

(Predavanje I)

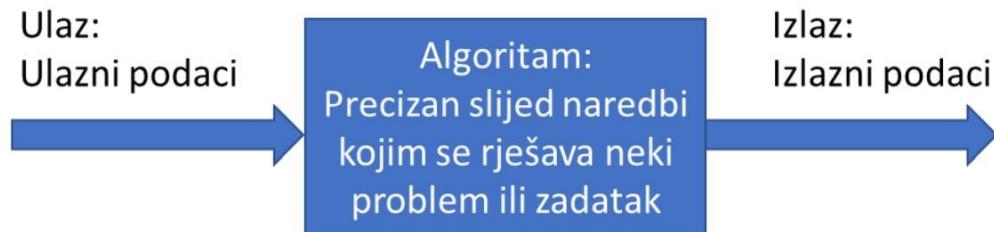
Algoritmi i strukture podataka

- Muhamed ibn Musa al Horezmi – IX stoljeće – „Račun o Hindu brojkama“, pravila za obavljanje aritmetičkih operacija nad brojevima zapisanih u dekadnom sistemu
- Njegova knjiga je u XII stoljeću prevedena na latinski jezik - Algoritmi de numero indorum, u ovom dijelu je upotrebljena riječ Algoritam što se u prijevodu odnosi na ime muslimanskog naučnika, nadalje se ova riječ koristi pri manipulaciji sa indijskim brojevima koji se zbog utjecaja ove knjige nazivaju arapskim brojevima
- Algoritam je precizno definirana procedura u obliku uređenog skupa koraka kojim se definira redoslijed izvođenja procedure
- **Svaki algoritam treba da se sastoji od:**
 - Definiranog ulaza (koji se sastoji od objekata)
 - Definiranog izlaza (objekti, odnosno rezultat)
 - Konačnog broja koraka
 - Procedura odnosno koraka procedure koji moraju se izvršiti u konačnom vremenu
 - Naredbe moraju biti jasne i nedvosmislene, tako da se može izvesti svaki korak algoritma sa konačnim brojem instrukcija

Značaj algoritma i strukture podataka

- Struktura podataka je način organiziranja podataka i informacija
- Algoritam je način manipulacije i obrade podataka
- Pisanje koda zavisi od ove dvije stvari: izbor prikladnih struktura podataka i dizajn efikasnog algoritma
- Za dizajn i razvoj složenijih softverskih proizvoda, koji će svoju upotrebu naći u stvarnom svijetu, je potrebno koristiti mnoge tehnike dizajniranja, standarde kodiranja, metode,...
- Kada programi postaju veći pored kodiranja treba se usmjeriti i na druge aspekte koji se odnose na dizajn softvera, odnosno treba voditi računa o softverskom inženjeringu
- **Životni ciklus nekog softvera**
 - analiza problema;
 - definicija zahtjeva;
 - projektiranje na visokom i niskom nivou;
 - implementacija;
 - testiranje i verifikacija;
 - isporuka softvera;
 - upotreba softvera;
 - održavanje softvera.
- **Ciljevi dobro napravljenog softvera**
 - Mora ispuniti projektni zadatak. Precizno i jasno definiranje problema koji treba da ispuni. Zadatak se sastoji nekada od stotina stranica koje softver treba da nakon implementacije zadovolji.
 - Softver treba da bude prilagodljiv. Promjene od strane naručioca, promjene nastale u toku kodiranja, promjene nastale u toku testiranja. Često se pojavi problem u fazi produkcije koji treba da se riješi (neotkriven u fazi testiranja).

- **Lakša manipulacija uz činjenice:**
 - Program bi trebao biti čitljiv, razumljiv i dobro dokumentiran.
 - Velike programe treba podijeliti na manje logičke cjeline, koje su relativno nezavisne jedna od druge.
 - Kvalitetan softver treba biti višekratno iskoristiv.
 - Jedan od načina da se uštedi vrijeme za dizajniranje softverskih rješenja je ponovna upotreba programa, klasa, funkcija i drugih komponenti koje su razvijene u prethodnim projektima.
 - Kvalitetan softver se treba završiti u predviđenim rokovima
- **Generički prikaz algoritma**



Algoritmi i strukture podataka

- **Izvršavanje algoritma:**
 - Paralelno – više procesora izvršava svoj zadatak – svaki svoju nit (engl. threads) – uslovna paralelnost – koraci se izvršavaju istovremeno
 - Sekvencionalno – korak po korak
- Algoritamski proces su deterministički – svaki korak algoritma ima svoje međurezultate u većini slučajeva
- Algoritmi se implementiraju prevođenjem instrukcija, odnosno koraka u kompjuterski program koji se izvodi na kompjuteru (računaru)
- Ovaj proces se naziva programiranje
- Iako je programiranje važan dio kompjuterskih nauka, kompjuterske nauke ne studiraju programiranje niti programske jezike nego ih inženjeri koriste da bi riješili neki problem
- Pisanje programa u programskim jezicima zamagljuje osnovu ideju kojom se rješava neki problem, pa se algoritmi često zapisuju pomoću pseudojezika ili danas sve rjeđe blokovskom predstavom
- Teorijski algoritmi su algoritmi koji rješavaju matematičke probleme
- Eksperimentalni algoritmi su algoritmi koji povezuju neke objekte, procese ili mehanizme u cilju rješavanja nekog problema
- Konstrukcijski algoritmi – konstrukcija novih algoritama na temelju nekih starih rješenja – sličnost problema
- Korištenje pseudojezika zadržavat će nas na osnovnim idejama, zamislima i suštini algoritma
- While, repeat, for – kontrolna struktura koja se završava sa end + ime strukture, pa tako end_while, end_for
- Struktura izbora if se završava sa end_if
- Dodjela vrijednosti ↓, ako se na primjer varijabli a dodjeljuje vrijednost b; a ↓ b... Kako bi bila višestruka dodjela a, b i c?

- Razmjena vrijednosti između varijabli . Npr. a varijabla i b treba da se izvede razmjena vrijednosti... priv
 \Downarrow a, a \Downarrow b, b \Downarrow priv.
- Pristup elementima niza A[5], indeks označava pristupanje 6 elementu s obzirom da je prvi indeks jednak 0,
- A[2..6], koristimo za pristup nizu elemenata, odnosno za pristup A[2], A[3], A[4], A[5], A[6]
- Složeni podaci su organizirani u objekte, koji se sastoje od atributa
- Atributima se prema pseudojeziku koji će se koristiti u ovom kursu pristupa sa navođenjem imena a zatim navođenjem objekta u uglatim zagradama. Npr. atribut duzina pokazuje koliko se elemenata nalazi u nekom nizu duzina[A] – broj elemenata u nizu A
- Varijabla koja predstavlja niz ili objekat A se tretira kao pokazivač na podatke u nizu. Ako npr. dodijelimo y-onu x tj. y \Downarrow x tada postižemo da se sva polja dodjeljuju nizu y, odnosno nakon ove operacije pokazivač pokazuje na isti objekat. Vrijednost pokazivača koji ne pokazuje na objekat ne NULL.
- Parametri se iz funkcije prenose po vrijednosti i po referenci. Kod funkcija koje vraćaju vrijednost izračunavanjem koristi se return
- Pokazivači – pitanje šta su???
- Nekom polju zapisa na kojeg pokazuje pokazivač pristupamo navođenjem imena polja, a zatim imena pokazivača u malim zagradama. Na primjer, ako neki zapis, na koji pokazuje pokazivač p, sadrži polja a, b i c, onda pojedinim poljima pristupamo navođenjem a(p), b(p) i c(p), respektivno.
- Logički izrazi se označavaju sa „and“ ili „or“ ili „not“
- Izlazi iz logičkih izraza su „true“ ili „false“
- Npr. x and y – prvo se vrednuje x i provjerava da li je istinit (true), ako jeste onda se vrednuje y i provjerava da li je istinit (true),
- Probati sa x or y ili not x
- ERROR, PRINT, GETNODE (alokacija memorijskog prostora za neki element u dinamičkoj strukturi), FREENODE (oslobađanje memorijskog prostora)

