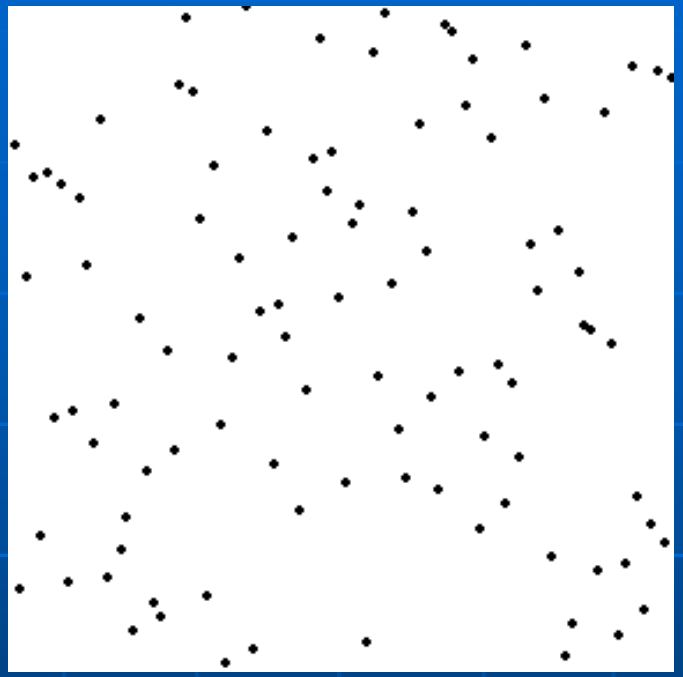
- Algoritam se izvršava na principu pronalaženja najmanjeg elementa u nerazvrstanom nizu kojeg smo doveli na ulaz, te kad ga pronađe zameni ga sa elementom koji je skroz levo u nesortiranom nizu (stavljajući ga u položaj koji je fiksiran i već sortiran), nakon toga pomiče se granica podniza za jedan element u desno jer je onaj prvi već sortiran. (Donald Knuth,1997)



Složenost Selection sorta

- Selection sort ima $O(n^2)$ vremensku složenost, što ga čini neučinkovitim za sortiranje sa velikim nizovima, i uglavnom je čak lošiji i od sličnog njemu insertion sort-a. Selection sort nije teško analizirati u odnosu na druge algoritme sortiranja.
- Odabere se najmanji element niza koji zahteva skeniranje svih n elemenata niza (to traje n - 1 uporedjivanja),
- zatim ga postavi na prvu poziciju. Pronalaženje sledećeg najmanjeg elementa koji zahteva skeniranje preostalih n - 1 elementa i
- tako dalje, za $(n - 1) + (n - 2) + ... + 2 + 1 = n (n - 1) /2 \in \Theta (n^2)$ uporedjivanja.
- Svaka od ovih poredjenja zahtieva jednu zamjenu za n - 1 elemenata (konačni element je već na mestu).

Sortiranje- Redjanje izborom (*selection sort*)



- **Algoritam „sortiranje selekcijom“ u svakoj iteraciji nalazi određeni element i stavlja ga na svoje mesto.**
- Najjednostaviji oblik algoritma je da u prvoj iteraciji pronađe najmanji element i postavi ga na prvo mesto, u drugoj iteraciji pronađe najmanji preostali element i stavi ga na drugo mesto, itd.
- Može se opisati u jednoj rečenici: „Ako niz ima više od jednog elementa, zameni početni element sa najmanjim elementom niza i zatim rekurzivno sortiraj rep (elemente iza početnog)“.

85	33	25	86	27	35	68	50	41	23	72	66	40	27	1	2

85	33	25	86	27	35	68	50	41	23	72	66	40	27	1	2

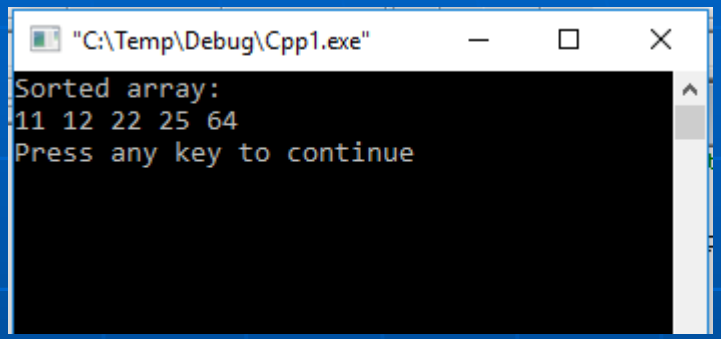
1	33	25	86	27	35	68	50	41	23	72	66	40	27	85	2

1	33	25	86	27	35	68	50	41	23	72	66	40	27	85	2

1	2	25	86	27	35	68	50	41	23	72	66	40	27	85	33

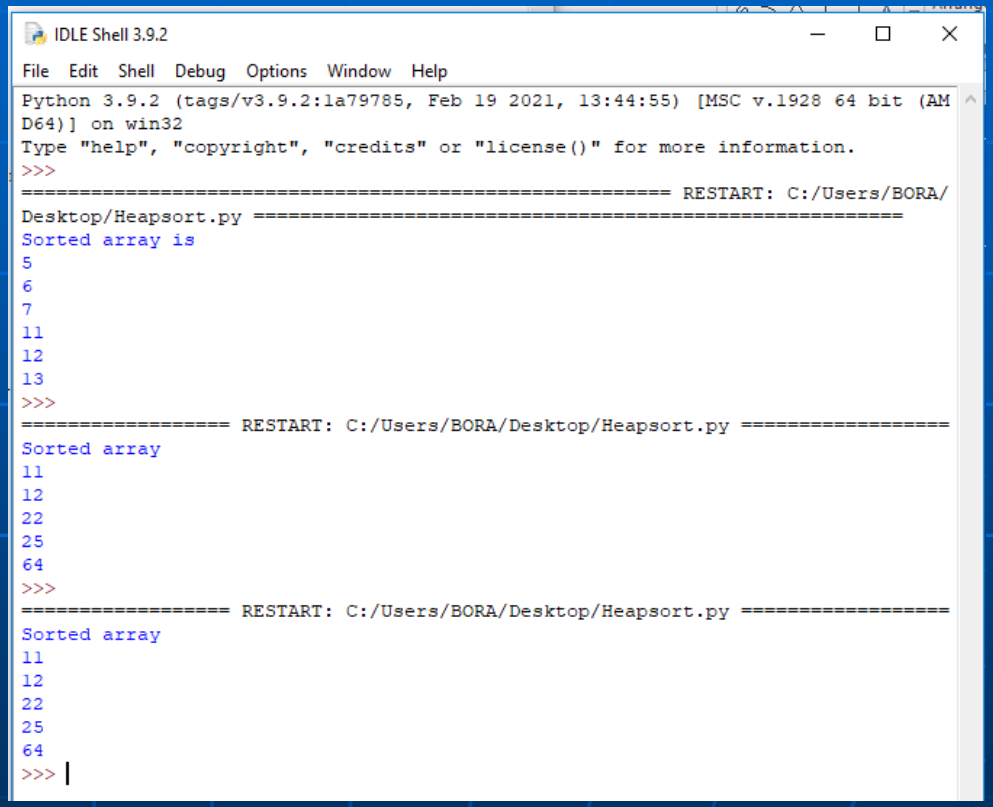
C program za SS

```
// C program for implementation of selection sort
#include <stdio.h>
 void swap(int *xp, int *yp)
{    int temp = *xp;
    *xp = *yp;
    *yp = temp;}
 void selectionSort(int arr[], int n)
{    int i, j, min_idx;
     // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    { // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;
          // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);  }}
 /* Function to print an array */
void printArray(int arr[], int size)
{    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");}
 // Driver program to test above functions
int main()
{  int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;}
```



Python program za SS

```
# Python program for implementation of Selection
# Sort
import sys
A = [64, 25, 12, 22, 11]

# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]

# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i]),
```



Insertion sort

- Sortiranje umetanjem (engl. Insertion sort) je jednostavan algoritam za sortiranje, koji gradi završni sortirani niz jednu po jednu stavku. Mnogo je manje efikasan na većim listama od mnogo složenijih algoritama kao što su quicksort, heapsort ili mergesort. (Sedgewick, 1983).

  Međutim sortiranje umetanjem ima svoje prednosti:

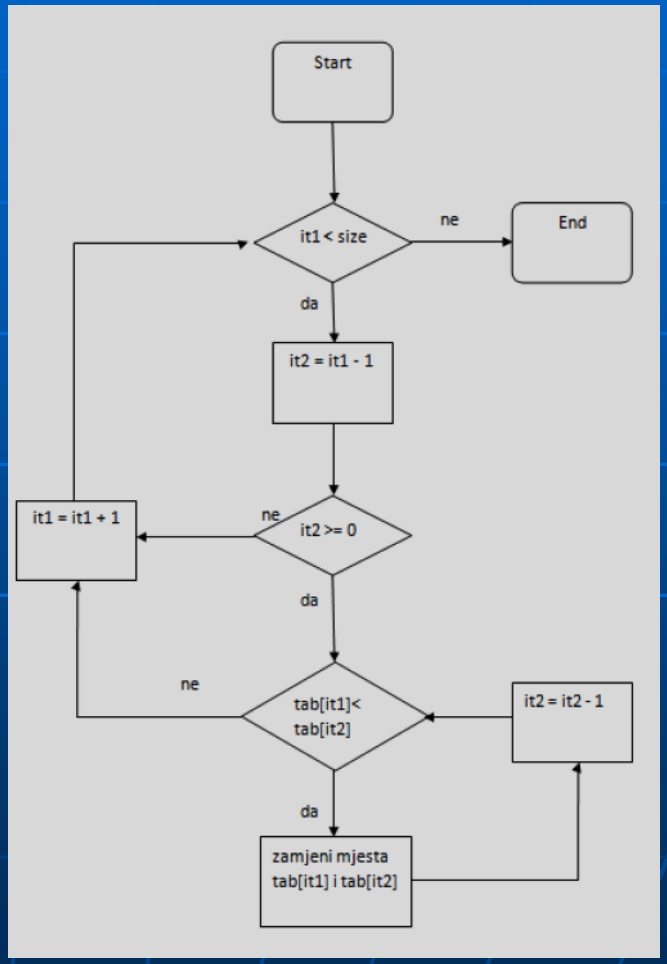
- Jednostavna primena
- Efikasan na malim skupovima podataka
- Prilagodljiviji za skupove podataka koji su već značajno sortirani: Vreme kompleksnosti je O(n+d), gde je d broj inverzija
- Efikasniji u praksi od većine drugih kvadratnih (tj. O (n^2)) algoritama, kao što su selection sort ili bubble sort
- Stabilan tj. ne menja relativni redosled elemenata sa jednakim vrednostima
- U mestu (tj. zahteva samo konstantan iznos O (1) dodatnog memorijskog prostora)
- Trenutan (online, može sortirati listu, odmah pri primanju)

Insertion sort

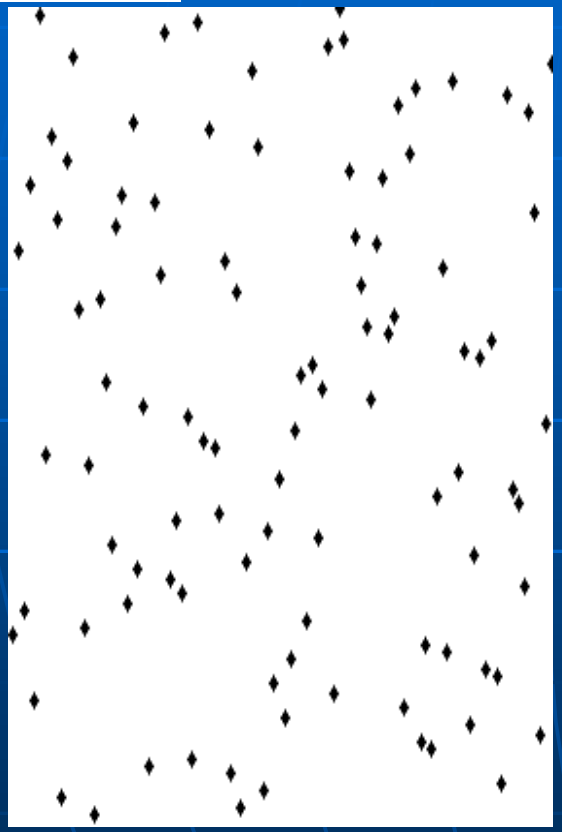
Najčešće varijante sortiranja umetanjem, koji radi sa nizovima, može se opisati kao:

1. Pretpostavimo da postoji funkcija Insert koja ubacuje vrednost u sortiranu sekvencu na početku niza. Počinje rad sa kraja sekvence i pomiče elemente jedno mesto udesno, dok ne nađe odgovarajuće mesto za novi element. Propratna pojava ove funkcije je uništavanje prvog sledećeg člana iz nesortiranih ulaznih elemenata.

2. Da bi se izvršilo sortiranje umetanjem, potrebno je uzimati „najleviji" element ulaznog niza i pozivati funkciju Insert. Ovim je osigurano da funkcija uvek upisuje preko „najlevijeg elementa" što je u stvari element koji se sortira.



Sortiranje- Redjanje umetanjem (*insertion sort*)



- ***Insertion sort* algoritam sortira niz tako što jedan po jedan element niza umeće na odgovarajuće mesto, u do tada sortirani deo niza.**
- Koncepcijski, postoje dva niza — polazni niz iz kojeg se uklanjaju elementi i niz koji čuva rezultat, u koji se dodaju elementi. Međutim, obično implementacije koriste memorijski prostor polaznog niza za obe uloge — početni deo niza predstavlja rezultujući niz, dok krajnji deo predstavlja preostali deo polaznog niza.
- Može se formulisati na sledeći način: „Ako niz ima više od jednog elementa, sortiraj rekurzivno sve elemente ispred poslednjeg, a zatim umetni poslednji u već sortirani prefiks.“

Nekaje početni raspored A={3,6,2,7,1,4,5}. Sortiranje počinje od drugog elementa i završava se poslednjim. U svakom koraku je potcrtan element koji ulazi na sortiranu listu:

3	<u>6</u>	2	7	1	4	5
3	6	<u>2</u>	7	1	4	5
2	3	6	<u>7</u>	1	4	5
2	3	6	7	<u>1</u>	4	5
1	2	3	6	7	<u>4</u>	5
1	2	3	4	6	7	<u>5</u>
1	2	3	4	5	6	7

Složenost Insertion sorta

- Najbolji slučaj je kada je unos već sortiran niz, u ovom slučaju sortiranje umetanjem ima linearno vreme rada (tj. O(n)). Tokom svake iteracije, prvi preostali element ulaznog niza se samo upoređuje sa zadnjim elementom sortiranog niza.

- Najjednostavniji najgori slučaj je kada je niz u obrnutom redosljedu. Skup svih najgorih slučajeva ulaza sastoji se od nizova gde je svaki element najmanji ili drugi najmanji od elemenata pre njega. U ovim slučajevima svaka iteracija unutrašnje petlje će proći kroz svaki sortirani element i pomaknuti ga, pre nego što ubaci sledeći element.

- To daje sortiranju umetanjem kvadratno vreme izvršavanja (O (n^2)). Prosečan slučaj ima takodje kvadratno vreme izvršavanja, što čini sortiranje umetanjem nepraktičnim za sortiranje većih nizova.

- Međutim, sortiranje umetanjem je jedan od najbržih algoritama za sortiranje vrlo malih nizova, čak brži od quicksort-a.

Implementiranje Insertion Sort u Python_u

- Evo primene u Python_u:

```
def insertion_sort(array):
  # Loop from the second element of the array until
  # the last element
    for i in range(1, len(array)):
   # This is the element we want to position in its
   # correct place
       key_item = array[i]
    # Initialize the variable that will be used to
    # find the correct position of the element referenced
    # by `key_item`
       j = i - 1
    # Run through the list of items (the left
    # portion of the array) and find the correct position
    # of the element referenced by `key_item`. Do this only
    # if `key_item` is smaller than its adjacent values.
       while j >= 0 and array[j] > key_item:
    # Shift the value one position to the left
    # and reposition j to point to the next element
    # (from right to left)
          array[j + 1] = array[j]
          j -= 1
    # When you finish shifting the elements, you can position
    # `key_item` in its correct location
      array[j + 1] = key_item
    return array
```

Za razliku od bubble sort, ova implementacija sortiranja umetanja gradi sortiranu listu gurajući manje stavke ulevo. Hajde da razjasnimo Insertion_sort () red po red:

Linija 4 postavlja petlju koja određuje stavku kei_it koju će funkcija postaviti tokom svake iteracije. Petlja započinje drugom stavkom na listi i ide sve do poslednje stavke.