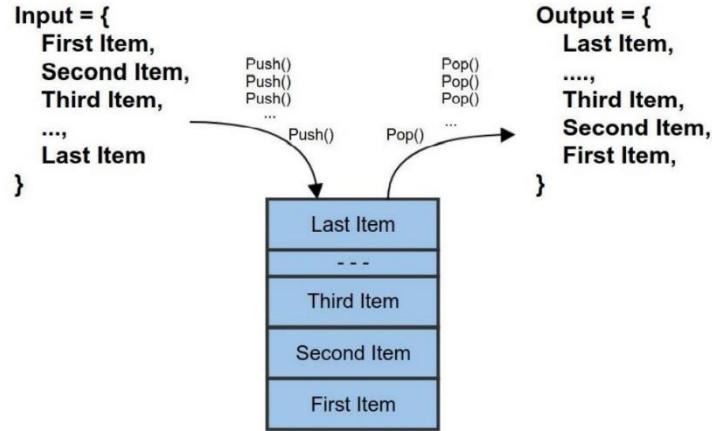
Strukture podataka

- **Elementarni tipovi podataka su vrijednosti tipa:**
 - cjelobrojnog
 - realnog
 - logičkog
 - znakovnog
- Osnovna im je značajka da se ne mogu dijeliti u podtipove, kreiraju apstraktne tipove podataka
- Apstraktni tipovi podataka:
- Ovim tipom se definišu definicije vrijednosti i definicije operacija
- Moraju se osnovni tipovi manipulacijom prevesti programskim jezikom u apstraktni tip podataka
- Skupovi koji se sastoje od ovih manipulatora se nazivaju strukture podataka (mogu da sadrže i skup operacija)
- Logički povezani elementi, odnosno strukture se objedinjuju u logički povezane elemente: nizove i zapise
- Nizovi su homogene strukture koji sadrže elemente istog tipa
- Zapisi su strukture koje se sastoje od logičnih struktura istih ali i različitih elemenata
- **Klasifikacija struktura podataka:**
 - Linearnost

- Linearna struktura je struktura u kojoj su elementi međusobno povezani na način da je jedan element povezan sa prethodnikom i sljedbenikom – nizovi, povezana lista, stek, red, itd.
 - Nelinearna struktura je struktura u kojoj su elementi u relaciji sa više elemenata – stablo, graf
- **Promjena osnove veličine:**
 - Statička struktura – veličina se ne mijenja u toku izvršenja programa, memorijski resursi se moraju unaprijed alocirati iz čega proističe da ova struktura ne omogućuje optimiziranu alokaciju memorije
 - Dinamička struktura – veličina se može mijenjati u toku izvršenja programa, omogućuje efikasno korištenje memorijskih resursa
 - **Statička struktura:**
 - Polja (nizovi – jednodimenzionalna struktura ili vektori)
 - (1, 1, 2, 3, 5, 8, 13, ... – Fibonacci-jev niz npr.)
 - Dvodimenzionalna (matrice)
 - $$\begin{matrix} 1 & 3 & 13 \\ 1 & 5 & 21 \\ 2 & 8 & 24 \end{matrix}$$
 - Višedimenzionalna (tenzori)
 - Često se memorija alocira uzastopno
 - **Sadrži:**
 - Naziv polja
 - Adresu prvog elementa polja
 - Najmanje i najveće vrijednosti indeksa
 - Tip elementa polja
 - Broj lokacija po jednom elementu polja (byte)
 - **Polustatičke strukture** – dodavanja i uklanjanje i pristup elementima je moguć samo na tačno određenim mjestima u strukturi. Zašto se ovo koristi?
 - Ovo su linearne strukture, a razlikuju se samo po mjestu pristupa elementima
 - Stek – princip LIFO (Last In First Out)
 - Red – princip FIFO (First IN First Out)
 - Dek

Stek

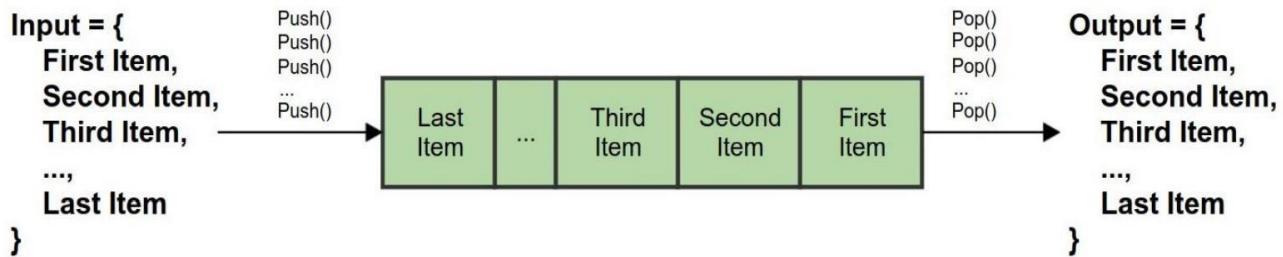
- Za element x_n kaže se da je na vrhu stek, a za element x_1 se kaže da je na početku steka. Neki element u steku je x_i .
- Pristup steku se ostvaruje logičkom operacijom POP i na taj način se pristupa ne prvom elementu nego elementu koji je na vrhu steka
- Operacija PUSH u osnovi služi za dodavanje elemenata steku
- Uklanjanje elemenata sa steka se vrši također sa vrha steka DELETE i odnosi se na zadnji element koji je dodat u stek
- Iz ovoga razloga ovaj princip se zove LIFO



- **Organizacija steka može biti:**
 - Sekvencionajna
 - Spregnuta
- Sekvencialna se odnosi na takvu organizaciju koja se može predstaviti pomoću fizičkog vektora (niza memorijskih lokacija) dopunjenoj deskriptorom
- Kod spregnute realizacije steka memoriski prostor dodijeljen elementima steka se dopunjuje sa pokazivačem koji se uvećava za jedan za svaki slijedeći element
- Posljedica je da kod ove realizacije fizički možemo dodati element proizvoljno

Red

- Ova struktura je slična steku
- Ova struktura je linearna

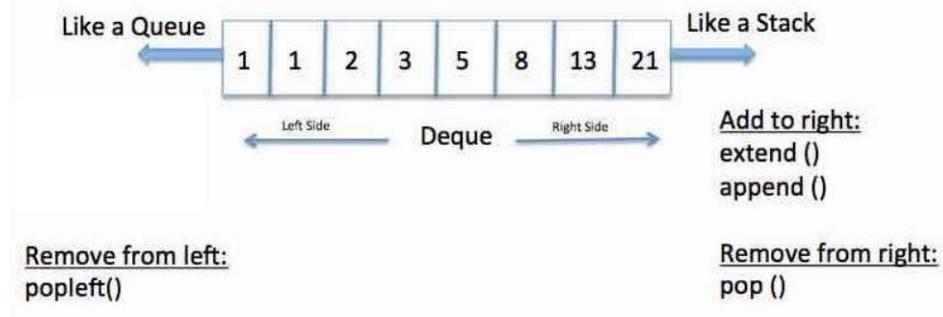


- **Organizacija reda može biti:**
 - Sekvencionajna
 - Spregnuta
- Sekvencialna se odnosi na takvu organizaciju koja se može predstaviti pomoću fizičkog vektora (niza memorijskih lokacija) dopunjenoj deskriptorom
- Kod spregnute realizacije steka memoriski prostor dodijeljen elementima steka se dopunjuje sa pokazivačem koji pokazuje na prvi element i od pokazivača koji pokazuje na posljednji element u redu
- Posljedica je da kod ove realizacije je da se fizički uklanja prvi element, a dodaje posljednji koji je u nizu

DEK (engl. deque)