Red 7 inicijalizuje stavku kei_item na tačku koju funkcija pokušava da postavi.

Red 12 inicijalizuje promenljivu koja će uzastopno ukazivati na svaki element levo od ključne stavke. To su elementi koji će se uzastopno upoređivati sa kei_item.

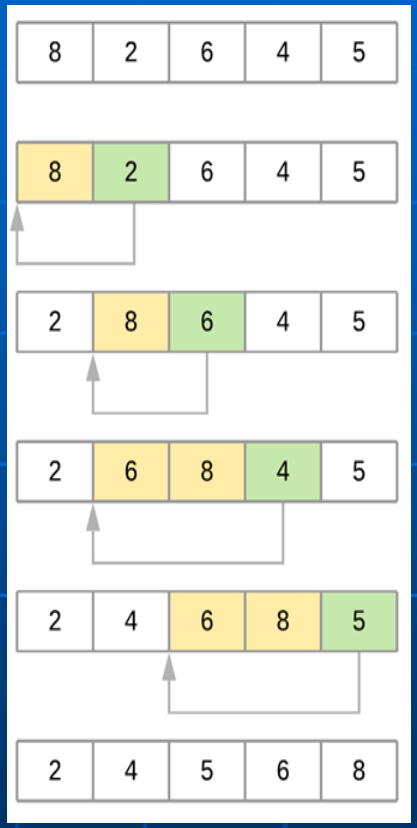
Red 18 upoređuje stavku kei_item sa svakom vrednošću sa njegove leve strane pomoću vhile petlje, pomerajući elemente kako bi se napravilo mesto za postavljanje kei_item.

Linija 27 postavlja stavku kei_item na svoje tačno mesto nakon što algoritam pomera sve veće vrednosti udesno.

I na kraju evo slike koja ilustruje različite iteracije algoritma pri sortiranju niza [8, 2, 6, 4, 5]:

Proces Insertion Sort

- Algoritam započinje sa kei_item = 2 i prolazi kroz podniz sa leve strane da bi pronašao tačnu poziciju za njega. U ovom slučaju, podniz je [8].
- Pošto je 2 <8, algoritam pomera element 8 za jedan položaj udesno. Rezultanta niza u ovom trenutku je [8, 8, 6, 4, 5].
- Budući da u podnizu više nema elemenata, stavka kei_it je sada postavljena na novi položaj, a konačni niz je [2, 8, 6, 4, 5].
- Drugi prolaz započinje sa kei_item = 6 i prolazi kroz podniz koji se nalazi sa njegove leve strane, u ovom slučaju [2, 8].
- Pošto je 6 <8, algoritam pomera element 8 jednu poziciju udesno. Rezultujući niz u ovom trenutku je [2, 8, 8, 4, 5].
- Budući da je 6> 2, algoritam ne treba da nastavi da prolazi kroz podniz, tako da pozicionira kei_item i završava drugi prolaz. Trenutno je rezultujući niz [2, 6, 8, 4, 5].
- Treći prolazak kroz listu postavlja element 4 u njegov tačan položaj, a četvrti prolaz postavlja element 5 na tačno mesto, ostavljajući niz sortiranim.



Vreme implementacije Insertion Sort algoritma

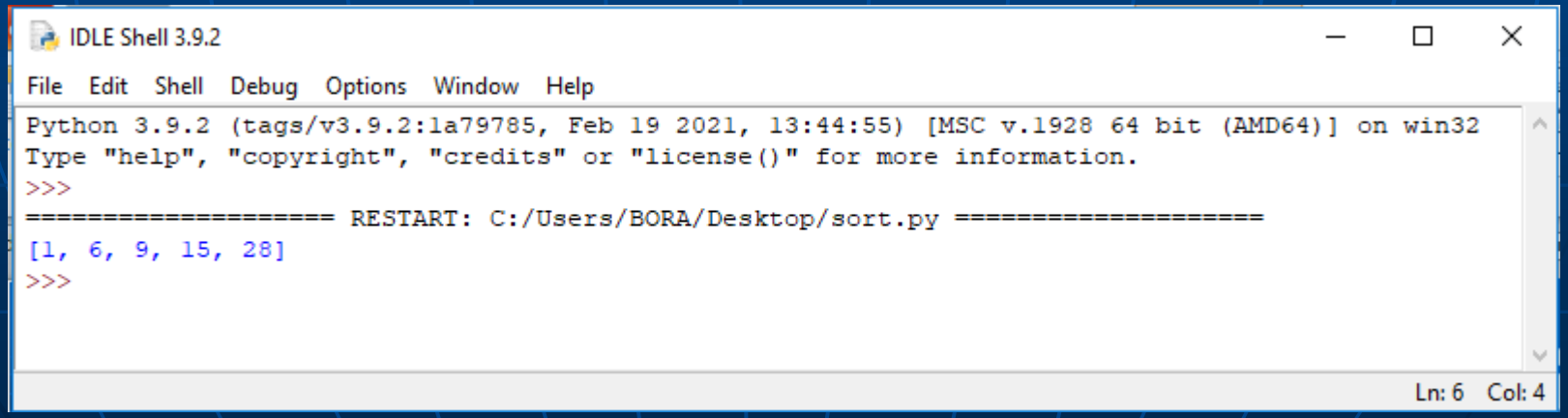
- Da bi dokazali tvrdnju da je sortiranje umetanjem efikasnije od bubble sort, možemo vremenski odrediti algoritam sortiranja umetanja i uporediti ga sa rezultatima sortiranja bubble sort. Da bi to uradili, potrebno je samo da zamenimo poziv run_sorting_algorithm() imenom naše implementacije sortiranja umetanja:

- if __name__ == "__main__":
- # Generate an array of `ARRAY_LENGTH` items consisting
- # of random integer values between 0 and 999
- array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
-
- # Call the function using the name of the sorting algorithm
- # and the array we just created
- run_sorting_algorithm(algorithm="insertion_sort", array=array)

- You can execute the script as before:
- $ python sorting.py
- Algorithm: insertion_sort. Minimum execution time: 56.71029764299999

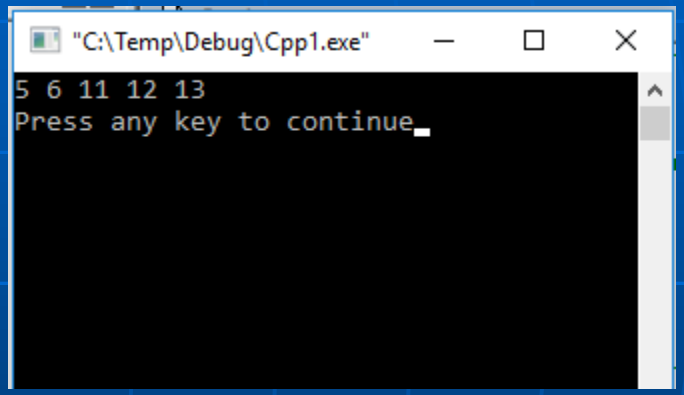
Python Implementacija IS

```
def insertion_sort(nums):
    # Start on the second element
    for i in range(1, len(nums)):
        item_to_insert = nums[i]
        # And keep a reference of the index of the previous element
        j = i - 1
        # Move all items of the sorted segment forward if they are larger than
        # the item to insert
        while j >= 0 and nums[j] > item_to_insert:
            nums[j + 1] = nums[j]
            j -= 1
        # Insert the item
        nums[j + 1] = item_to_insert


# Verify it works
random_list_of_nums = [9, 1, 15, 28, 6]
insertion_sort(random_list_of_nums)
print(random_list_of_nums)
```



C Implementacija IS

```
// C program for insertion sort
#include <math.h>
#include <stdio.h>
 /* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{    int i, key, j;
    for (i = 1; i < n; i++)
{        key = arr[i];
        j = i - 1;
         /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key)
{            arr[j + 1] = arr[j];
           j = j - 1;        }
        arr[j + 1] = key;    }
}
 // A utility function to print an array of size n
void printArray(int arr[], int n)
{    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");}
 /* Driver program to test insertion sort */
int main()
{    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);
     insertionSort(arr, n);
    printArray(arr, n);
     return 0;}
```



Merge sort

- Sortiranje spajanjem (engl. merge sort) je algoritam sortiranja baziran na uspoređivanju složenosti O(n log n). Vecina implementacija daje stabilan niz što znači da implementacija čuva redosld jednakih elemenata sa ulaza. Sortiranje spajanjem je zavadi-pa-vladaj algoritam koji je izmislio John von Neumann 1945-te. (Katajainen and Larsson,1977)

- Detaljan opis i analiza sortiranja spajanjem odozdo-nagore su se pojavili u izvještaju koji su napisali Goldstine i Neumann još davne 1948-e.

- Konceptualno, sortiranje spajanjem radi po sedećem principu:

· Podeliti nesortiran niz na n podnizova, od kojih svaki sadrži 1 element (niz od jednog elementa se smatra sortiranim).

· U više navrata spajati podnizove sve dok se ne dobije novi podniz. Ovo ce biti sortiran niz.

Merge sort

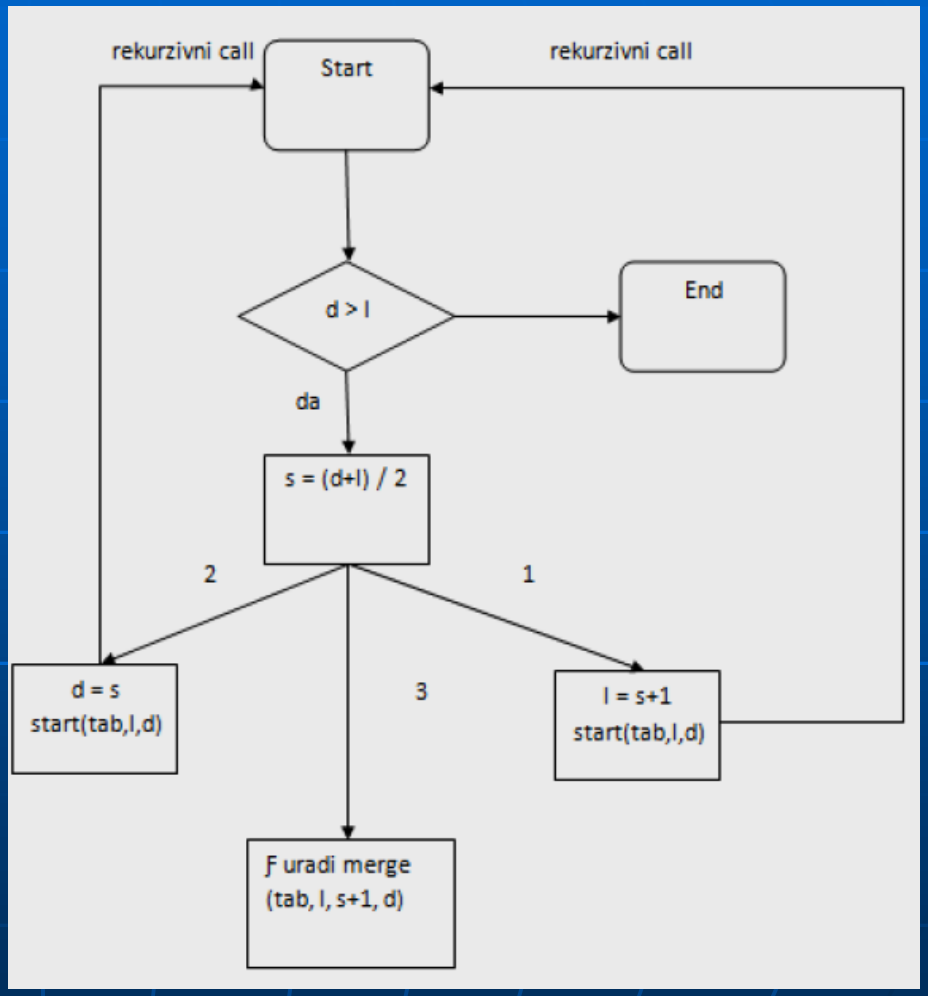
- Primer: Sedeća slika prikazuje korake za sortiranje niza (3, 7, 4, 9, 5, 2, 6, 1).
- U svakom koraku, element u razmatranju je podvučen. Deo koji je pomaknut (ili ostavljen na mestu zato što je najveći dotad razmatrani) u prethodnom koraku biće podebljan.

3 | 7 | **4** | 9 | 5 | **2** | 6 | **1**

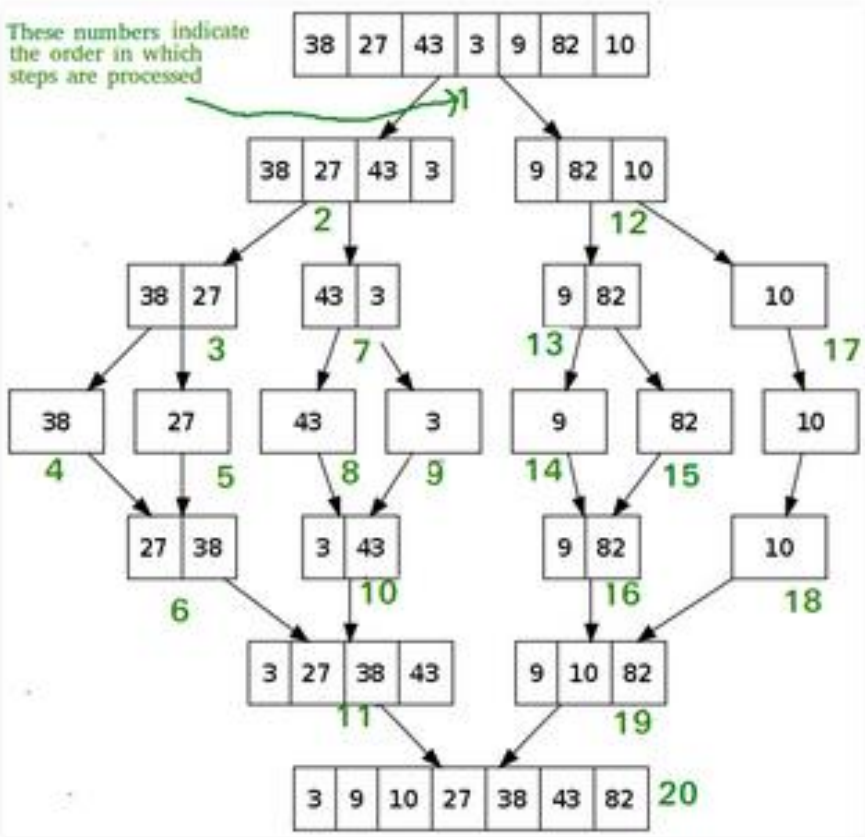
3 7 | **4** 9 | 2 **5** | **1** 6

3 **4** 7 **9** | **1** **2** **5** **6**

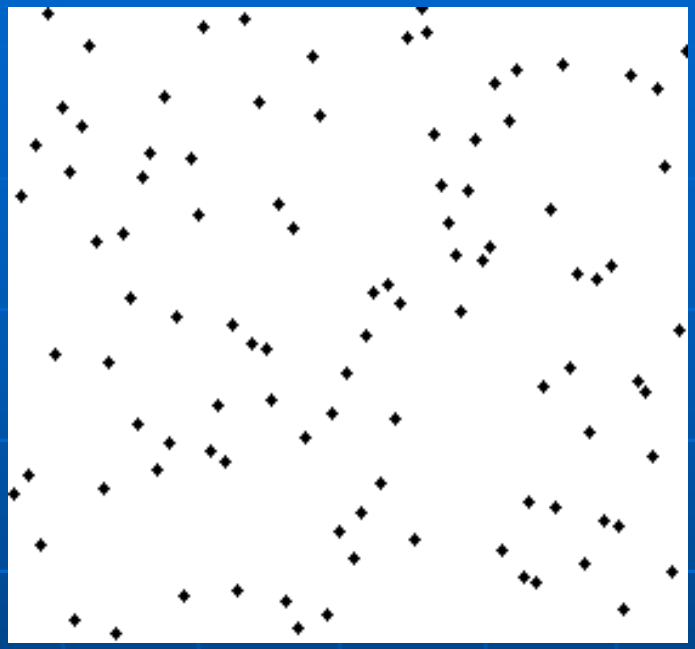
1 2 3 4 5 6 7 9



Merge sort


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
2
9 82 10
12
38 27
3
43 3
7
9 82
13
10
17
38
4
27
5
43
8
3
9
9
14
82
15
10
27 38
6
3 43
10
9 82
16
10
18
3 27 38 43
11
9 10 82
19
3 9 10 27 38 43 82
20

Sortiranje- Redjanje spajanjem (*merge sort*)



- **Ovaj algoritam koristi strategiju "podeli pa vladaj". Niz se rekurzivno deli u segmente koji se zasebno sortiraju, a zatim se sortirani segmenti spajaju u konačno sortirani niz.**
- Dva već sortirana niza se mogu objediniti u treći sortirani niz samo jednim prolaskom kroz nizove (tj. u linearnom vremenu O(m + n) gde su m i n dimenzije polaznih nizova).
- *Merge sort* algoritam deli niz na dve polovine (čija se dužina razlikuje najviše za 1), rekurzivno sortira svaku od njih, i zatim objedinjuje sortirane polovine.

Pogledajmo to na jednom primeru:
(1) Početna lista glasi A{6,3,2,5,1,7,4}.
(2) Delimo je u dve podliste:{6,3,2,5}i{1,7,4}
(3) Sortiramo podliste koristeći identičan algoritam, kako sledi:
  (a) Svaku od njih delimo na podliste (koje na kraju postupka imaju najviše dva elementa) :{6,3}, {2,5} i {1,7},{4}.
  (b) Sortiramo dobijene podliste: {3,6},{2,5} i {1,7}, {4}.
  (c) Spajamo podliste:{2,3,5,6}i{1,4,7}.
(4) Spajamo sortirane podliste:{1,2,3,4,5,6,7}
(5) Kraj postupka.

Složenost Merge sorta

- Za sortiranje n elemenata, složenost u prosečnom i najgorem slucaju je $O(n\log n)$. Ako je vreme izvršavanja za niz dužine $nT(n)$onda relacija $T(n) = 2R(n / 2)+ n$ sledi iz definicije algoritma (primeni algoritam na dva niza čija je dužina polovina originalnog niza i dodaj n koraka za spajanje dva dobijena niza).

Ovo sledi iz Master teorema. U najgorem slučaju, broj upoređivanja je manji ili jednak ($n \lg n - 2^{\lg n} +1$), što je između ($n\lg n - n +1$) i ($n\lg n + n + O(\lg n)$). Za veliko n i slučajno sortiran ulazni niz, očekivani broj upoređivanja je $a \times n$ manji nego u najgorem slučaju, gde je:

$$\alpha = -10 + \sum_{k=0}^{\infty} \frac{1}{2^k + 1} \approx 0.2645$$

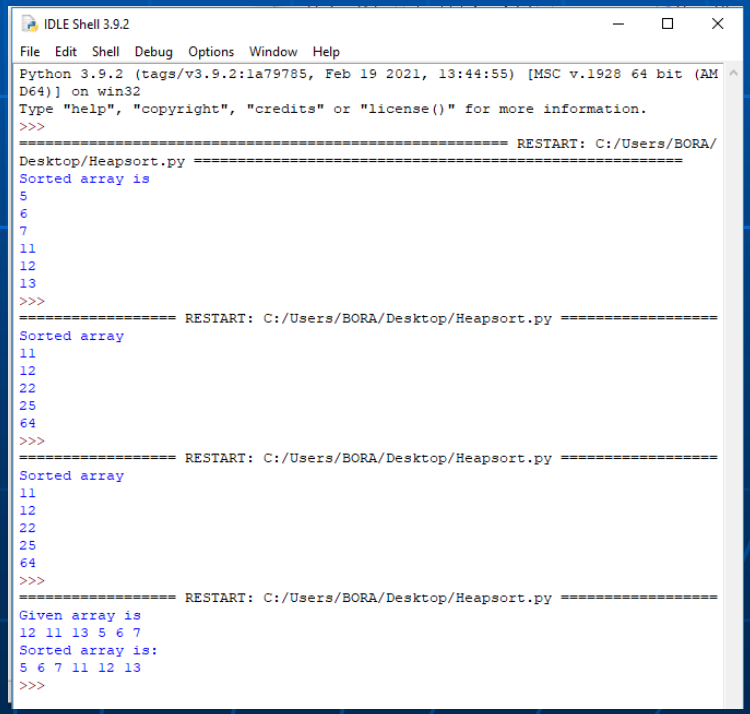
- U najgorem slučaju, kod sortiranja spajanjem imamo 39% manje upoređivanja nego što imamo kod quicksorta u prosečnom slučaju. U broju koraka, složenost u najgorem slučaju iznosi O (n log n) - ista složenost kao u najboljem slučaju quicksort-a, a najbolji slučaj sortiranja spajanjem ima upola manje upoređivanja nego njegov najgori slučaj.

- Najčešća implementacija sortiranja spajanjem koristi pomoćni niz za koji je potrebno alocirati prostor u memoriji.

Python Merge sort

```
# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) > 1:
         # Finding the mid of the array
        mid = len(arr)//2
         # Dividing the array elements
        L = arr[:mid]
         # into 2 halves
        R = arr[mid:]
         # Sorting the first half
        mergeSort(L)
         # Sorting the second half
        mergeSort(R)
         i = j = k = 0
         # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
         # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
         while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

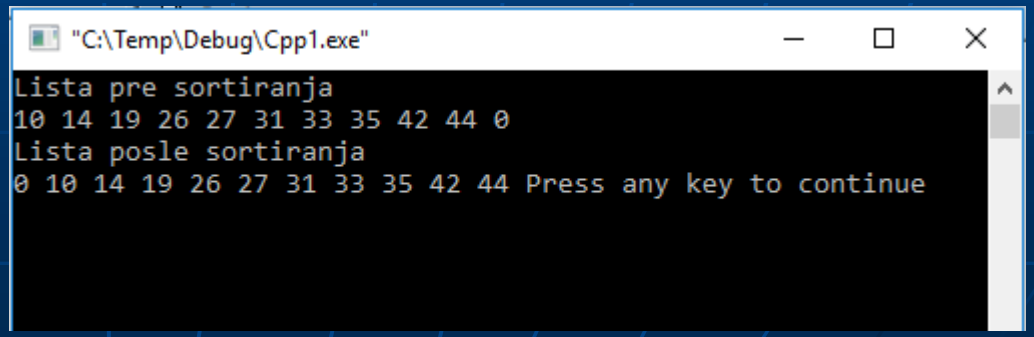
```
#Nastavak skripta
# Code to print the list
def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()
 # Driver Code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)
```



C Merge sort

```
#include <stdio.h>
#define max 10
int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];
void merging(int low, int mid, int high)
{   int l1, l2, i;
  for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++)
{     if(a[l1] <= a[l2])
       b[i] = a[l1++];
     else
       b[i] = a[l2++];  }
     while(l1 <= mid)
     b[i++] = a[l1++];
  while(l2 <= high)
     b[i++] = a[l2++];
  for(i = low; i <= high; i++)
     a[i] = b[i]; }
void sort(int low, int high)
{   int mid;
     if(low < high) {
     mid = (low + high) / 2;
     sort(low, mid);
     sort(mid+1, high);
     merging(low, mid, high);   }
else {       return;   } }
```

```
//Nastavak skripta
int main()
{    int i;
  printf("Lista pre sortiranja");
     for(i = 0; i <= max; i++)
     printf("%d ", a[i]);
  sort(0, max);
  printf("\nLista posle sortiranja\n");
     for(i = 0; i <= max; i++)
     printf("%d ", a[i]);}
```



Quicksort

- Quicksort (engl. Quicksort, qsort; od engl. quick, brz, brzo; i engl. sort, sortiranje) je poznat algoritam sortiranja koji radi na principu podeli pa vladaj, a razvio ga je Toni Hor 1960. godine.

- U proseku, čini O (n log n) uporedjenja kako bi sortirao n stavki. U najgorem slučaju, čini O (N2) poredjenja, iako je takvo ponašanje quicksort-a retkost. Quicksort je u praksi često brži od drugih O (n log n) algoritama. Osim toga, dobro radi s cach memorijom.

- Quicksort je sortiranje upoređivanjem i, u učinkovitim implementacijama, nije stabilna vrsta sortiranja. ("Data structures and algorithm: Quicksort". Auckland University.)

- Sledi opis rada algoritma koji elemente sortira u rastućem poretku. Osnovni princip rada algoritma se deli u tri sledeće celine:

· Izabiranje pivot-elementa na datom intervalu

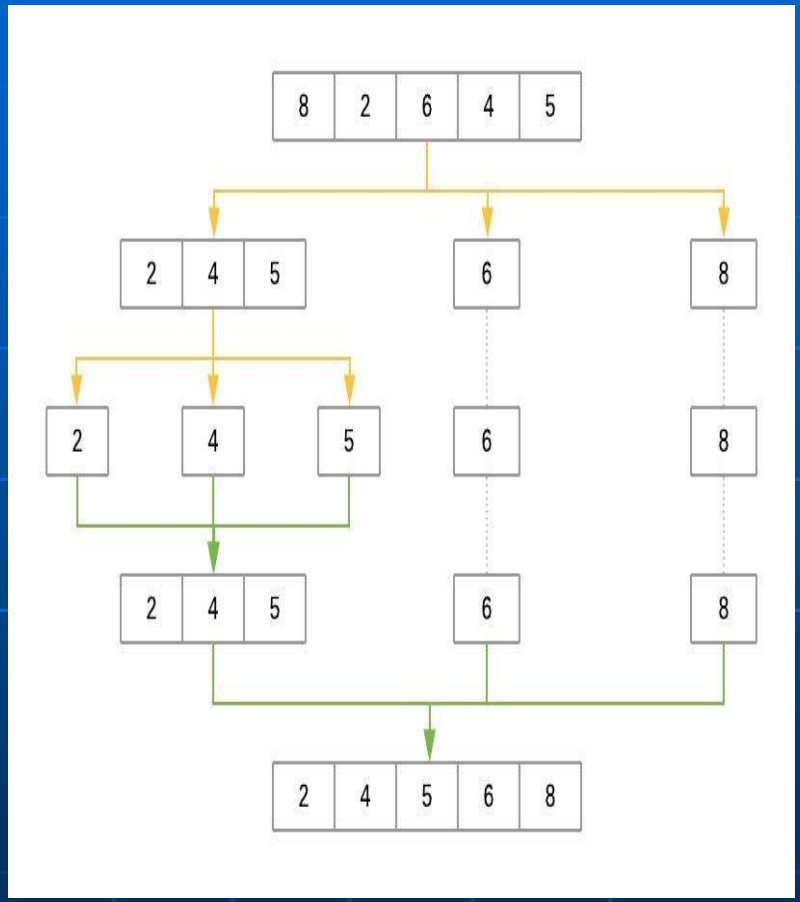
· Raspored svih elemenata manjih ili jednakih ovom pivot-elementu levo od njega, a svih većih desno od njega u nizu

· Rekurzivno ponavljanje ovog postupka na novonastale intervale levo i desno od ovog pivot-elementa

Quicksort

- Baš poput merge sort algoritma, algoritam brzog sortiranja primjenjuje načelo ''podeli-osvoji'' ulazni niz na dve liste, prva sa malim stavkama, a druga sa velikim stavkama. Zatim algoritam sortira obe liste rekurzivno dok se rezultantna lista potpuno ne sortira. Deljenje ulazne liste naziva se particionisanje liste.
- Quicksort prvo odabire pivot element i razdvaja listu oko pivota, stavljajući svaki manji element u manji niz, a svaki veći element u veći niz.
- Stavljanjem svakog elementa sa manjeg niza levo od pivota i svakog elementa sa većeg niza na desnu poziciju pivota element se postavlja tačno tamo gde treba biti na konačnoj razvrstanoj listi.
- To znači da funkcija se sada može rekurzivno primeniti uz isti postupak na nižu i visoku listu, dok se cela lista ne sortira.

Važno upozorenje u vezi sa brzim sortiranjem je da je njegovo najgore izvodjenje spada u O (n^2); iako je to retko. Najsloženije pitanje u brzom sortiranju je stoga izbor dobrog pivot elementa,



Quicksort

- Sljedeća slika prikazuje korake za sortiranje niza (3, 7, 4, 9, 5, 2, 6, 1). U svakom koraku, element u razmatranju je podvučen. Deo koji je pomaknut (ili ostavljen na mestu zato što je najveći dotad razmatrani) u prethodnom koraku biće podebljan.

Izaberemo pivota = 1

| **3** 7 4 9 5 2 6 **1**

Novi pivot = 3

1 | 7 4 9 5 **2** 6 **3**

1 2 | 7 4 9 5 6 **3**

Novi pivot = 6

1 2 3 | 7 **4** 9 5 **6**

1 2 3 4 | 7 9 **5** **6**

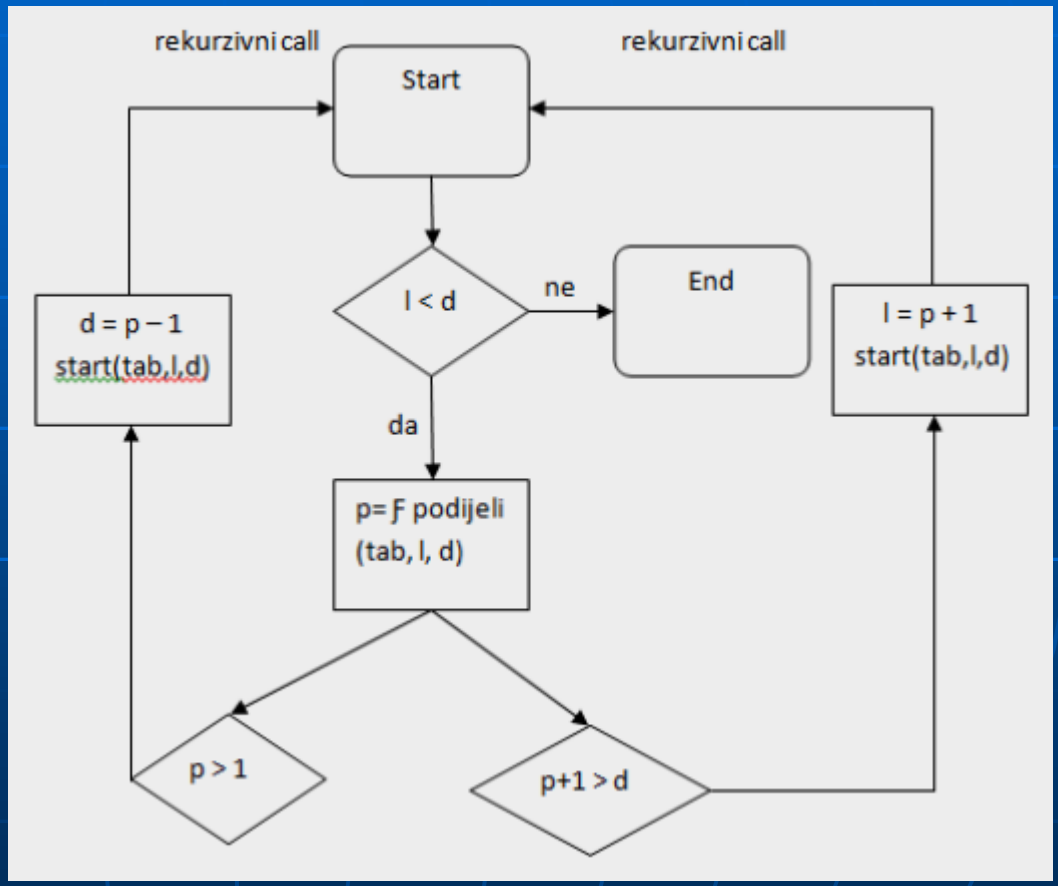
1 2 3 4 5 | 7 9 **6**

Novi pivot = 9

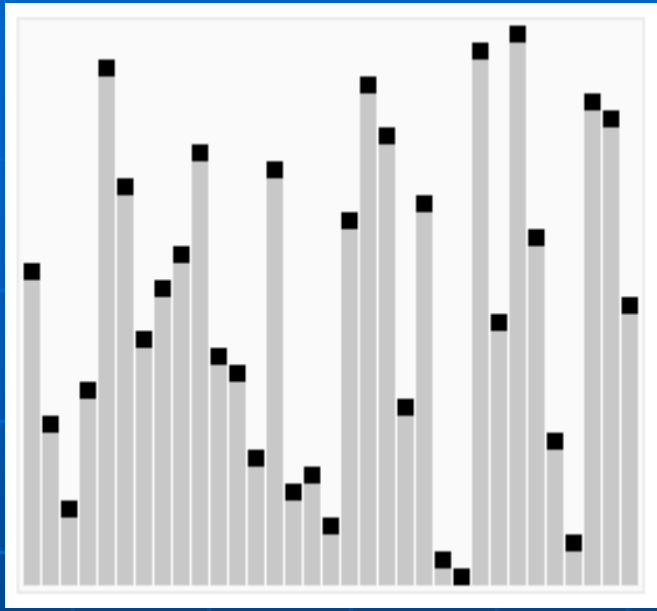
1 2 3 4 5 6 | 7 **9**

1 2 3 4 5 6 7 | **9**

1 2 3 4 5 6 7 9



Sortiranje- Brzo redjanje (quicksort)



“Quicksort” je do sada najbrži poznati algoritam za sortiranje. I ovo je rekurzivni algoritam zasnovan na strategiji “podeli pa vladaj”.

Algoritam se sastoji od sledećih koraka:

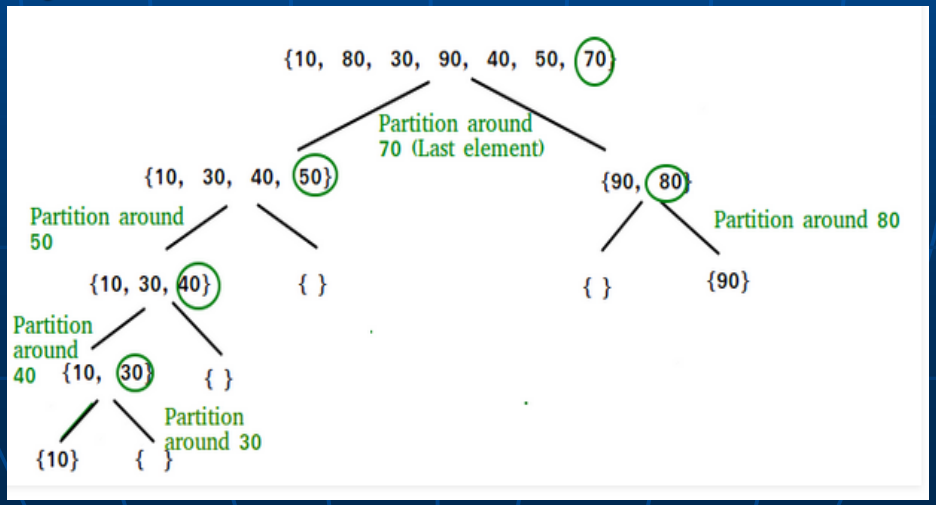
1. Odabir jednog člana niza, tzv. *pivot*-a.
2. Raspodeliti članove niza tako da sve elemente manje od pivota stavimo ispred njega (podniz 1), a veće iza njega (podniz 2). Nakon te raspodele pivot se nalazi na svojoj konačnoj poziciji.
3. Rekurzivno sortirati svaki podniz na isti način.

Dakle, značajno je efikasnije da se problem dimenzije n svodi na dva problema dimenzije n/2 nego na jedan problem dimenzije n-1 — ovo je osnovna ideja takozvanih podeli i vladaj algoritama u koje spada i *quick sort*.

Pogledajmo algoritam na primeru:
(1) Početna lista glasiA {6,3,2,5,1,7,4}.
(2) Odaberemo stožerni element 5.
(3) Uporedjujemo taj element sa ostalima i time dobijamo podliste {3,2,1,4}, {5} i {6,7}.
    (a) Sortiramo drugu podlistu (ostavimo je kakva jeste).
    (b) U prvoj podlisti biramo stožer 2.
    (c) Uporedjujemo taj element i dobijamo nove podliste{1},{2},{3,4}
    (d) Sortiramo podlistu{3,4}
(4) Spajamo sve podatke:{1},{2},{3,4},{5},{6,7}.
(5) Kraj postupka

Složenost Quicksorta

- Algoritam na početku dobije niz i levu i desnu granicu intervala koga treba sortirati. Za ove granice mora važiti da leva ima manji indeks od desne, inače se algoritam prekida. Unutar tog intervala, algoritam izabere tzv. pivot-element, koji se obično uzima sa njegove sredine ili njene okoline. Potom, algoritam zamenjuje mesta pivot-elementa i poslednjeg elementa u nizu i sortiranje može početi.
- Algoritam prolazi kroz celi zadat interval i sve elemente koji su manji ili jednaki tom pivot-elementu slaže na prvo, drugo, treće itd. mesto u nizu.
- Pritom elementi koji su se zatekli na tom prvom, drugom, trećem itd. mestu zamenjuju (eng. swap) svoja mjesta za mesta nađenih elemenata. Elementi koji se već nalaze na nekom od ovih mesta i ispunjavaju uslov da na njemu ostanu se ne premeštaju.
- Po završetku ovog procesa, pivot-element sa kraja intervala se stavlja na kraj ovog niza, iza zadnjeg postavljenog elementa. Taj element je sad na svom mestu i neće biti više potrebe premeštati ga. Zbog ovoga je na početku i bio premešten na kraj da se tokom razmeštanja ne bi moralo pratiti njegovo mesto u nizu te da premeštanje na kraj bude lakše.



Implementacija Quicksort u Python_u