- Red sa dva kraja

- Predstavlja sintezu elemenata steka i reda



- Operacije na deku su:**

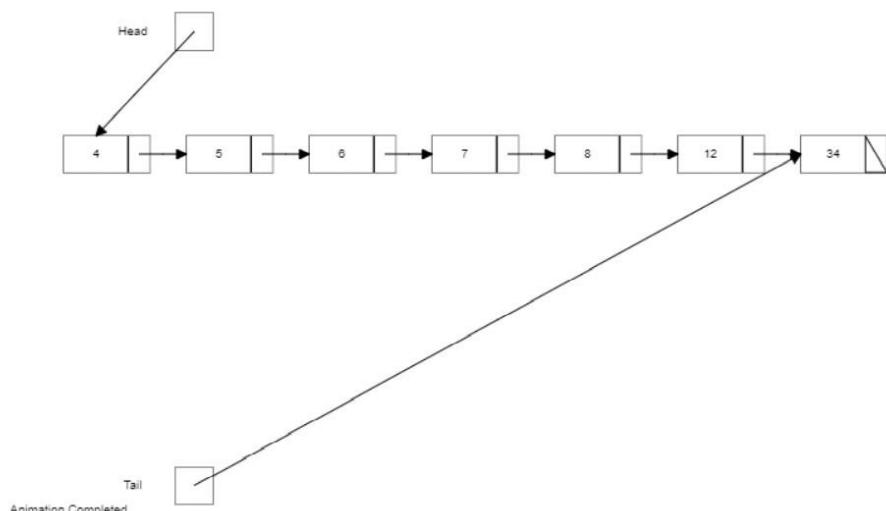
- FRONT (prvi)
- REAR (zadnji)
- POPF (ukloni prvi)
- POPR (ukloni zadnji)
- PUSHF (dodaj prvi)
- PUSHR (dodaj posljednji)

Dinamičke strukture podataka

- Osnovna karakteristika ovih struktura je mogućnost pristupa svakom elementu (čvoru) u strukturi bez mjenjenja strukture, kao i mogućnost dodavanja i uklanjanja čvorova bez ikakvih ograničenja
- Broj čvorova je teoretski neograničen, a u fizičkoj strukturi se logički susjedni čvorovi često ne nalaze na fizički susjednim memorijskim lokacijama
- Osnovne dinamičke strukture su liste i stabla

Dinamičke strukture podataka - Liste

- Jednostruko spregnute liste (linearne liste)
- Dvostruko spregnute liste
- Višestruke liste

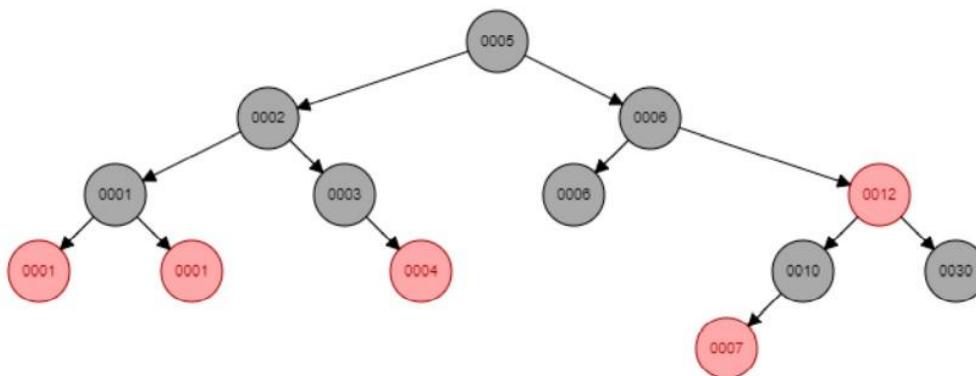


Jednostruko spregnute liste (linearne liste)

- Struktura je linearna (svaki čvor ima svog prethodnika osim ako se radi o jednom čvoru... Imaju li dva čvora prethodnika i sljedbenika???). IP je indeks početka – početni čvor
- Moguć je pristup elementu bez promjene strukture
- Moguće je dodavanja elemenata iza i ispred svakog elementa osim kod zatvorenih listi
- Uklanjanje elemenata je dozvoljeno osim IP-a
- Pristupanje elementu počinje od IP-a, a zatim se pokazivač uvećava dok se ne dođe do željenog elementa ili konstatira da takav element ne postoji
- **Značajnije operacije**
 - Određivanje broja elemenata u listi
 - Formiranje kopije liste
 - Spajanje dvije ili više lista u jednu
 - Podjela liste na dvije ili više
 - Sortiranje liste
- **Deskriptorsko polje se sastoji od:**
 - Koda strukture podataka
 - Naziv liste
 - Indeks početka
 - Opis elemenata
 - Broj elemenata u listi
- Jednostruko spregnuta lista se može organizirati kao cirkularna lista – pokazivač pokazuje na posljednji te ujedno i prvi element

Dinamičke strukture podataka – Stablo - Tree

- Najčešće korištena dinamička nelinearna struktura sa sljedećem osobinama:
 - Svaki element ima najviše jednog neposrednog prethodnika
 - Postoji tačno jedan element bez neposrednog prethodnika, koji se naziva korijen stabla
- Elementi koji nemaju sljedbenike nazivaju se listovi



- Neki pojmovi za strukturu podataka Stablo

- Ako se formira stablo kao x_1, x_2, \dots, x_n i označava put dužine $n-1$
- Pod redom stabla podrazumijeva se najveći broj sljedbenika nekog elementa
- Broj elemenata najdužeg puta se naziva visina stabla
- Podstablo je svaki element stabla sa svojim sljedbenicima

- **Funkcije kod Stabla su:**

- Sljedeći
- Izaberi – za skup sljedbenika omogućava izbor jedinstvenog sljedećeg elementa na osnovu nekog kriterijuma
- Korijen – zadatak da odredi korijen stabla
- Pristupi

- **Strukture podataka smještenih na masovnu memoriju**

- Podaci na masovnim memorijama se organiziraju kao datoteke
- Datoteka predstavlja uređeni skup slogova (zapisa)
- Skup slogova o studentima na fakultetu predstavlja datoteku studenata
- SLOG – uređeni skup srodnih polja koji predstavljaju jedinicu (broj indeksa, ime i prezime studenta, godina studija, smjer)
- POLJE – osobina entiteta za posmatrani slog – predstavlja elementarni podatak

- **Memorijski prikaz struktura podataka**

- Sekvencijalni (kontinualni) prikaz
- Ulančeni prikaz
- Kod sekvencijalnog prikaza, elementi strukture su raspoređeni u kontinualnom memorijskom prostoru. Elementi mogu biti istog ili različitog tipa. Za pristup elementima treba poznavati početnu adresu, indeks pozicije i veličinu pojedinih elemenata. Sekvencijalne strukture se mogu koristiti i za prikaz nelinearnih struktura
- Kod ulančenog prikaza elementi strukture su raspoređeni na proizvoljnim mjestima u memorijskom prostoru
- Fizički poredak se ne poklapa sa logičkim poretkom elemenata
- Pokazivači kao elementarni tip podataka koriste se da označe logički poredak
- Svaki element strukture, pored informacionog sadržaja, ima jedan ili više pokazivača, koji sadrže adrese nekih drugih elemenata strukture.
- Ulančeni prikaz je pogodan za nelinearne strukture podataka iako se mogu sa njim prikazati i linearne strukture radi bolje alokacije memorije
- Sporiji pristup naspram bolje alokacije resursa
- Ali s druge strane potrebno je osigurati dodatnu memoriju za memorisanje pokazivača
- Ipak, ulančani prikaz omogućuje efikasnije korištenje memorijskih resursa, zbog korištenja dinamičke alokacije i dealokacije memorije.
- Često se ovi prikazi kombiniraju u toku nekog programskog koda

- **Operacije sa strukturama podataka**

- pristup elementima radi čitanja
- dodjela novih vrijednosti,

- obilazak svih elemenata,
 - pretraživanje,
 - umetanje novog elementa,
 - Brisanje elementa strukture,
- Obilazak elemenata – pristupanje elementima strukture na sistematičan način (obično jednom) – linearna se lakše obilazi od nelinearne strukture
- Pretraživanje – pronalaženje elementa u strukturi sa traženim sadržajem – obilazak sa poređenjem elemenata sa traženim sadržajem – kada se pronađe element može se operacija završiti ili ako se ne pronađe rezultata je neuspješno pretraživanje
- Ako su strukture uređene ne mora se provjeriti svaki element nego se nekom operacijom može pronaći odgovarajući element
- Umetanje novog elementa – operacija u kojoj se ne postavlja zahtjev za premještanjem već prisutnih elemenata u strukturi podataka
- Lakša je implementacija umetanja u neuređenu strukturu podataka
- Zbog nekog redoslijeda u strukturi podataka teža je implementacija umetanja u ovom slučaju zbog činjenice da se i nakon umetanja mora održati sadržaj prije ove operacije
- Brisanje elementa – operacija u kojoj se zahtjeva da pri njenom obavljanju ne dolazi do pomjeranja već prisutnih elemenata u strukturi podataka
- Ovaj zahtjev je lakše ispuniti kod ulančanog prikaza, jer se fizičkim uklanjanjem elementa koji se briše, ne narušava nekontinualni karakter memorijskog prikaza, a povezivanje razdvojenih dijelova strukture se svodi na preusmjeravanje pokazivača.
- S druge strane, kod sekvencijalnog prikaza, nakon fizičkog uklanjanja elementa koji se briše, narušava se kontinualni karakter memorijskog prikaza, pa se obično preostali elementi moraju premještati da bi se struktura nakon brisanja reorganizirala na način da se popune upražnjene lokacije u strukturi, te na taj način obnovi njen kontinualni karakter.