```
from random import randint
def quicksort(array):
    # If the input array contains fewer than two
elements,
    # then return it as the result of the function
    if len(array) < 2:
        return array

    low, same, high = [], [], []

    # Select your `pivot` element randomly
    pivot = array[randint(0, len(array) - 1)]

    for item in array:
 # Elements that are smaller than the `pivot` go to
 # the `low` list. Elements that are larger than
 # `pivot` go to the `high` list. Elements that are
  # equal to `pivot` go to the `same` list.
        if item < pivot:
            low.append(item)
        elif item == pivot:
            same.append(item)
        elif item > pivot:
            high.append(item)

    # The final result combines the sorted `low` list
    # with the `same` list and the sorted `high` list
    return quicksort(low) + same + quicksort(high)
```

U nastavku je opis koda:

- ✓ Linija 6 zaustavlja rekurzivnu funkciju ako niz sadrži manje od dva elementa.
- ✓ Linija 12 nasumično bira pivot element sa liste i nastavlja sa deljenjem lista.
- ✓ Linije 19 i 20 stavljaju svaki element koji je manji od pivota na listu zvanu ''nisko''.
- ✓ Linije 21 i 22 stavljaju svaki element koji je jednak pivotu na listu koja se naziva ''isti''.
- ✓ Linije 23 i 24 stavljaju svaki element koji je veći od pivota na listu koja se naziva ''visokim''.
- ✓ Linija 28 rekurzivno sortira nisku i visoku listu i kombinuje ih zajedno sa sadržajem ''iste'' liste.
- ✓ Evo ilustracije koraka koje quicksort preduzima za sortiranje niza [8, 2, 6, 4, 5]:

Vremena Quicksort implementacije

```
if __name__ == "__main__":
    # Generate an array of `ARRAY_LENGTH` items consisting
    # of random integer values between 0 and 999
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]

    # Call the function using the name of the sorting algorithm
    # and the array you just created
    run_sorting_algorithm(algorithm="quicksort", array=array)
```

Izvršavanjem skripta dobijamo:

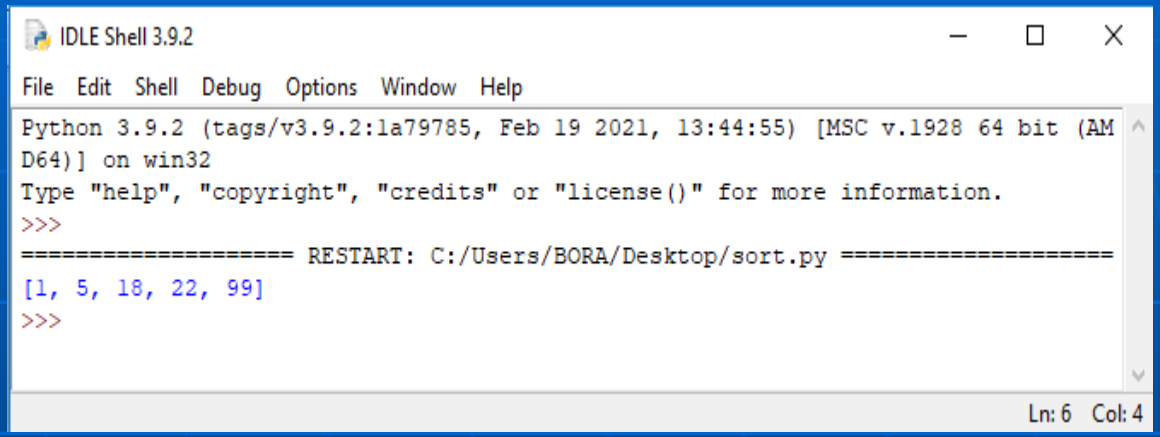
$ python sorting.py

Algorithm: quicksort. Minimum execution time: 0.1167541760002902

- Ne samo da se brzo sortiranje završava za manje od jedne sekunde, već je i puno brže od sortiranja spajanjem (0,11 sekunde naspram 0,61 sekunde).

Python skript QS

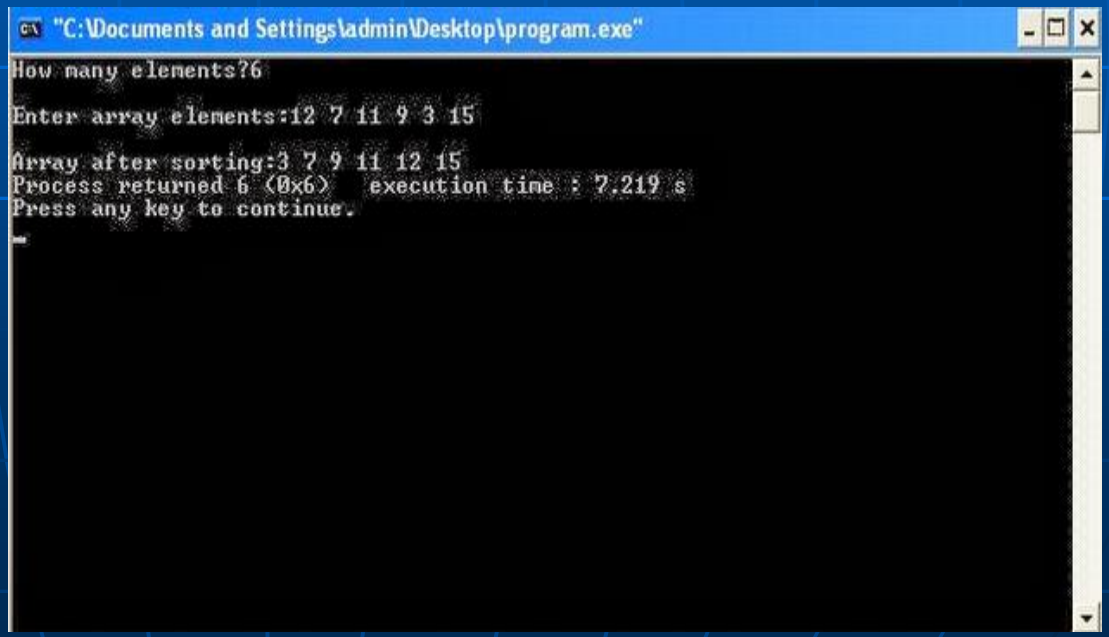
```
# There are different ways to do a Quick Sort partition, this implements the
# Hoare partition scheme. Tony Hoare also created the Quick Sort algorithm.
# There are different ways to do a Quick Sort partition, this implements the
# Hoare partition scheme. Tony Hoare also created the Quick Sort algorithm.
def partition(nums, low, high):
    # We select the middle element to be the pivot. Some implementations select
    # the first element or the last element. Sometimes the median value becomes
    # the pivot, or a random one. There are many more strategies that can be
    # chosen or created.
    pivot = nums[(low + high) // 2]
    i = low - 1
    j = high + 1
    while True:
        i += 1
        while nums[i] < pivot:
            i += 1
        j -= 1
        while nums[j] > pivot:
            j -= 1
        if i >= j:
            return j
        # If an element at i (on the left of the pivot) is larger than the
        # element at j (on right right of the pivot), then swap them
        nums[i], nums[j] = nums[j], nums[i]
def quick_sort(nums):
    # Create a helper function that will be called recursively
    def _quick_sort(items, low, high):
        if low < high:
            # This is the index after the pivot, where our lists are split
            split_index = partition(items, low, high)
            _quick_sort(items, low, split_index)
            _quick_sort(items, split_index + 1, high)
    _quick_sort(nums, 0, len(nums) - 1)
# Verify it works
random_list_of_nums = [22, 5, 1, 18, 99]
quick_sort(random_list_of_nums)
print(random_list_of_nums)
```



C skript QS

```
#include <stdio.h>
void quick_sort(int[],int,int);
int partition(int[],int,int);
 int main()
{int a[50],n,i;
printf(„Koliko elemenata?");
scanf("%d",&n);
printf("\nUnesi elemente polja:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
quick_sort(a,0,n-1);
printf("\nPolje posle sortiranja:");
for(i=0;i<n;i++)
printf("%d ",a[i]);
return 0; }
 void quick_sort(int a[],int l,int u)
{int j;
if(l<u)
{j=partition(a,l,u);
quick_sort(a,l,j-1);
quick_sort(a,j+1,u);}}
 int partition(int a[],int l,int u)
{int v,i,j,temp;
v=a[l];
i=l;
j=u+1;
do
{do
i++;
while(a[i]<v&&i<=u);
do
```

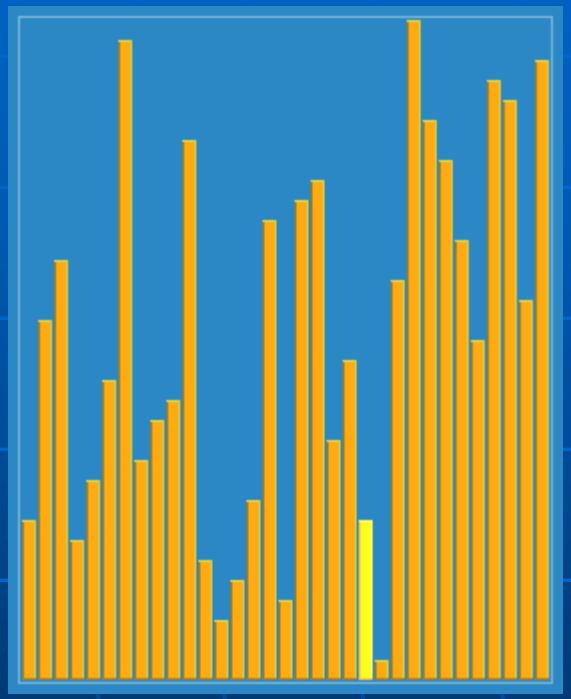
```
//Nastavak skripta
j--;
while(v<a[j]);
if(i<j)
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}while(i<j);
a[l]=a[j];
a[j]=v;
return(j);
}
```



Kompleksnost

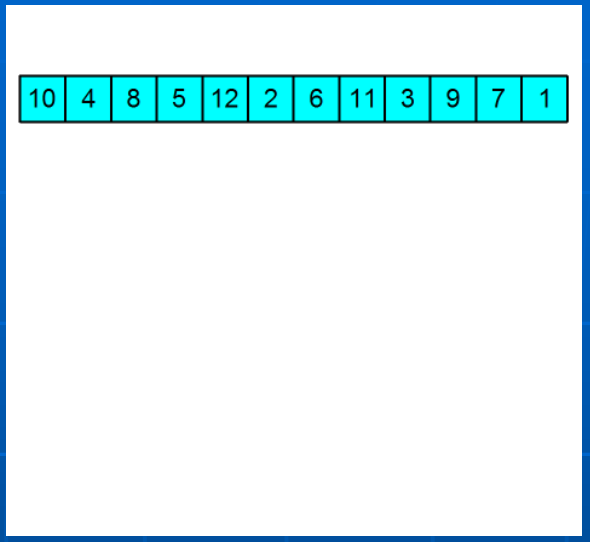
- Najgori je scenarijum kada se najmanji ili najveći element uvek odaberu kao pivot. To bi stvorilo particije veličine n-1, uzrokujući rekurzivne pozive n-1 puta. To nas dovodi do vremenski složene situacije O (n^2) u najgorem slučaju.
- Iako je ovo najgori slučaj, Quick Sort se često koristi jer je prosečna vremenska složenost puno brža.
- Iako funkcija particije koristi ugnježđene while petlje, ona vrši uporedjivanja svih elemenata niza kako bi izvršila svoje zamene.
- Kao takav, ima vremensku složenost O(n).
- Sa dobrim izborom pivota, funkcija brzog sortiranja podelila bi niz na polovine koje rastu logaritamski sa n.
- Stoga je prosečna vremenska složenost algoritma za brzo sortiranje je O (nlog (n)).

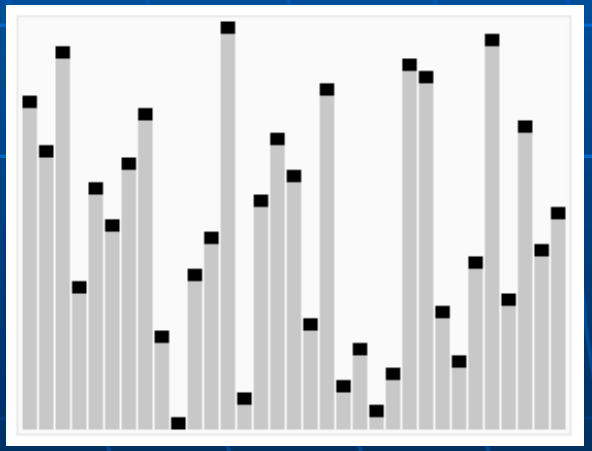
Sortiranje- Šelovo redjanje (*Shell sort*)



- *Shell sort* koristi činjenicu da *insertion sort* funkcioniše odlično kod nizova koji su „skoro sortirani“. Algoritam radi tako što se niz deli na veći broj kratkih kolona, koje se sortiraju primenom *insertion sort* algoritma, čime se omogućava direktna razmena udaljenih elemenata.
- Broj kolona se zatim smanjuje, sve dok se na kraju insertion sort ne primeni na ceo niz. Međutim, do tada su „pripremni koraci“ deljenja na kolone doveli niz u „skoro sortirano“ stanje te se završni korak prilično brzo odvija.
- Ono što se može proizvoljno određivati je broj kolona na koji se vrši deljenje niza u fazama (broj kolona se obično označava sa gap i ovaj niz se u literaturi često označava *gap sequence*).

Sortiranje- Redjanje hrpom (*heapsort*)





- "Heap" sortiranje pripada u familiju "selekcijskih" algoritama sortiranja. Takvi algoritmi sortiranja odredjuju prvo najveći (ili najmanji) element u listi pa ga postavljaju na kraj (ili početak) liste, i na isti način nastavljaju sa ostatkom liste. "Heapsort" izvršava ovaj zadatak korišćenjem strukture podataka koja se zove gomila ili hrpa (eng. *heap*).
- Struktura gomile je binarno stablo u kojem za svaki čvor vredi da je vrednost u čvoru veća ili jednaka od vrednosti svih njegovih sledbenika. Lista se pretvara u gomilu, a korenski čvor je sigurno najveći element liste. Korenski čvor gomile se izuzima i stavlja se na kraj sortirane liste, tj. gomila se skraćuje za 1 element i ponovo podešava.

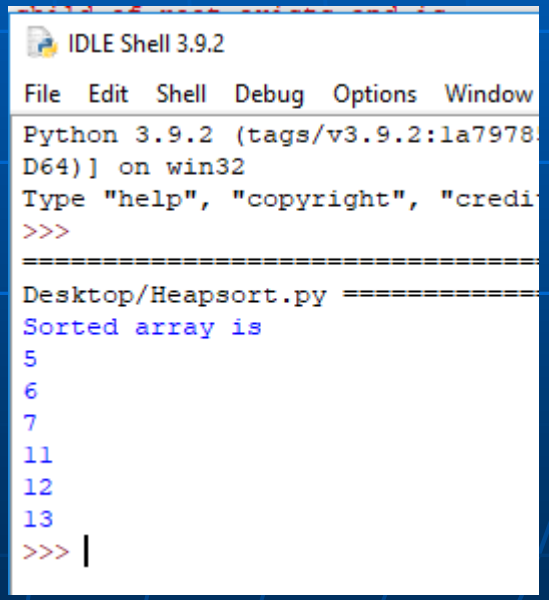
Sortiranje- Redjanje hrpom (*heapsort*)

```
Input data: 4, 10, 3, 5, 1
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)

The numbers in bracket represent the indices in the array
representation of data.

Applying heapify procedure to index 1:
         4(0)
        /   \
    10(1)    3(2)
    /   \
5(3)    1(4)

Applying heapify procedure to index 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)
The heapify procedure calls itself recursively to build heap
 in top down manner.
```

Python (*heapsort*)

```
# Python program for implementation of heap Sort
  # To heapify subtree rooted at index i.
# n is size of heap
def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1     # left = 2*i + 1
    r = 2 * i + 2     # right = 2*i + 2
      # See if left child of root exists and is
    # greater than root
    if l < n and arr[i] < arr[l]:
        largest = l
      # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r
      # Change root, if needed
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]  # swap
          # Heapify the root.
        heapify(arr, n, largest)
  # The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)
      # Build a maxheap.
    # Since last parent will be at ((n//2)-1) we can start at that location.
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
      # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # swap
        heapify(arr, i, 0)
```

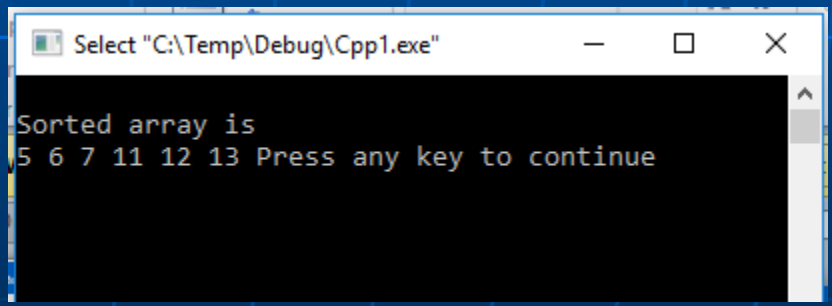
```
# U nastavku skripta sledi deo za testiranje
# algoritma:

# Driver code to test above
arr = [ 12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("Sorted array is")
for i in range(n):
    print ("%d" %arr[i]),
# This code is contributed by Mohit Kumra
```



C (*heapsort*)

```
// C implementation of Heap Sort
#include <stdio.h>
#include <stdlib.h>
// A heap has current size and array of elements
struct MaxHeap
{    int size;    int* array;};
// A utility function to swap to integers
void swap(int* a, int* b) { int t = *a; *a = *b;  *b = t; }
// The main function to heapify a Max Heap. The function
// assumes that everything under given root (element at
// index idx) is already heapified
void maxHeapify(struct MaxHeap* maxHeap, int idx)
{  int largest = idx;  // Initialize largest as root
    int left = (idx << 1) + 1;  // left = 2*idx + 1
    int right = (idx + 1) << 1; // right = 2*idx + 2
    // See if left child of root exists and is greater than  root
    if (left < maxHeap->size &&
        maxHeap->array[left] > maxHeap->array[largest])
        largest = left;
    // See if right child of root exists and is greater than
    // the largest so far
    if (right < maxHeap->size &&
        maxHeap->array[right] > maxHeap->array[largest])
        largest = right;
    // Change root, if needed
    if (largest != idx)
    {swap(&maxHeap->array[largest], &maxHeap->array[idx]);
        maxHeapify(maxHeap, largest);    }}
// A utility function to create a max heap of given capacity
struct MaxHeap* createAndBuildHeap(int *array, int size)
{  int i;
struct MaxHeap* maxHeap =(struct MaxHeap*) malloc(sizeof(struct MaxHeap));
    maxHeap->size = size;   // initialize size of heap
    maxHeap->array = array; // Assign address of first element of array
    // Start from bottommost and rightmost internal mode and heapify all
    // internal modes in bottom up way
    for (i = (maxHeap->size - 2) / 2; i >= 0; --i)
        maxHeapify(maxHeap, i);
    return maxHeap; }
```

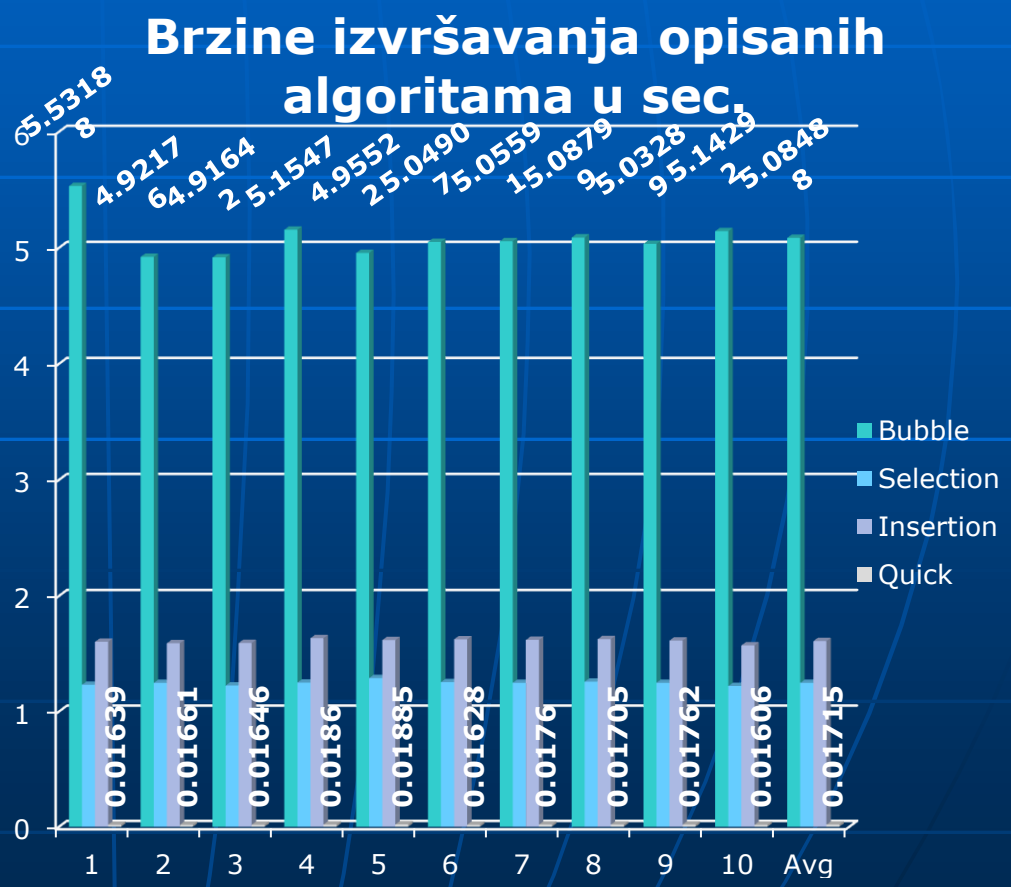
```
// The main function to sort an array of given size
void heapSort(int* array, int size)
{    // Build a heap from the input data.
struct MaxHeap* maxHeap = createAndBuildHeap(array, size);
    // Repeat following steps while heap size is greater than 1.
 // The last element in max heap will be the minimum element
while (maxHeap->size > 1)    {
// The largest item in Heap is stored at the root. Replace
// it with the last item of the heap followed by reducing the
// size of heap by 1.
swap(&maxHeap->array[0], &maxHeap->array[maxHeap->size - 1]);
maxHeap->size;  // Reduce heap size
// Finally, heapify the root of tree.
maxHeapify(maxHeap, 0);    }}
// A utility function to print a given array of given size
void printArray(int* arr, int size)
{    int i;
for (i = 0; i < size; ++i)
printf("%d ", arr[i]);}
/* Driver program to test above functions */
int main(){    int arr[] = {12, 11, 13, 5, 6, 7};
 int size = sizeof(arr)/sizeof(arr[0]);
heapSort(arr, size);
printf("\nSorted array is \n");
printArray(arr, size);    return 0;}
```



UPOREDJENJE

- Da bismo dobili saznanje o tome koliko brzo se opisani algoritmi izvode, generišemo listu od 5000 brojeva između 0 i 1000. Zatim određujemo koliko vremena treba da se svaki algoritam izvrši. To se ponavlja 10 puta kako bismo pouzdanije mogli uspostaviti obrazac vremena izvršavanja. Ovo su rezultati, vreme je u sekundama:

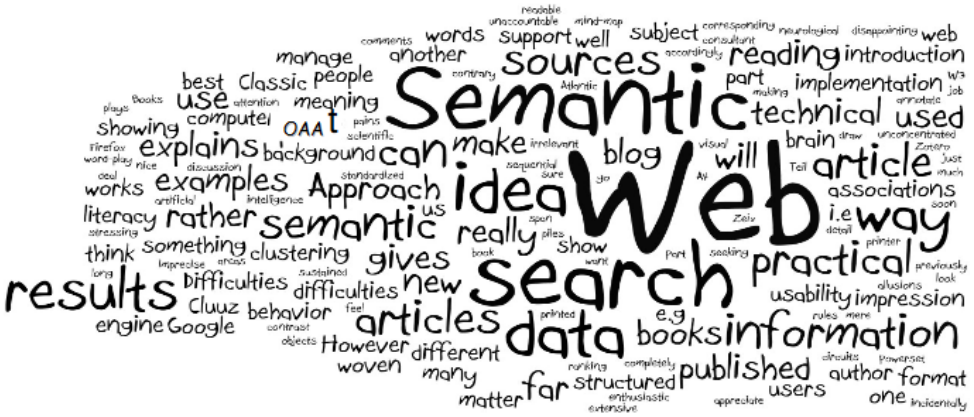
Run	Bubble	Selection	Insertion	Quick
1	5.53188	1.23152	1.60355	0.01639
2	4.92176	1.24728	1.59103	0.01661
3	4.91642	1.22440	1.59362	0.01646
4	5.15470	1.25053	1.63463	0.01860
5	4.95522	1.28987	1.61759	0.01885
6	5.04907	1.25466	1.62515	0.01628
7	5.05591	1.24911	1.61981	0.01760
8	5.08799	1.25808	1.62603	0.01705
9	5.03289	1.24915	1.61446	0.01762
10	5.14292	1.22021	1.57273	0.01606
Avg	5.08488	1.24748	1.60986	0.01715



Boyer-Moore-Horspool algoritam pretraživanja teksta

Uvod

- *podatci nestrukturirani*, u *tekstualnom obliku*
- velika količina digitalnih tekstualnih podataka
- sustavi za analizu i pretraživanje postaju dio informacijsko-komunikacijske infrastrukture



Pronađite OAA!!

Boyer-Moore-Horspool algoritam pretraživanja teksta

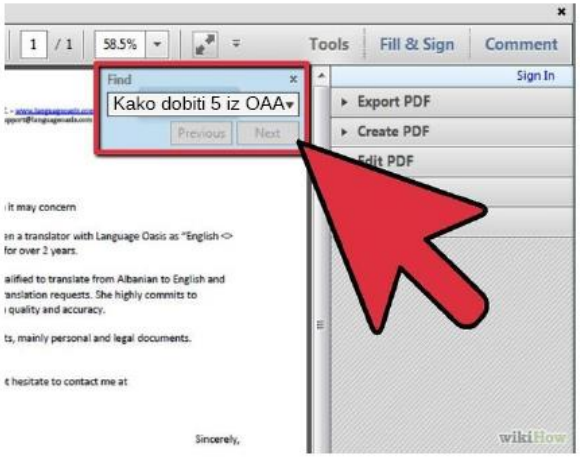
Pretpostavke i oznake

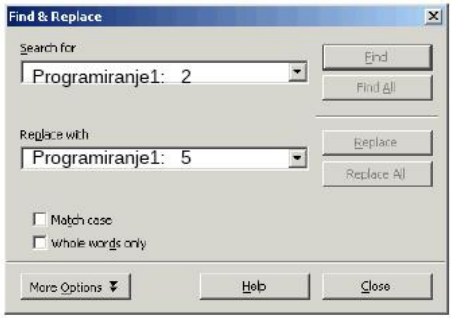
- Pretražujemo tekst na engleskom jeziku, sa engleskim znakovima.
- Oznake:
  - veliko *O je* notacija za vremensku složenost.
  - *P* za traženi uzorak, *T* za izvorni tekst.
  - m = |*P*| i n = |*T*| (broj znakova u *P* odnosno *T*).
  - δ = |znakovi koji se mogu pojaviti u tekstu/uzorku|.

Boyer-Moore-Horspool algoritam pretraživanja teksta

Boyer-Moore algoritam (BM)

- efikasan u stvarnim primjenama
- Bob Boyer i J Strothert Moore 1977
- Boyer-Moore-Horspool (BMH), Boyer-Moore-Horspool-Sunday (BMHS) ...
- Upotreba





Boyer-Moore-Horspool algoritam pretraživanja teksta

Ideja algoritma i implementacija

- Ideja je uspoređivati znakove s desna na lijevo.

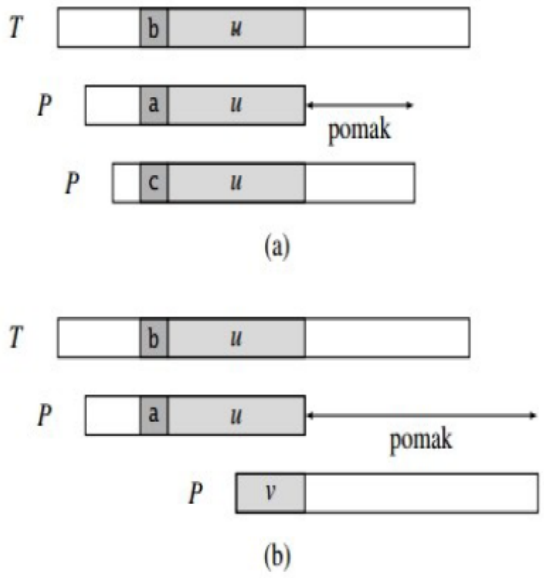