• **Primjer strukture podataka**

- Sklapanje automobila (naziv, količina, šifra).
- Relacije između slogova:
- Elementi a i b su u relaciji r1 ako se dio b ugrađuje u a Elementi a i b su u relaciji r2 ako im je broj na zalihamama manji od 4 i ako je broj dijelova a na zalihamama manji ili jednak broju dijelova b

- Primjer strukture podataka
- $S=\{k1, k2, k3, \dots, r1, r2\}$
- Elementi u S su:
- $k1[\text{automobil}, 40, 612]$
- $k2[\text{motor}, 2, 802]$
- $k3[\text{blok}, 3, 105]$
- $k4[\text{filter}, 2, 117]$
- $k5[\text{pumpa}, 10, 118]$
- $k6[\text{klip}, 60, 230]$
- $k7[\text{mijenjač}, 30, 406]$
- $k8[\text{karoserija}, 17, 507]$
- $k9[\text{vrata}, 30, 230]$
- $k10[\text{gepek i hauba}, 1, 702]$

• Relacije

- $r1[(k1, k2), (k1, k8), (k2, k3), (k2, k7), (k3, k4), (k3, k5), (k5, k6), (k8, k9), (k8, k10)]$
- $r2[(k10, k4), (k4, k2), (k2, k3)]$



(Predavanje II)

Analiza složenosti algoritma

- Pri rješavanju nekog problema često može postojati više algoritama koji predstavljaju rješenje za taj problem.
- Analizom vremenske složenosti nekog algoritma određujemo trajanje algoritamskog procesa, pri čemu nas ne zanima stvarno vrijeme izvođenja (milisekunde, sekunde, minute, itd.) algoritma, nego trajanje algoritamskog procesa izražava se aproksimacijom broja elementarnih operacija koje će taj algoritam izvesti.
- Također, kada analiziramo efikasnost nekog algoritma, najčešće nas zanima u koju klasu stupnja rasta vremenske složenosti taj algoritam spada, a ne tačan broj operacija koje će taj algoritam izvršiti

- **Treba analizirati slijedeći algoritam:**

- ČOKOLADICANAPOLICI(n)
- Za $i \leftarrow n$ uzmi 1
- Čitaj „i čokoladica je na polici“
- Čitaj „uzmi čokoladicu sa police, dodaj kupcu, $i-1$ čokoladica na polici“
- Provjeri „Nema čokoladica na polici“
- Izvrši „Iди у набавку n čokoladica 😊, n čokoladica je na polici“
- Kako se može izmjeriti vrijeme izvršavanja?
- Očigledno je da je ono funkcija ulazne veličine n
- S obzirom na vrijeme može se reći da je različito vrijeme čitanja gore navedenog problema od osobe do osobe
- Tehnologija proširuje mogućnosti
- Diktiranje teksta ima svoj jedinstven takt
- Korištenje Morse-ove abecede uzet će minutu vremena za svaki korak
- Ako se download-uje ovaj tekst uzet će se desetina sekunde za korak
- Važno je primjetiti da sa porastom n , raste i vrijeme izvršavanja, ovo se reflektira u asimptotskom vremenu $O(n)$

EKVIVALENTNI ALGORITMI

- Treba dizajnirati algoritam za zbrajanje prvih n (prirodnih) brojeva
- ULAZ: n
- IZLAZ: $s = 1 + 2 + 3 + \dots + n$, $n \geq 1$

PRVO RJEŠENJE:

SUMA (n)

```
s ↓ 0
for i ↓ 1 to n do
    s ↓ s + i
end_for
return s
```

- Suma prirodnih brojeva
- num = input („Unesite pozitivan broj: “)
- if num < 0:


```
print(„Unesite pozitivan broj“)
else:
```
- sum = 0
- while(num > 0):


```
sum += num
      num -= 1
```
- print(„Suma je, sum)

EKVIVALENTNI ALGORITMI

- Treba dizajnirati algoritam za zbrajanje prvih n brojeva
- ULAZ: n
- IZLAZ: $s = 1 + 2 + 3 + \dots + n$, $n \geq 1$

DRUGO RJEŠENJE:

- **SUMA (n)**
- **$s = n*(n+1)/2$**
- **return s**
- Ovdje se primjenjuje matematička relacija za sumu aritmetičkog niza:
- **$S = \frac{n \cdot (n+1)}{2}$**
- Algoritmi u oba slučaju daju isti rezultat ali postoji jedna ključna razlika. Koja?

TRAJANJE ALGORITAMSKOG PROCESA

- U prvom algoritmu for/while petlja se izvršava n puta, pa ukupan broj operacija koje se izvršavaju unutar petlje zavisi o veličini broja n.
- U drugom algoritmu je potrebno obaviti jedno zbrajanje, jedno množenje, jedno dijeljenje, te jednu operaciju dodjele vrijednosti, bez obzira na veličinu broja n.
- Prema tome, trajanje algoritamskog procesa prvog algoritma je proporcionalno sa n, dok je trajanje drugog algoritamskog procesa konstantno bez obzira na veličinu broja n.

Kriteriji za ocjenu algoritma:

- Jednostavnost implementacije vodi do dva važna resursa:
 - Vrijeme
 - Memorijski prostor
- Uzimajući u obzir da je tehnološki napredak u proizvodnji različitih tipova memorije omogućio da memorija predstavlja sve manje ograničenje, vrijeme algoritamskog procesa zapravo postaje ključni kriterij pri dizajniranju i izboru algoritama.
- **Vrijeme**
- Ne u milisekundama, sekundama ili minutama
- Aproximacijom broja elementarnih operacija, koje algoritam treba da izvede
- Efikasnost se mjeri vremenskom složenošću algoritma

- **Trajanje algoritamskog procesa zavisi od slijedećih faktora:**

- tehnoloških parametara koji zavise od konkretne arhitekture i organizacije računara na kojem se algoritam izvršava;
- kvalitete generiranog mašinskog koda od strane prevoditelja;
- ulaznih podataka;
- vremenske složenosti algoritma.
- Prva dva kriterija: tehnološki parametri koji zavise od konkretne arhitekture i organizacije računara na kojem se algoritam izvršava i kvalitete generiranog mašinskog koda od strane prevoditelja;
- Uključuju: računarske arhitekture sa frekvencijama takta, performansama, intuitivna grafička sučelja, ali i računarske mreže i objektno-orientirane sisteme
- Da li je vrijeme izvršavanja algoritma ključan kriterij i uz ovu tehnologiju koja se stalno usavršava
- Za funkcionalisanje svega navedenog treba osmisiliti efikasne algoritme
- Iako aplikacije čak i ne uključuju algoritamske sadržaje, one se zapravo temelje na algoritmima zbog sljedećih razloga:
- Dizajn i implementacija bilo kojeg grafičkog sučelja se u biti temelje na algoritmima.
- Rutiranje u računarskim mrežama se temelji na algoritmima.
- Aplikacije su pisane u nekom od programskih jezika, a to znači da će se koristiti kompjajler, interpreter ili možda asembler, koji se u velikoj mjeri temelje na algoritmima.