```
n ← length [T]
m ← length [P]
for s ← 0 to n - m do
    if P[1 . . m] = T[s +1 . . s + m]
        then return valid shift s
```

- *Implementacija je ostvarena pomoću jednostavnog pretraživanja teksta uz unaprijed izračunate vrijednosti za ažuriranje pomaka: pomak dobrog sufiksa i pomak lošeg znaka.*

- *Za svaki pomak biramo veći rezultat te dvije funckije.*

Boyer-Moore-Horspool algoritam pretraživanja teksta

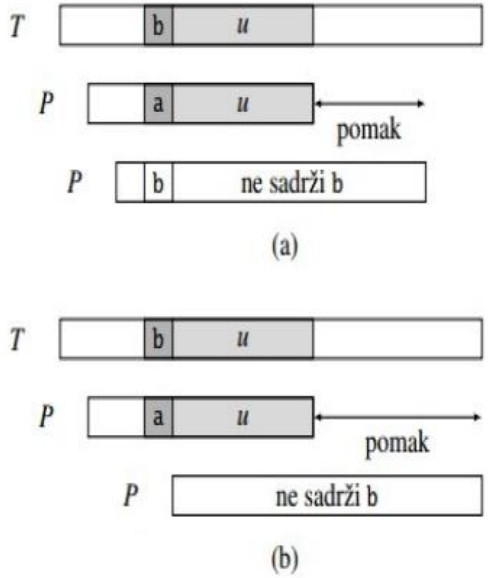
Pomak dobrog sufiksa



Slika: pomak dobrog sufiksa. **(a)** Podniz *u* se ponovo pojavljuje u uzorku s time da mu prethodni znak *c* različit od *a*. **(b)** Podniz *u* se pojavljuje samo u sufiksu uzorka.

Boyer-Moore-Horspool algoritam pretraživanja teksta

Pomak lošeg znaka



Slika: pomak lošeg znaka. **(a)** Znak b pojavljuje se u uzorku P. **(b)** Znak b se ne pojavljuje u uzorku P.

Boyer-Moore-Horspool algoritam pretraživanja teksta

Analiza

- Prednosti:
  - heuristike *dobrog sufiksa* i *lošeg znaka* daju dobru vrijednost pomaka.

- Mane:
  - heuristika dobrog sufiksa je teška za shvatiti i implementirati.
  - heuristika lošeg znaka će dati malen pomak ako dođe do nepodudaranja nakon mnogo istih znakova.

- Vremenska složenost u fazi pripremanja tablice pomaka je $O(m + \delta)$.

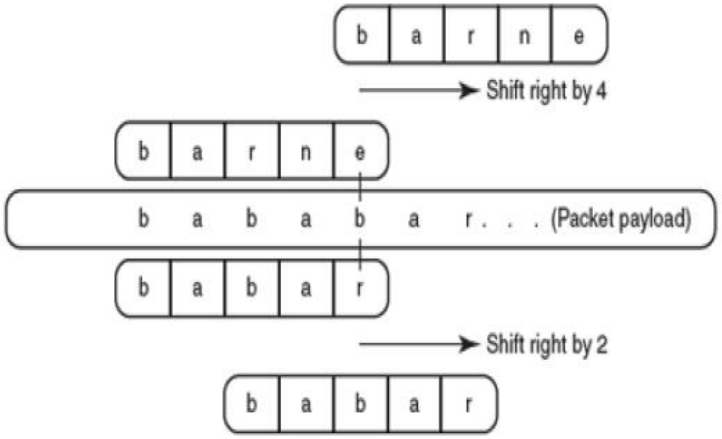
- Vremenska složenost najboljeg slučaja je $O(n/m)$.

- Vremenska složenost je $O(nm)$.

Boyer-Moore-Horspool algoritam pretraživanja teksta

Boyer-Moore-Horspool algoritam (BMH)

- Horspool 1980. uklonio heuristiku dobrog sufiksa
- neovisan o mjestu nepodudaranja



Boyer-Moore-Horspool algoritam pretraživanja teksta

Pseudokod

- U nastavku je dana funkcija za računanje tablice pomaka te algoritam BMH.

```
function preprocess(pattern)
    T ← new table of 256 integers
    for i from 0 to 256 exclusive
        T[i] ← length(pattern)
    for i from 0 to length(pattern) - 1 exclusive
        T[pattern[i]] ← length(pattern) - 1 - i
    return T
```

```
function search(needle, haystack)
    T ← preprocess(needle)
    skip ← 0
    while length(haystack) - skip ≥ length(needle)
        i ← length(needle) - 1
        while haystack[skip + i] = needle[i]
            if i = 0
                return skip
            i ← i - 1
        skip ← skip + T[haystack[skip + length(needle) - 1]]
    return not-found
```

Boyer-Moore-Horspool algoritam pretraživanja teksta

Analiza

- Prednosti:
  - lako za implementirati.
  - pomicanje se uvijek odrađuje na temelju zadnje znamenke čime se postiže veća efikasnost.
  - koristi manje memorije od BM.

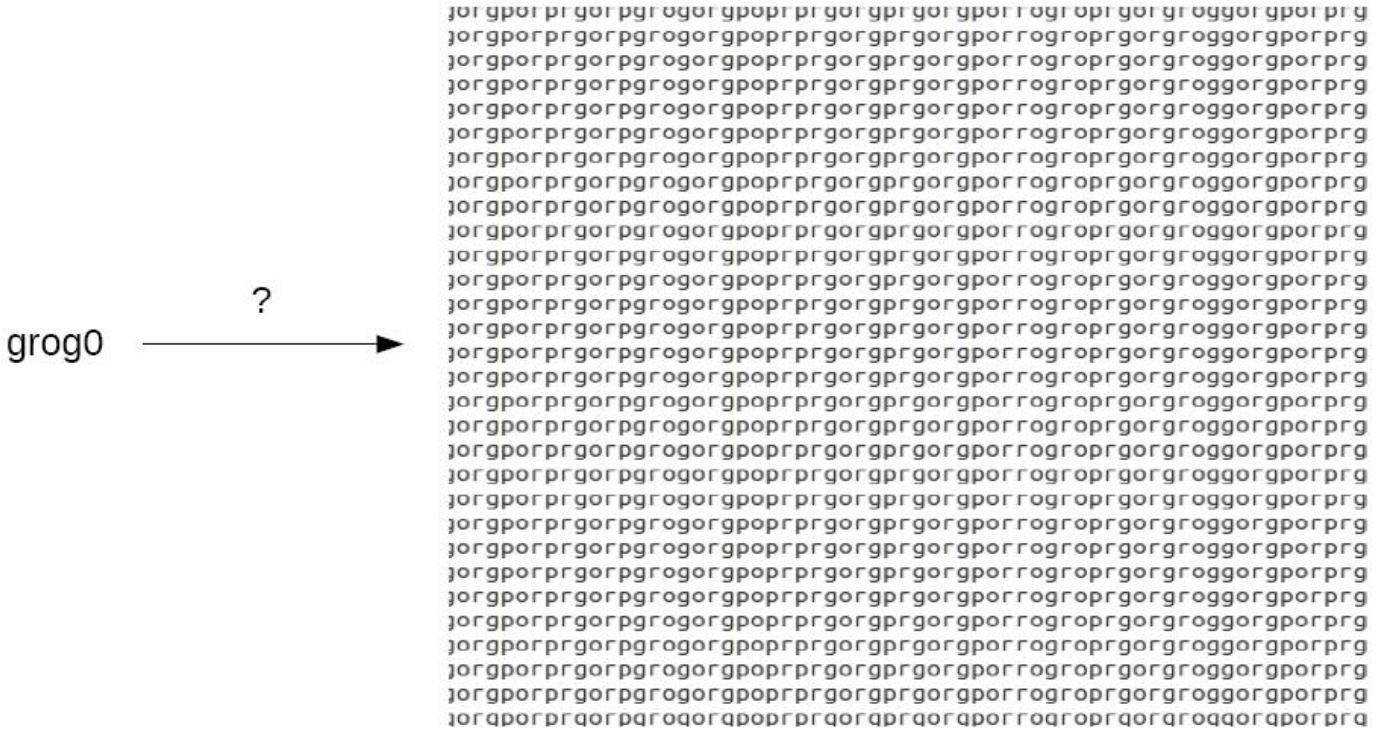
- Mane:
  - nekad heuristika dobrog sufiksa daje veci pomak od ostalih heuristika.

Boyer-Moore-Horspool algoritam pretraživanja teksta

Primjer po koracima...

Boyer-Moore-Horspool algoritam pretraživanja teksta

Primjer 1 – najbolji slučaj



Boyer-Moore-Horspool algoritam pretraživanja teksta

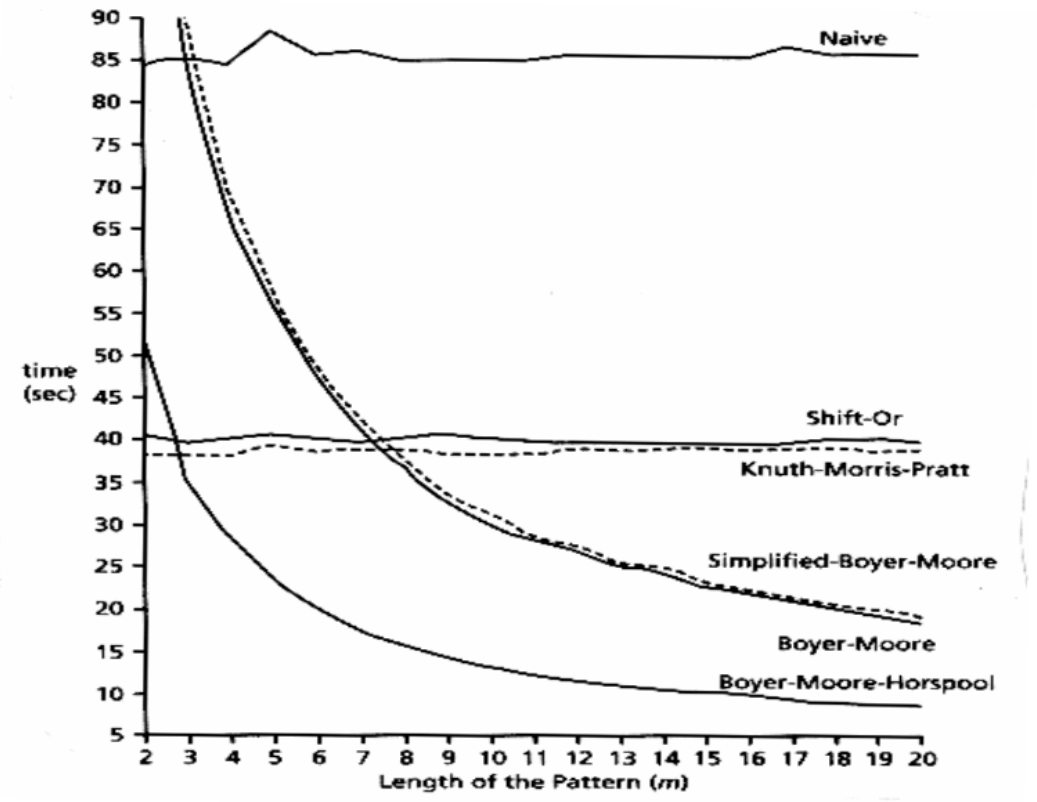
Analiza

KEYWORD	FOUND AT	TIME	TEST LENGTH
grog0	150	0.0000290870666504	155
grog1	297	0.0000400543212891	302
grog2	591	0.0000770092010498	596
grog3	1179	0.000112056732178	1184
grog4	2355	0.000232934951782	2360
grog5	4707	0.000488042831421	4712
grog6	9411	0.000770092010498	9416
grog7	18819	0.00166797637939	18824
grog8	37635	0.00309801101685	37640
grog9	75267	0.0063488483429	75272
grog10	150531	0.0107190608978	150537
grog11	301059	0.0194098949432	301065
grog12	602115	0.0384359359741	602121
grog13	1204227	0.0765979290009	1204233
grog14	2408451	0.155214071274	2408457
grog15	4816899	0.31344294548	4816905
grog16	9633795	0.615411043167	9633801
grog17	19267587	1.23498392105	19267593

Kako vreme raste sa obzirom na veličinu uzorka P ?

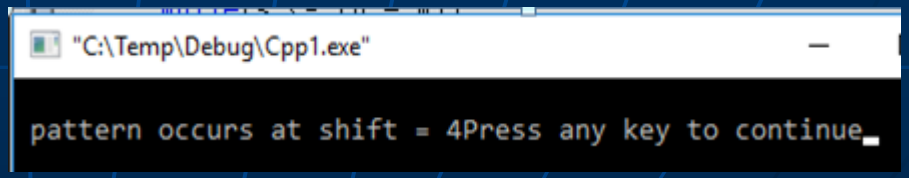
Boyer-Moore-Horspool algoritam pretraživanja teksta

Usporedba algoritama



C Program for Bad Character Heuristic of Boyer Moore String Matching Algorithm

```
/* C Program for Bad Character Heuristic of Boyer
   Moore String Matching Algorithm */
# include <limits.h>
# include <string.h>
# include <stdio.h>
  # define NO_OF_CHARS 256
  // A utility function to get maximum of two integers
int max (int a, int b) { return (a > b)? a: b; }
  // The preprocessing function for Boyer Moore's
// bad character heuristic
void badCharHeuristic( char *str, int size,
     int badchar[NO_OF_CHARS])
{    int i;
     // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;
     // Fill the actual value of last occurrence
    // of a character
    for (i = 0; i < size; i++)
        badchar[(int) str[i]] = i;}
  /* A pattern searching function that uses Bad
   Character Heuristic of Boyer Moore Algorithm */
void search( char *txt,  char *pat)
{  int m = strlen(pat);
    int n = strlen(txt);
    int badchar[NO_OF_CHARS];
     /* Fill the bad character array by calling
      the preprocessing function badCharHeuristic()
      for given pattern */
    badCharHeuristic(pat, m, badchar);
     int s = 0;  // s is shift of the pattern with
// respect to text
    while(s <= (n - m))
    { int j = m-1;
        /* Keep reducing index j of pattern while characters of pattern
and text are matching at this shift s */
       while(j >= 0 && pat[j] == txt[s+j])
          j--;
        /* If the pattern is present at current shift, then index j will
become -1 after the above loop */
       if (j < 0)
       { printf("\n pattern occurs at shift = %d", s);