- **ALGORITMI SU JEZGRA VEĆINE TEHNOLOGIJA**

- Prije analize vremenske složenosti algoritama, potrebno je usvojiti generički model koji pomaže u usporedbi algoritama
- **Jednoprocesorski sistem sa RAM modelom izračunavanja.**
- Ovaj model podrazumijeva da se instrukcije izvršavaju jedna za drugom sekvencijalno, bez mogućnosti izvršavanja bilo kojih konkurentnih operacija. RAM model uključuje instrukcije koje se mogu naći i kod stvarnih računara: aritmetičke (zbrajanje, oduzimanje, množenje, dijeljenje, ostatak, itd.), pomjeranje podataka (load, store, copy, itd.) i kontrolu toka programa (uvjetna i bezuvjetna grananja, pozivi podrutina, itd.).
- **Generički RAM model** ne uključuje tzv. memoriju hijerarhiju, a to znači da analizom nisu obuhvaćeni model cash memorije ili model virtualne memorije, iako su navedene memorije prisutne u stvarnim računarima.
- Uključivanje memorije hijerarhije bi rezultiralo puno kompleksnijim modelom
- Ovakav pristup zapravo omogućuje analizu algoritama, koja je nezavisna od konkretne računarske platforme na kojoj će se izvoditi dotični algoritam.
- Analiza složenosti algoritma s obzirom na skup instrukcija, matematičku kompleksnost, identifikaciju članova u formulama, veličinu ulaznih varijabli je jako kompleksna, pa treba **OSMISLITI EFIKASAN NAČIN ZA MJERENJE SLOŽENOSTI**
- U analizi se koristi oznaka $T(n)$ za veličinu ulaza
- Ako se radi o sortiranju niza veličine n složenost reda ovog algoritma je n^2

Primjer analize:

- Oblik $T(n) = an^2 + bn + c$
- a, b i c su konstante

- $T(n)$ broj elementarnih operacija, koje algoritam treba da izvede za veličinu ulaza n
- Za veliki broj ulaza n dominantan je prvi član prilikom procjene performansi nekog algoritma
- U gornjem slučaju radi se o složenosti reda n^2 , uz zanemarenje konstante a , nižeg stepena n (zbog velikog n)
- U Big-O notaciji, složenost algoritma gornjeg oblika je $O(n^2)$

Analiza algoritma maksimalnog elementa – zavisnost o ulaznim podacima

```
def find_max(data):
    #Maksimalni element u Python listi
    biggest = data[0] # inicijalna vrijednost
    for val in data: # Za svaku vrijednost
        if val > biggest # Ako je vrijednost veća od maksimalne
            biggest = val # neka to postane maksimalna vrijednost
    return biggest # Kada se petlja završi vrati maksimalnu vrijednost
```

Analiza algoritma maksimalnog elementa – zavisnost o ulaznim podacima

```
MAX_ELEMENT (K)
1. max ⇐ K[0]
2. for i ⇐ 1 to n – 1 do
3.     if (K[i] > max) then
4.         max ⇐ K[i]
5.     end_if
6. end_for
7. return max
```

- **Ako se posmatra samo operacija dodjele maksimalne vrijednosti (1 i 4 linija):**
- na početku procedure prije for petlje dodijelit će se samo jednom ako su ulazni podaci u opadajućem poretku,
- ako je niz sortiran u rastućem poretku dodjela će se izvršiti jednom i $n-1$ puta unutar for petlje,
- ako je niz sortiran na način da je najveći element na prvom mjestu, a ostali nasumično dodjela će se izvršiti jednom, a ako je najveći element na drugom mjestu dva puta
- Isto tako ako se radi o nizu $K[0...8]=\{20, 3, 4, 5, 6, 7, 23, 19, 15\}$ broj dodjela je 2, iako se najveći element nalazi na sedmom mjestu $K[6]$.
- Prethodnom analizom može se zaključiti da će broj operacija dodjele biti između 1 i n , odnosno da će izvođenje algoritma zavisiti ulaznim podacima koje algoritam treba da obradi
- **Ograničenja:**
 - Dio koji se ne vidi iz PSEUDOKODA
 - zanemarene su operacije dodjele u for petlji varijable i
 - varijabla i se mijenja u rasponu od 1 do n
 - zanemarene su operacije usporedbe u for petlji
 - utvrđivanja da li je varijabla i veća od $n-1$
 - Dio koji se vidi if kontrola

- **ZAKLJUČAK:** broj operacija koje smo izostavili zapravo linearo raste sa n pri izvođenju algoritamskog procesa, te da razmještaj podataka u ulaznom nizu K nema utjecaja na njihov broj, pa je to još jedno opravdanje zašto te operacije nisu bile u fokusu prethodnog razmatranja.

Najbolji, najgori i prosječni slučaj

- Treba procijeniti koje operacije će biti korištene za usporedbu efikasnosti algoritma
- Operacije usporedbi (jednako, manje, veće, manje ili jednako i sl.)
- Operacije premještanja ili pomjeranja elemenata
- Aritmetičke operacije (aditivne (zbrajanje, oduzimanje, inkrementiranje, dekrementiranje) i množenje (množenje, dijeljenje moduo)) – interesantan slučaj je slučaj cjelobrojnog množenja ili dijeljenja sa potencijom broja dva (primjer 10-100) – svodi se na operaciju pomaka
- Operacije dodjele vrijednosti

Analiza najboljeg slučaja

- najbolji slučaj kod sekvencijalnog algoritma pretraživanja je situacija kada se vrijednost koja se traži nalazi na prvoj poziciji u nizu. U tom slučaju se izvršava samo jedna usporedba, pa je za izvođenje algoritma potrebno neko konstantno vrijeme, bez obzira na dužinu niza koji se pretražuje.

Analiza najgoreg slučaja

- U ovom slučaju očigledno će se dobiti garantirana efikasnost algoritma odnosno algoritam će se izvršiti sigurno u svakom slučaju
- U ovom slučaju se identificiraju ulazne vrijednosti za koje je trajanje algoritamskog procesa najduže (ako vrijednost nije prisutna u nizu)
- Ako se ograničimo za uspješna pretraživanja onda je to slučaj kada imamo traženi element kao zadnji u nizu

Klase ulaza i prosječni slučaj

- Ako se pretražuje niz sa porastom n raste broj mogućih ulaza. Npr za 10 ulaza postoji $10! = 3\ 628\ 800$ mogućnosti na koji način se mogu razmjestiti elementi u nizu
- Ako prepostavimo da je traženi element na prvoj poziciji, onda možemo reći da postoji 362 880 različitih ulaza. Za sve te ulaze će biti potrebna samo jedna operacija usporedbe da bi se pronašao traženi element, pa sve dotične ulaze možemo tretirati kao jednu klasu ulaza. Isto tako, postoji 362 880 različitih ulaza kod kojih je najveći element smješten na drugoj poziciji, itd.

Algoritam za sekvencionalno pretraživanje

```
SEKV (K, k)
1. i ← 0
2. while (i <= n - 1) do
3.   if (K[i] = k) then
4.     return i
5.   else
6.     i ← i + 1
7. end_if
8. end_while
9. return -1
```

- Ako se traženi ključ nalazi u nizu postoji na poziciji na kojim može da bude
- Ako se nalazi na prvoj poziciji izvršit će se samo jedan operacija usporedbe (3)
- Ako se nalazi na drugoj poziciji onda treba da se izvrše dvije operacije i vrijedi općenito ako se element nalazi na i -toj poziciji izvršit će se i operacija usporedbi
- Algoritam možemo grupisati u n klasa, svi kod kojih se traženi element nalazi na prvoj poziciji spadaju u prvu klasu, ...
- S obzirom na prethodno vrijedi uz podjednaku vjerovatnost pojavljivanja klasa na ulazu da je prosječan broj usporedbi jednak:

$$T(n) = \frac{1}{n} \sum_{i=1}^n i = \frac{1+2+\dots+n}{n} = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Ako se sada prepostavi da u nizu ne postoji element koji se traži tada imamo $n+1$ klasu ulaza i tada s obzirom na prethodno vrijedi

$$T(n) = \frac{1}{n+1} \sum_{i=1}^n i + n = \frac{1+2+\dots+n+n+1}{n+1} = \frac{1}{n+1} \frac{n(n+1)}{2} + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

- S obzirom na prethodno očigledno je da se za veliko n operacija usporedbe povećava za $\frac{1}{2}$ što je beznačajno

Analiza u slučaju da klase nemaju istu vjerovatnoću pojavljivanja

- Ako je distribucija vjerovatnosti:
 - vjerovatnost da je traženi broj smješten na prvoj poziciji je $1/2$, $p(u_1)=1/2$
 - vjerovatnost da je traženi broj smješten na drugoj poziciji je $1/4$, $p(u_2)=1/4$
 - vjerovatnost da se najveći broj nalazi na nekoj od preostalih pozicija je jednaka i iznosi

$$p(u_i) = \frac{1 - \frac{1}{2} - \frac{1}{4}}{n-2} = \frac{1}{4(n-2)} \quad i = 3, 4, \dots, n$$

Analiza slučaja – vrijedi za $T(n)$ kada nemamo istu raspodjelu vjerovatnoća pojavljivanja

$$T(n) = \frac{1}{n} \sum_{i=1}^n p(u_i) \cdot t(u_i) = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \sum_{i=3}^n \frac{1}{4(n-2)} \cdot i$$

$$T(n) = 1 + \frac{\frac{n(n+1)}{2} - 3}{4(n-2)} = 1 + \frac{n(n+1) - 6}{8(n-2)} = 1 + \frac{n+3}{8}$$

- za velike vrijednosti n, prosječan broj operacija poređenja je približno $n/8$, pa ako se prisjetimo da je kod uniformne $n/2$ približno u ovom slučaju je prosječan broj operacija manji i algoritam je brži 4 puta

Poopćenje gornjeg primjera

- Ako je $T(n)$ funkcija kojom se izražava broj operacija u zavisnosti o veličini ulaza n, onda se, uz određene pretpostavke, vremenska složenost za prosječni slučaj UZ JEDNAKE VJEROVATNOSTI može se dobiti na sljedeći način:
- Utvrditi broj različitih klasa (m) u koje se mogu svrstati svi mogući ulazi (n) za neki algoritam
- Za svaku grupu treba pronaći broj operacija koje se izvršavaju u algoritmu $t(ui)$
- Izvesti sumu svih operacija i podijeliti sa brojem grupa odnosno pronaći:

$$T(n) = \frac{1}{m} \sum_{i=1}^m t(u_i)$$

- Ako je $T(n)$ funkcija kojom se izražava broj operacija u zavisnosti o veličini ulaza n, onda se, uz određene pretpostavke, vremenska složenost za prosječni slučaj UZ NEJEDNAKE VJEROVATNOSTI može se dobiti na sljedeći način:
- Utvrditi broj različitih klasa (m) u koje se mogu svrstati svi mogući ulazi (n) za neki algoritam
- Za svaku grupu treba pronaći broj operacija koje se izvršavaju u algoritmu $t(ui)$
- Izvesti sumu svih operacija umnoženu sa vjerovatnost pojavljivanja klase $p ui$:

$$T(n) = \sum_{i=1}^m p(u_i) \cdot t(u_i) \text{ gdje je } \sum_{i=1}^m p(u_i) = 1$$

Asimptotska analiza i klasifikacija stepena rasta

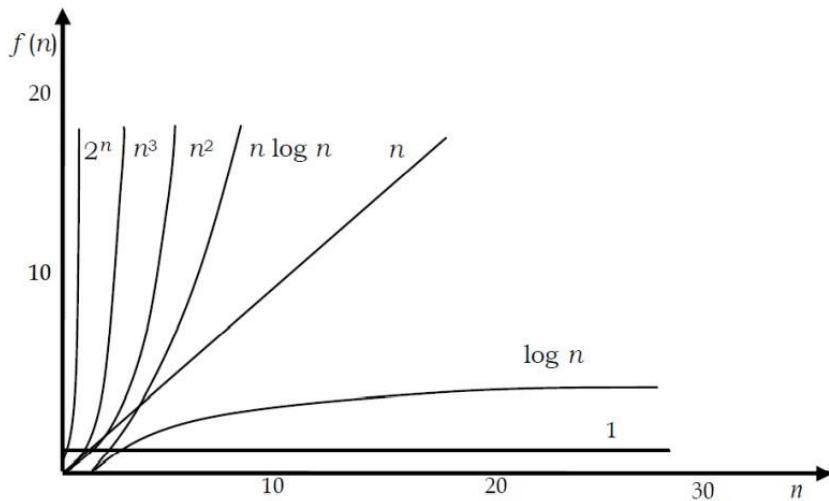
- Zanimanje za stepen rasta sa porastom veličine ulaza. Ne za broj operacija koje se izvršavaju nego interesovanje za klase stepena porasta
- Interesovanja šta se događa sa algoritmom za jako velike ulaze
- Suština je ako smo analizom utvrdili da je vremenska složenost kombinacija dvije ili više klasa složenosti, onda zbog velikog stupnja rasta pojedinih klasa, te klase počinju dominirati nad ostalim funkcijama koje imaju sporiji rast

Primjer:

$$T(n) = n^3 - 5n$$

- Nakon nekog vremena i neke vrijednosti prvi član počinje dominirati nad drugim tako da ovaj algoritam zapravo spada u klasu algoritama koji imaju stepen n^3 .

Primjer nekih klasa algoritma:



NOTACIJA BIG-O

- Notacija za klasifikaciju algoritma koja se odnosi na stepen rasta vremenske složenosti
- Za funkciju $f(n)$ kažemo da je $O(g(n))$, ako postoje pozitivne konstante c i n_0 tako da je ispunjeno:

$$f(n) \leq cg(n), \quad \forall n \geq n_0$$

- Ovim se definira da je $cg(n)$ veća od $f(n)$ za sve vrijednosti od $n \geq n_0$

Primjer:

- Ako se pretpostavi da funkcija ima slijedeći oblik:

$$f(n) = 3n^2 + 5n + 1$$

- Za $g(n)$ uzimamo najveći stepen $g(n) = n^2$ s obzirom na prethodni slajd tvrdimo da vrijedi nejednakost

$$3n^2 + 5n + 1 \leq cn^2$$

$$3 + \frac{5}{n} + \frac{1}{n^2} \leq c$$

- Da bi se zadovoljila nejednakost:

$$3 + \frac{5}{n} + \frac{1}{n^2} \leq c$$

- Treba pronaći n i c – TABELA

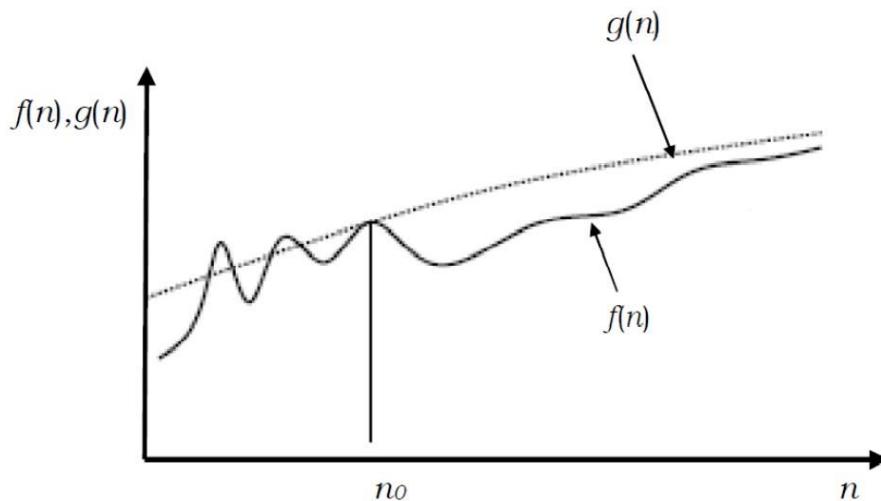
n_0	1	2	3	4	\rightarrow	∞
c	≥ 9	$\geq 23/4$	$\geq 43/9$	$\geq 69/16$	\rightarrow	3

- Očigledno za $n \rightarrow \infty$ beskonačno $c \rightarrow 3$
- **NOTACIJA BIG-O – OPĆENITO SE MOŽE NAPISATI**

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c \quad (*)$$

- Ustvari sa big-O notacijom se definiše gornja granica složenosti algoritma, jer je funkcija $f(n)$ s obzirom na (*) asimptotski ograničena sa $g(n)$
- Može se sa sigurnošću tvrditi da $f(n)$ nema veći red rasta nego $g(n)$

NOTACIJA BIG-O



- Napomenimo da može postojati beskonačno puno funkcija $g(n)$ za datu funkciju $f(n)$. Na primjer, za funkciju $f(n)$, pored izbora za $g(n) = n^2$, u skladu s definicijom mogle bi se izabrati i sljedeće funkcije $g(n) = n^3, g(n) = n^4, g(n) = n^5$, itd.
- Da bi se izbjegle različite nejasnoće, za $g(n)$ se izabire funkcija s najmanjim rastom.