          /* Shift the pattern so that the next character in text aligns
with the last occurrence of it in pattern. The condition s+m < n is
necessary for the case when pattern occurs at the end of text */
          s += (s+m < n)? m-badchar[txt[s+m]] : 1;  }
         else
          /* Shift the pattern so that the bad character in text aligns with
the last occurrence of it in pattern. The max function is used to  make
sure that we get a positive shift.  We may get a negative shift if the
last  occurrence  of bad character in pattern  is on the right side of the
current  character. */
          s += max(1, j - badchar[txt[s+j]]);  }
}

/* Driver program to test above function */
int main()
{  char txt[] = "ABAAABCD";
    char pat[] = "ABC";
    search(txt, pat);
    return 0;}
```



Python Program for Bad Character Heuristic of Boyer Moore String Matching Algorithm

```
# Python3 Program for Bad Character Heuristic
# of Boyer Moore String Matching Algorithm
 NO_OF_CHARS = 256
 def badCharHeuristic(string, size):
  '''
  The preprocessing function for
  Boyer Moore's bad character heuristic
  '''
    # Initialize all occurrence as -1
  badChar = [-1]*NO_OF_CHARS
    # Fill the actual value of last occurrence
  for i in range(size):
      badChar[ord(string[i])] = i;
    # retun initialized list
  return badChar
 def search(txt, pat):
  '''
  A pattern searching function that uses Bad Character
  Heuristic of Boyer Moore Algorithm
  '''
  m = len(pat)
  n = len(txt)
    # create the bad character list by calling
  # the preprocessing function badCharHeuristic()
  # for given pattern
  badChar = badCharHeuristic(pat, m)
    # s is shift of the pattern with respect to text
  s = 0
  while(s <= n-m):
     j = m-1
```

```
# Keep reducing index j of pattern while
      # characters of pattern and text are matching
      # at this shift s
      while j>=0 and pat[j] == txt[s+j]:
         j -= 1

      # If the pattern is present at current shift,
      # then index j will become -1 after the above loop
      if j<0:
         print("Pattern occur at shift = {}".format(s))

         '''
            Shift the pattern so that the next character in text
aligns with the last occurrence of it in pattern. The condition s+m < n is
necessary for the case when pattern occurs at the end of text
            '''
         s += (m-badChar[ord(txt[s+m])] if s+m<n else 1)
      else:
         '''
            Shift the pattern so that the bad character in text aligns with
the last occurrence of it in pattern. The max function is used to make
sure that we get a positive shift. We may get a negative shift if the last
occurrence of bad character in pattern is on the right side of the
current character.
            '''
         s += max(1, j-badChar[ord(txt[s+j])])

  # Driver program to test above function
def main():
   txt = "ABAAABCD"
   pat = "ABC"
   search(txt, pat)

if __name__ == '__main__':
   main()
```

```
========== RESTART:
Pattern occur at shift = 4
>>>
```

Primer za Bad Character Heuristic of Boyer Moore String Matching Algorithm

```
Input:  txt[] = "THIS IS A TEST TEXT"
        pat[] = "TEST"
Output: Pattern found at index 10

Input:  txt[] =  "AABAACAADAABAABA"
        pat[] =  "AABA"
Output: Pattern found at index 0
        Pattern found at index 9
        Pattern found at index 12
```

Ideja heuristike lošeg karaktera je jednostavna. Karakter teksta koji se ne podudara sa trenutnim karakterom uzorka naziva se Loš karakter. Posle pronadjene neusklađenosti, menjamo patern dok -

1) Neusklađenost postaje podudarnost
2) Uzorak P kreće se pored neusklađenog znaka.

Slučaj 1 - Neusklađenost se poklapa

Tražićemo položaj posljednje pojave nepodudaranja znakova u uzorku, a ako znak neusklađenosti postoji u uzorku, pomaknut ćemo obrazac tako da se poravna sa nepodudarajućim znakom u tekstu T.

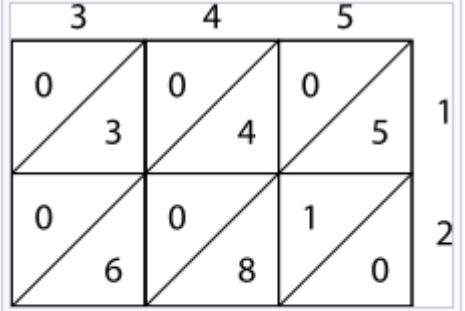
Slučaj 2 - Uzorak se pomiče iza znaka neusklađenosti Tražitćemo položaj poslednje pojave nepodudaranja znakova u uzorku, a ako znak ne postoji, pomaknut ćemo uzorak iza znaka neusklađenosti.

Množenje rešetkom

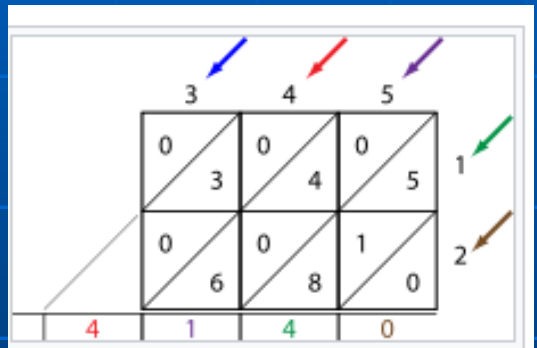
- Množenje rešetkom ili sitom je algoritamski ekvivalent dugog množenja. Zahteva pipremu rešetke, tj mreže nacrtane na papiru koja vodi računanje i razdvaja množenja od sabiranja. Prvi put je korišćen u Evropi 1202. godine u Fibonačijevoj "Liber Abaci". Matrakci Nasuh je predstavio 6 različitih varijanti ove metode u knjizi iz 16.veka "Umdet-ul Hisab".
- Kao što je prikazano u primeru, množenik i množilac su napisani iznad i desno od rešetke ili sita. Tokom faze množenja, rešetka se popunjava dvocifrenim proizvodima odgovarajućih cifara koje označavaju svaki red i kolonu. Desetice idu u gornji levi ugao.
- Tokom faze sabiranja, rešetka se sabira na dijagonalama. Na kraju, ako je neophodna faza pomeranja, rešenje prikazano na levim i donjim stranama rešetke se konvertuje u normalnu formu prenosom cifara desetica kao u algoritmu dugog množenja.

Množenje rešetkom

- Slika pokazuje kako izračunati 345 × 12 koristeći množenje rešetkom. (4140)



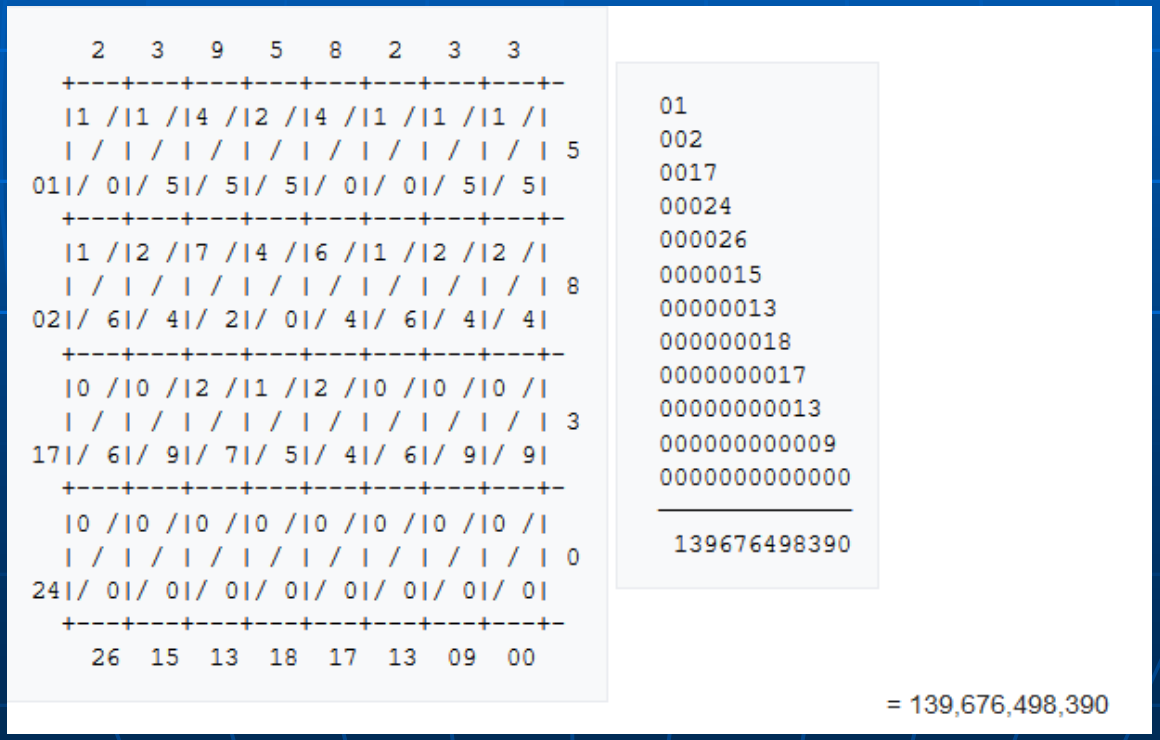
Prvo, postaviti mrežu tako da se prvo obeleže redovi i kolone sa brojevima koji će biti pomnoženi. Onda popuniti kutije sa deseticama u gornjim trouglovima i jedinicama u donjim.



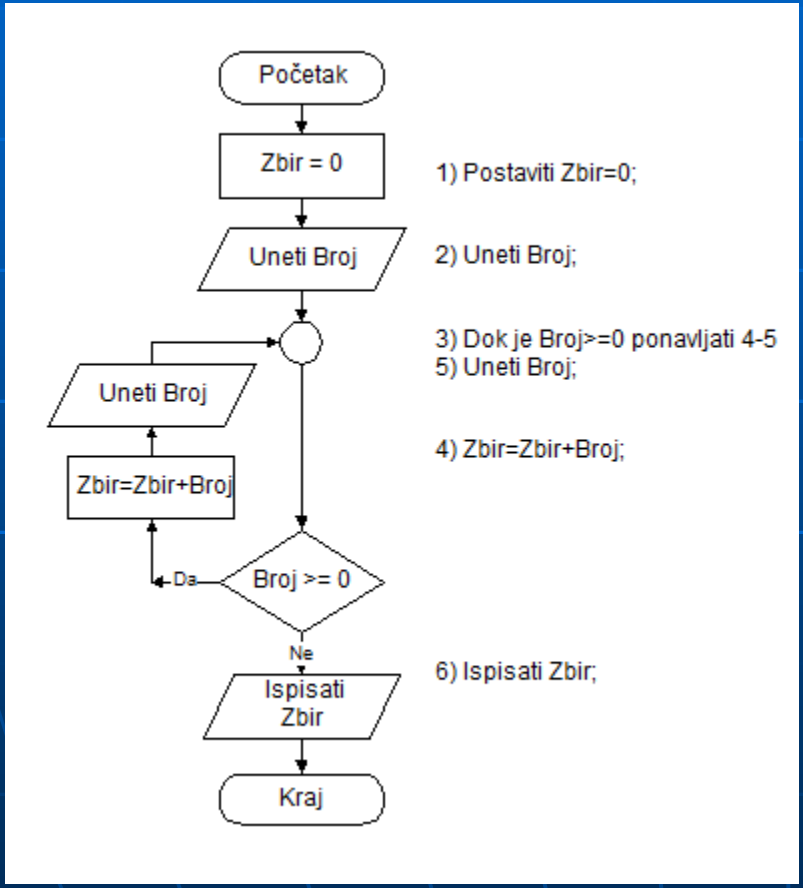
Na kraju, sabrati brojeve duž dijagonala i preneti koliko treba da bi se dobio rezultat.

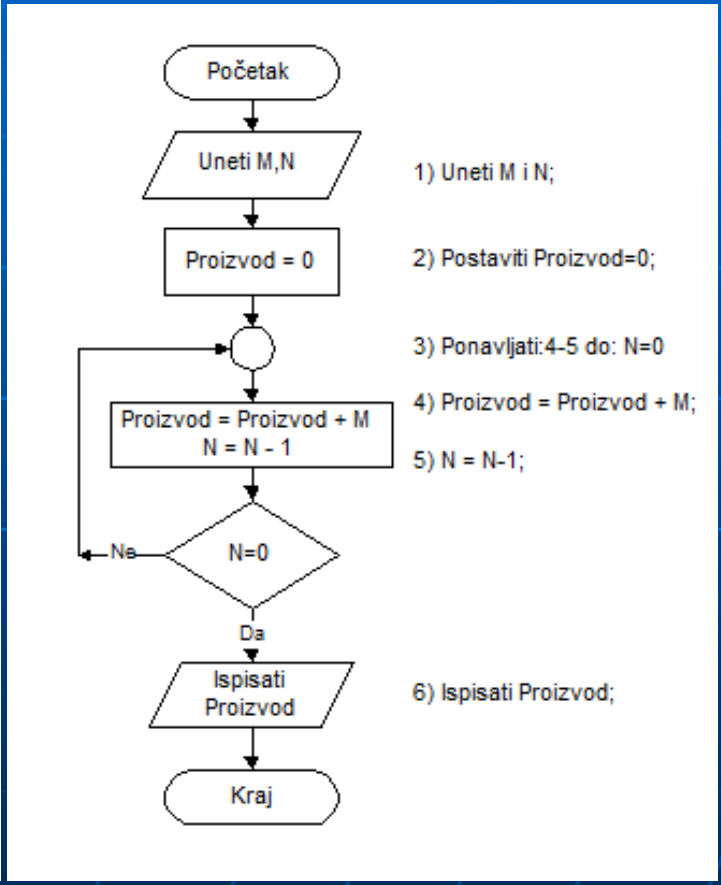
Množenje rešetkom

- Sliku ispod prikazuje množenje 23 958 233 sa 5830, rezultat je 139 676 498 390. Obratiti pažnju da je 23 958 233 na vrhu rešetke a 5830 je sa desne strane. Proizvodi popunjavaju rešetku ili sume tih proizvoda (na dijagonali) su duž levih i donjih strana. Zatim su te sume izračunate, kao što je prikazano.



Primer: Zbir i proizvod dva broja





Stabilnost algoritama

- Kada se sortira neka vrsta podataka, samo deo podataka se ispituje pri određivanju redosleda. Na primeru sa sortiranjem karata, u kartici sa desne strane se vidi da su karte sortirane prema njihovom rangu, a njihova boja se ignoriše. Ovo daje mogućnost višestrukih različitih verzija sortiranja originalne liste.

- *Algoritmi stabilnog sortiranja* se prema sledećem pravilu: ako dve stavke uporedi kao jednake (kao što su dve karte petice) onda će njihov relativni redosled biti sačuvan, tako da ako jedna dođe pre ostalih, onda će i da izađe pre ostalih. Formalnije rečeno, sortirani podaci mogu biti predstavljeni kao zapis ili kao vrednosti entorke, a deo podataka koji se koristi za sortiranje se zove ključ. U primeru sa kartama, karte su predstavljene kao vrednost (rang, iste boje) a ključ je rang.

- Algoritam za sortiranje je stabilan kad god postoje dve vrednosti P i S sa istim ključem, i P pojavi pre S u prvobitnom spisku, onda P ostaje ispred S u sortiranoj listi.

- Stabilnost ne predstavlja problem kada se elementi mogu razlikovati, kao što je kod celih brojeva, ili uopšte svi podaci gde su svi elementi ključ, stabilnost ne predstavlja problem. Stabilnost takođe nije problem ako su svi ključevi različiti.

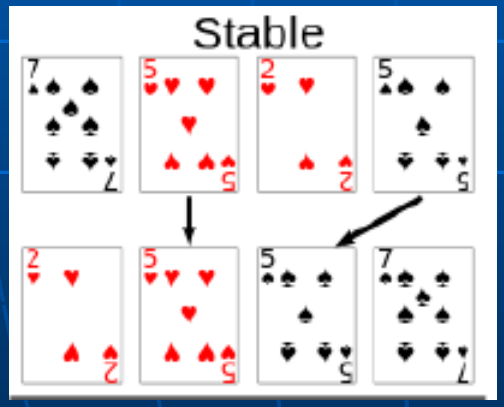
Uporedjenje algoritama sortiranja

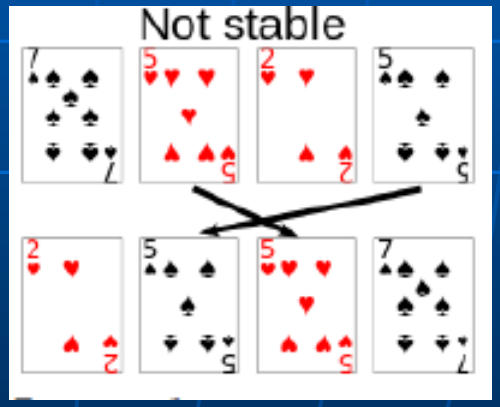
- U tabeli, n je broj elemenata niza koji će biti sortirani. Redovi "Prosečan slučaj" i
- "Najgori slučaj" daće nam uvid u vremensku složenost svakog sortiranja posebno (koju smo već opisali u predjašnjim poglavljima a tu ga koristimo samo radi uporedjenja), pod pretpostavkom da je dužina svakog niza elemenata jednaka, te da stoga sva uporedjenja, zamene, i druge potrebne operacije može napraviti u konstantnom vremenu.
- "Memorijski prostor" označava količinu pomoćnog prostora za pamćenje i izmene niza elemenata, pod istim pretpostavkama. Vreme izvođenja i memorijski prostor koji su navedeni u nastavku treba shvatiti da se nalaze unutar velikog O zapisa.

Ime	Najbolji slučaj	Prosječan slučaj	Najgori slučaj	Memorijski prostor
Selection	n^2	n^2	n^2	1
Insertion	n	n^2	n^2	1
Merge	n log n	n log n	n log n	n-najgori slučaj
Bubble	n	n^2	n^2	1
Quicksort	n log n	n log n	n^2	log n-prosječni slučaj, n-najgori slučaj

Stabilnost algoritama

- Nestabilni algoritmi sortiranja mogu biti prerađeni u stabilne. Jedan od načina da se to uradi je da se veštački produži poređenje ključeva, tako da poređenja između dva objekta sa jednakim ključevima koriste redosled unosa u originalnim ulaznoj listi kao u "taj-brejku".
- Međutim, pamćenje ove naredbe može da zahteva dodatno vreme i prostor. Jedna aplikacija za stabilne algoritme sortiranja, sortira listu koristeći primarni i sekundarni ključ.
- Na primer, želimo da sortiramo karte iz ruke po bojama i to po redosledu: detelina (♣), karo (♦), srce (♥), pik (♠), a unutar svakog znaka, karte budu sortirane po rangu. Ovo se može uraditi tako što se prvo sortiraju karte po rangu (koristeći bilo kakvo sortiranje), a zatim radi stabilno sortiranje znakova:

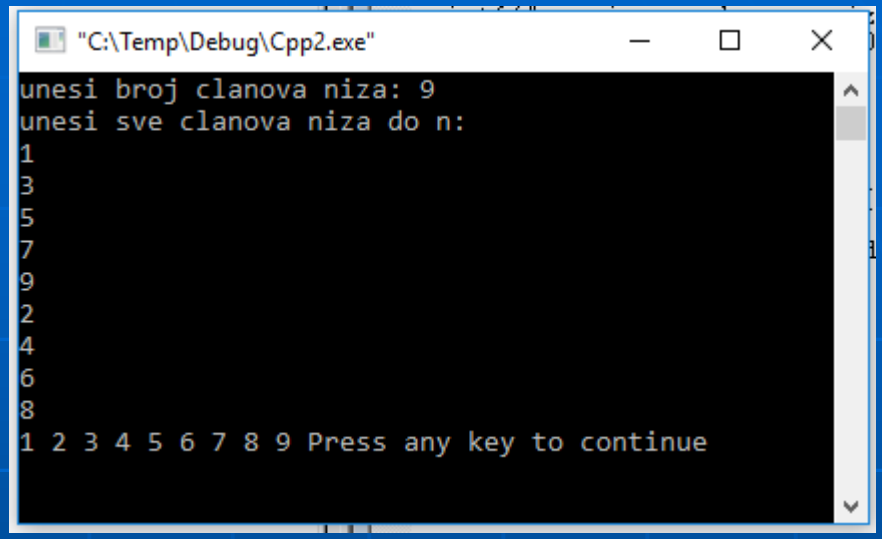




Unutar svakog znaka, stabilno sortiranje čuva redosled po rangu, koje je već urađeno. Ova ideja se može proširiti na bilo koji broj ključeva, a urađeno je uz pomoć osnovnog sortiranja. Isti efekat se može postići i sa nestabilnim sortiranjem pomoću poređenja leksiko-grafičkih ključeva, koje na primer poredi prvo znakove, a zatim upoređuje po rangu ako su znakovi isti.

Primer 1 u C_u: Kreirajte modifikovanu verziju count2 counting sort-a tako što cete odrediti min i max niza a. Prebrojavanje pojavljivanje članova niza a pamtite u pomoćnom nizu o prolaskom kroz elemente niza u intervalu od min do max.

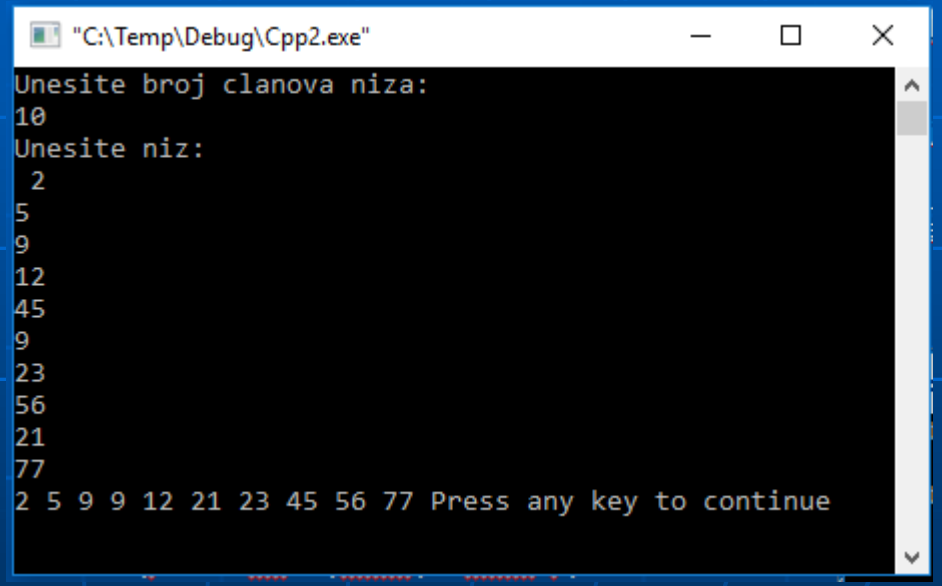
```
/* program count2;*/
#include<stdio.h>
#include<limits.h>
#define MAX 200
main()
{ int i,j,n,min,max,k;
int a[MAX],o[4*MAX];
printf("unesi broj clanova niza: ");
scanf("%d",&n);
min=INT_MAX;max=0; k=0;
printf("unesi sve clanova niza do n:\n");
for (j=0;j<=4*MAX;j++) o[j]=0;
for(i=1;i<=n;i++)
{scanf("%d",&a[i]);
o[a[i]]+=1;
if (a[i]>max) max=a[i];
if (a[i]<min) min=a[i];}
for (i=min;i<=max;i++) if (o[i]!=0) for (j=1;j<=o[i];j++)
{ k=k+1; a[k]=i; }
for (i=1;i<=n;i++) printf("%d ",a[i]); }
```



Primer 2 u C_u: ***Radix sortiranje*** koristi pozicionu reprezentaciju broja (ili niske karaktera), te odvojeno analizira cifre (ili znakove) na raznim pozicijama. Za demonstraciju ideje algoritma, bez gubitka opštosti, pretpostavlja se da su ključevi decimalni brojevi sa istim brojem od *k* cifara $d_{k-1}\ d_{k-2}\ ..\ d_0$.

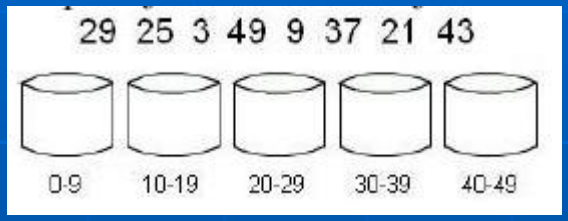
```
/* program Radix;*/
#include <stdio.h>
#include <math.h>
typedef int niz[100];
int cifra(int a,int mesto);
void prolaz(niz a,int n,int mesto);
void radixSort(niz a,int N);
main()
{niz x; int i,n;
printf("Unesite broj clanova niza: \n");
scanf("%d",&n);
printf("Unesite niz:\n");
for(i=1;i<=n;i++)scanf("%d",&x[i]);
radixSort(x,n);
for(i=1;i<=n;i++)printf("%d ",x[i]);}