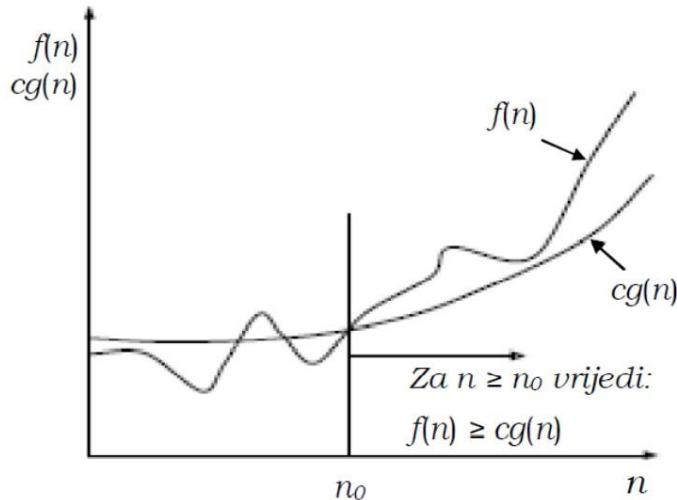
SVOJSTVA NOTACIJE BIG-O

- Ako je $f(n)$ reda $O(g(n))$ i ako je $h(n)$ reda $O(g(n))$, onda je i $f(n)+h(n)$ također reda $O(g(n))$.
- Funkcija reda $a n^k$ je reda $O(n^k)$
- Funkcija $a n^k$ je reda $O(n^{k+j})$ za sve pozitivne vrijednosti j
- Ako je $f(n)$ reda $O(g(n))$ i ako je $g(n)$ reda $O(h(n))$, onda je u slučaj $f(n) \leq g(n)$ i $f(n)$ reda $O(h(n))$.
- Ako je $f(n) = cg(n)$ onda je $f(n)$ reda $O(g(n))$
- **LOGARITMASKA FUNKCIJA IMA ISTI STEPEN RASTA BEZ OBZIRA NA BAZU ALGORITMA**
- Funkcija $f(n) = \log_a n$ je reda $O(\log_b n)$ za bilo koji a i b naravno $b > 1$

- Izražavanjem vremenske složenosti dobija se uvid u ponašanje algoritma prije svega za velike vrijednosti n
- Ponekad je podesno koristiti i donju granicu složenosti algoritma što se označava sa big- Ω
- Za funkciju $f(n)$ kažemo da je $\Omega(g(n))$, ako postoje pozitivne konstante c i n_0 tako da je ispunjeno:

$$f(n) \geq cg(n), \quad \forall n \geq n_0$$

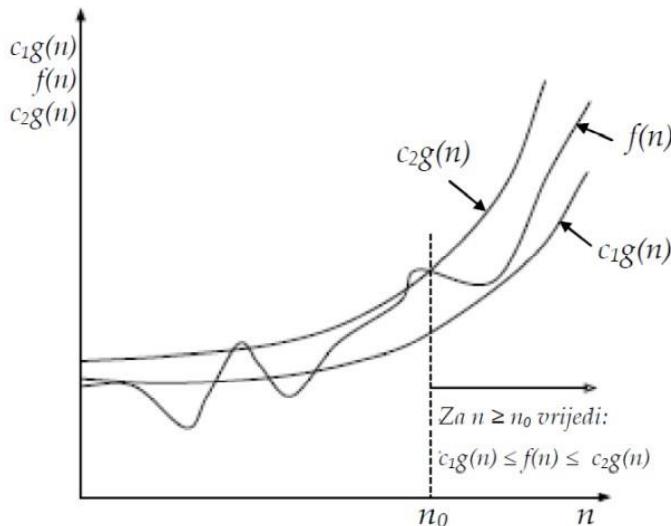
NOTACIJA BIG-O



- Ponekad je podesno koristiti i gornju i donju granicu složenosti algoritma što se označava sa big- Θ
- Na ovaj način se asymptotski određuju granice performansi i sa gornje i sa donje strane
- Za funkciju $f(n)$ kažemo da je $\Theta(g(n))$, ako postoje pozitivne konstante c_1, c_2 i n_0 tako da je ispunjeno:

$$c_1g(n) \leq f(n) \leq c_2g(n), \quad \forall n \geq n_0$$

- Može se zaključiti da je $f(n) = \Theta(g(n))$ ako je ujedno, $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$



NAJČEŠĆE KLASE STEPENA PORASTA

Notacija	Naziv algoritma	Opis
$O(1)$	Konstantni algoritmi	Kod ovih algoritama trajanje algoritamskog procesa je konstantno i ne zavisi od ulaznih podataka. Primjeri algoritma ove klase su: umetanje novog čvora na početak povezane liste, zamjena dva elementa u nizu, itd.
$O(\log n)$	Logaritamski algoritmi	Kod ovih algoritama problem se obično rješava redukcijom problema polovljenjem, sve dok se ne dođe do problema jedinične dužine. Primjer logaritamskog algoritma je binarno pretraživanje sortiranog niza
Notacija	Naziv algoritma	Opis
$O(n)$	Linearni algoritmi	Ovi algoritmi su karakteristični za probleme kod kojih se rješenje dobija ponavljanjem naredbi u petljama, gdje je broj tih ponavljanja proporcionalan veličini ulaza. Primjer algoritma koji ima linearu vremensku složenost je sekvenčno pretraživanje nesortiranog niza.
$O(n \log n)$	Linearno logaritamski algoritmi	Ovi algoritmi su karakteristični za probleme koji se rješavaju polovljenjem, pri čemu se ipak obrađuju svi ulazni podaci. Neki efkasniji algoritmi sortiranja imaju ovu vremensku složenost
Notacija	Naziv algoritma	Opis
$O(n^2)$	Kvadratni algoritmi	Ovi algoritmi najčešće imaju dvije ugnježđene petlje, pri čemu je broj izvršavanja svake od petlji proporcionalan sa n . Jednostavniji algoritmi sortiranja imaju ovu vremensku složenost.
$O(n^3)$	Kubni algoritmi	Ovi algoritmi najčešće imaju tri ugnježđene petlje, pri čemu je broj izvršavanja svake od petlji proporcionalan sa n .
Notacija	Naziv algoritma	Opis
$O(n^k)$	Stepeni algoritmi	Ovi algoritmi tipično imaju k ugnježđenih petlji, pri čemu je broj izvršavanja svake od petlji proporcionalan sa n . Kod ovih algoritama dolazi do vrlo brzog porasta trajanja algoritamskog procesa za velike veličine ulaznih podataka n .
$O(k^n)$	Eksponencijalni algoritmi	Ovi algoritmi su karakteristični za probleme koji se rješavaju iscrpnim pretraživanjem svih mogućnosti. Takve algoritme nazivamo algoritmi „grube sile“. Trajanje algoritamskog procesa kod ovih algoritama raste izuzetno brzo sa porastom veličine ulaznih podataka n , pa često imaju samo teorijsku važnost za probleme velikih dimenzija.

- broj operacija i potrebno vrijeme za izvršavanje tih operacija, uz pretpostavku da računar na kojem se izvodi algoritam može obaviti 10^7 operacija u sekundi, odnosno deset operacija u mikrosekundi (10^{-6} o/mikrosekundi)

Klasa veličina ulaza n	$O(1)$	$O(\log n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
10	1	3.32	33.2	10^2	10^3	1024
	0.1 μsec	0.3 μsec	3.3 μsec	10 μsec	100 μsec	1 msec
10^2	1	6.64	664	10^4	10^6	10^{30}
	0.1 μsec	0.7 μsec	66.4 μsec	1 msec	0.1 sec	$3.17 \cdot 10^{15}$ god.
10^3	1	9.97	$9.97 \cdot 10^3$	10^6	10^9	10^{301}
	0.1 μsec	1 μsec	1 msec	0.1 sec	1.67 min	$3.17 \cdot 10^{286}$ god.
10^4	1	13.3	$133 \cdot 10^3$	10^8	10^{12}	10^{3010}
	0.1 μsec	1.3 μsec	13.3 msec	0.17 min	1.16 dana	
10^5	1	16.6	$1.66 \cdot 10^6$	10^{10}	10^{15}	10^{30103}
	0.1 μsec	1.7 μsec	0.16 sec	16.7 min	3.17 god.	
10^6	1	19.93	$19.93 \cdot 10^6$	10^{12}	10^{18}	10^{301030}
	0.1 μsec	2 μsec	2 sec	1.16 dana	3170.9 god.	