int cifra(int a,int mesto)
{ int i;
for(i=1;i<=mesto-1;i++)a=a/10;
return(a%10);}

void prolaz(niz a,int n,int mesto)
{ int pc[10]; niz pomocni; int i,d;
for(d=0;d<=9;d++)pc[d]=0;
for(i=1;i<=n;i++)pc[cifra(a[i],mesto)]+=1;
for(d=1;d<=9;d++)pc[d]=pc[d]+pc[d-1];
for(i=n;i>=1;i--)
{pomocni[pc[cifra(a[i],mesto)]]=a[i];
pc[cifra(a[i],mesto)]-=1;}
for(i=1;i<=n;i++)a[i]=pomocni[i];}
```

```
void radixSort(niz a,int N)
{ int p;
for(p=1;p<=5;p++)prolaz(a, N, p);
// pod pretpostavkom INT_MAX 32767
}
```

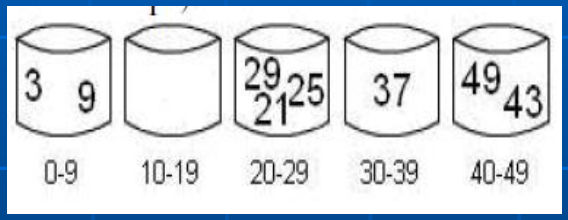


Bucket sort

- Bucket sort (bin sort) razvrstava elemente niza u korpice. Ovo je, takodje, metoda sortiranja koja nije zasnovana na poredjenju (tzv. nekomparativni metod, non-comparison sort).

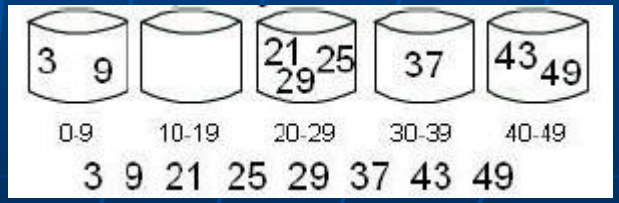


- U prvoj fazi, inicalizuje se niz praznih korpi na 0.



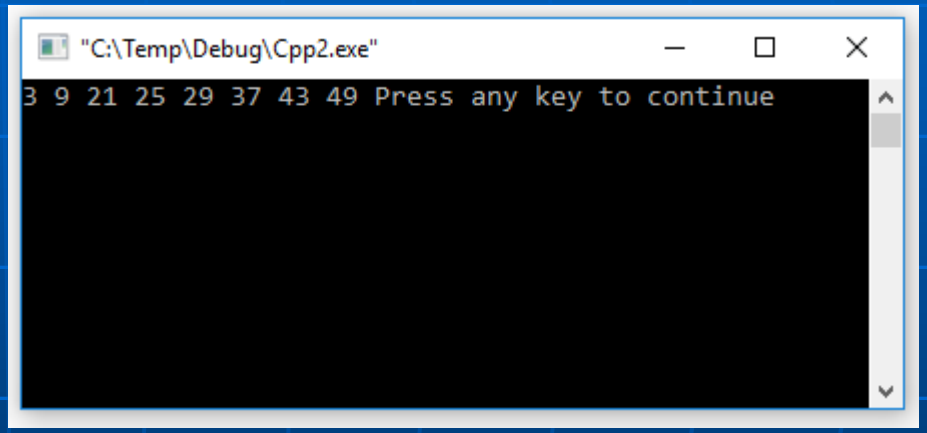
- Potom se prolazi kroz niz elemenata i svaki element se stavlja u svoju korpu (vrednost člana niza je adresa korpe. 10-19 prazna).

- U svakoj korpi su elementi sortirani i na kraju se elemnti prepisuju iz korpi u originalni niz.



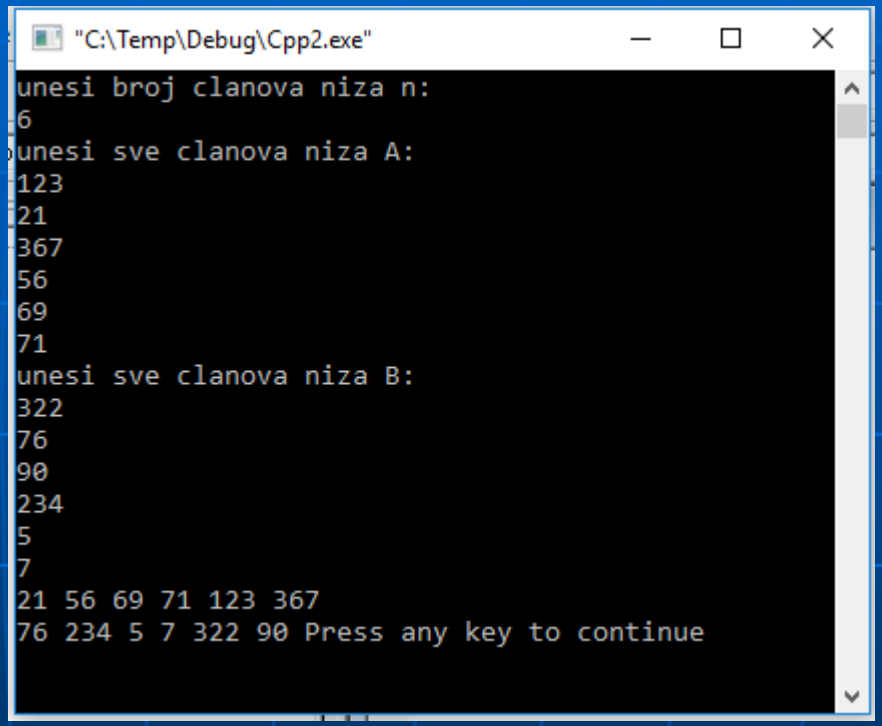
Bucket sort

```
#include <stdio.h>
#define NMAX 8
void bucketSort(int a[])
{
int i, j;
int count [7*NMAX]={0};
for (i = 0; i<NMAX; i++) count[a[i]]++;
for (i = 0, j = 0; i<7*NMAX; i++)
for (; count[i] > 0; (count[i])--)
a[j++] = i;
}
int main()
{
int a[] = { 29, 25, 3, 49, 9, 37, 21, 43 }, i;
bucketSort(a);
for (i=0; i<NMAX; i++) printf("%d ", a[i]);
return 0;
}
```



Primer 3 u C_u: Prvi red standardnog ulaza sadrzi prirodan broj N (1 <= N <= 100 000) koji predstavlja broj elemenata niza A i B. Drugi red sadrzi N prirodnih brojeva, razdvojenih jednim znakom razmaka, koji predstavljaju elemente niza A. Naredni red sadrzi N prirodnih brojeva, razdvojenih jednim znakom razmaka, koji predstavljaju elemente niza B. (-10^9 <= A[i], B[i] <= 10^9)

```
#include <cstdio>
#include <algorithm>
#define MAXN 100000
using namespace std;
struct s {int a, b;}
niz[MAXN];
bool cmp(s x, s y)
{return x.a < y.a;}
int main()
{int n,i;
printf("unesi broj clanova niza n:\n");
scanf("%d", &n);
printf("unesi sve clanova niza A:\n");
for (i = 0; i < n; i++)
scanf("%d", &niz[i].a);
printf("unesi sve clanova niza B:\n");
for (i = 0; i < n; i++)
scanf("%d", &niz[i].b);
stable_sort(niz, niz+n, cmp);
for (i = 0; i < n-1; i++)
printf("%d ", niz[i].a);
printf("%d\n", niz[n-1].a);
for (i = 0; i < n; i++)
printf("%d ", niz[i].b);
return 0;}
```



Rešenje:
Nama je potrebno da u ovom zadatku iskoriistimo stabilnost algoritma sortiranja. U biblioteci STD postoji implementacija stabilnog algoritma za sortiranje **stable_sort** koji garantuje da će elementi sa istom vrednosti biti poredjani onim redosledom u kom su bili pre sortiranja.