- Iz tabele možemo vidjeti da npr. algoritmi složenosti $O(n^3)$ za veličinu ulaza 10^5 praktično imaju samo teoretski značaj, jer je potrebno više od 3 godine da bi se algoritam te složenosti izveo.
- Ili, na primjer, za veličinu ulaza 10^6 , potrebno je više od 3170 godina.
- Još kompleksnija situacija je sa eksponencijalnim algoritmima, koji imaju složenost npr. $O(2^n)$. Iz tabele 2.5 možemo vidjeti da je već za veličinu ulaza 10^3 potrebno izvršiti 10^{301} operacija, dok bi za izvođenje tolikog broja operacija bilo potrebno $3 \cdot 17 \cdot 10^{286}$ godina.
- Dakako, još kompleksnija situacija je za veličine ulaza 10^5 ili 10^6 .
- Da bi dobili dojam o veličini nekih vremenskih intervala prikazanih u tabeli, spomenimo npr. da je starost planete Zemlje oko $4.5 \cdot 10^9$ godina. Ili npr. procjenjuje se da je ukupan broj atoma u svemiru između 10^{78} i 10^{81} .

(Predavanje III)

Nizovi i povezane liste

Nizovi

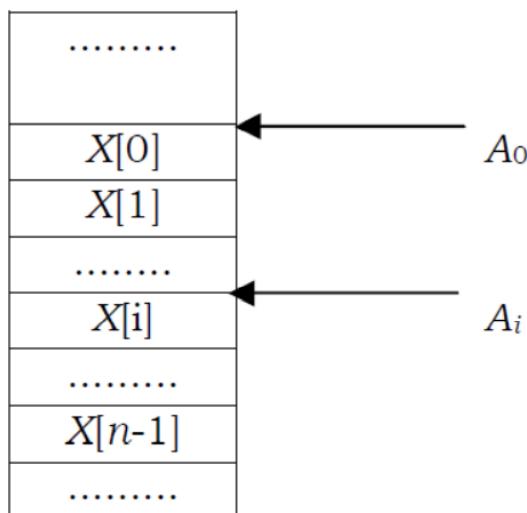
- Niz je linearno uređena struktura podataka koja se sastoji od konačnog broja homogenih elemenata.
- Pod pojmom homogenih elemenata podrazumijeva se da su svi elementi u nizu istog tipa
- Pod pojmom uređenost podrazumijeva se da se zna tačan raspored svih elemenata

Jednodimenzionalni nizovi

- Indeks pozicije služi za određivanje mesta u nizu. Indeks pozicije je iz opsega $[l \dots u]$ l – donja granica (obično 0 ili 1), u – gornja granica
- Ukupan broj elemenata niza je $u-l+1$. Nizu se pristupa na način da se navede ime i indeks u nizu
- Npr. četvrtom elementu pristupamo sa $A[3]$, ako je donja granica $l=0$
- Za prikaz nizova u memoriji se najčešće koristi sekvencionalni prikaz, u kojem se nizovi u memoriji redaju u skup susjednih memorijskih lokacija
- Logički koncept niza se pokapa sa fizičkim konceptom u memoriji
- Ovakav pristup uzrokuje problem pri umetanju ili brisanju elementa u nizu
- Umetanje – sa određene pozicije moraju se pomjeriti svi elementi u nizu
- Brisanje – elementi sa određene pozicije treba da se pomjere za jednu poziciju naniže

ALOKACIJA JEDNODIMENZIONALNIH NIZOVA

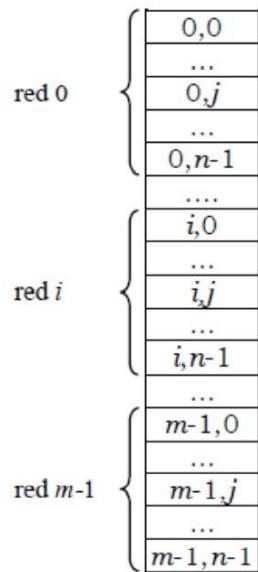
- Za smještanje jednog elementa u nizu potrebno je s memorijskih riječi, te ako imamo n elemenata tada nam treba n^* s memorijskih riječi
- Za pristup elementu koristi se adresna funkcija,, odnosno proračun adrese A_i gdje se nalazi traženi element na kojoj je pohranjen $X[i]$ $A_i = A_0 + i \cdot s$ $i = 0, 1, \dots, n - 1$
- Poopćeno ako se koristi l kao donja granica $A_i = A_l + i - l \cdot s$ $i = l, l + 1, \dots, l + n - 1$



ALOKACIJA DVODIMENZIONALNIH NIZOVA

- Dvodimenzionalni logički niz treba prevesti u jednodimenzionalni niz na fizičkim lokacijama, odnosno provesti linearizaciju
- Niz $X[l_1 \dots u_1, l_2 \dots u_2]$ treba prevesti na način da se dobije jednostavno izračunavanje npr. adrese $A_{i,j}$ za element $X[i, j]$
- Alokacija po redovima
- Alokacija po kolonama

	0	\dots	j	\dots	$n-1$
0	$0,0$	\dots	$0,j$	\dots	$0,n-1$
\dots	\dots	\dots	\dots	\dots	\dots
i	$i,0$		i,j	\dots	$i,n-1$
\dots	\dots	\dots	\dots	\dots	\dots
$m-1$	$m-1,0$		$m-1,j$	\dots	$m-1,n-1$



ALOKACIJA PO REDOVIMA

- Provodi se na način da se prvo smješta prvi red, zatim drugi red sve do reda $m-1$
- Ako se koristi raspon $i=0, \dots, m-1$ i $j=0, \dots, n-1$, adresna funkcija za element $X[i, j]$ ima oblik

$$A_{i,j} = A_{0,0} + i \cdot n + j \cdot s$$

$$i = 0, 1, \dots, m-1 \text{ i } j = 0, 1, \dots, n-1$$
gdje je $A_{0,0}$ adresa prvog elementa, m broj redova, n broj kolona i s memorijска veličina jednog elementa

ALOKACIJA PO REDOVIMA - POPĆENO

- Provodi se na način da se prvo smješta prvi red, zatim drugi red sve do reda $m-1$
- Ako se koristi raspon $i=l_1, \dots, u_1$ i $j=l_2, \dots, u_2$, adresna funkcija za element $X[i, j]$ ima oblik

$$A_{i,j} = A_{l_1,l_2} + i - l_1 \cdot n + j - l_2 \cdot s$$

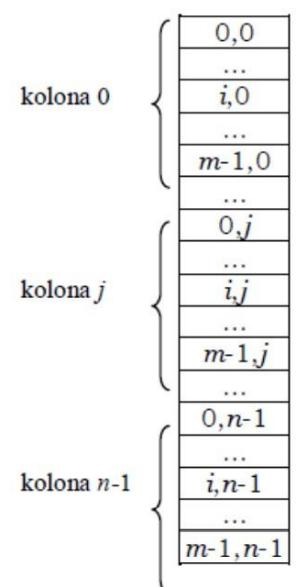
$$l_1 \leq i \leq u_1 \text{ i } l_2 \leq j \leq u_2$$
gdje je A_{l_1,l_2} adresa prvog elementa, m broj redova u_1-l_1+1 , n broj kolona u_2-l_2+1 i s memorijска veličina jednog elementa

ALOKACIJA PO KOLONAMA

- Provodi se na način da se prvo smjesti prva kolona, zatim druga kolona sve do kolone $n-1$
- Ako se koristi raspon $i=0, \dots, m-1$ i $j=0, \dots, n-1$, adresna funkcija za element $X[i, j]$ ima oblik

$$A_{i,j} = A_{0,0} + j \cdot m + i \cdot s$$

$$i = 0, 1, \dots, m-1 \text{ i } j = 0, 1, \dots, n-1$$
gdje je $A_{0,0}$ adresa prvog elementa, m broj redova, n broj kolona i s memorijска veličina jednog elementa



ALOKACIJA PO KOLONAMA - POOPĆENO

- Provodi se na način da se prvo smjesti prva kolona, zatim druga kolona sve do kolone n-1
- Ako se koristi raspon $i=i_1, \dots, i_u$ i $j=j_1, \dots, j_u$, adresna funkcija za element $X[i, j]$ ima oblik

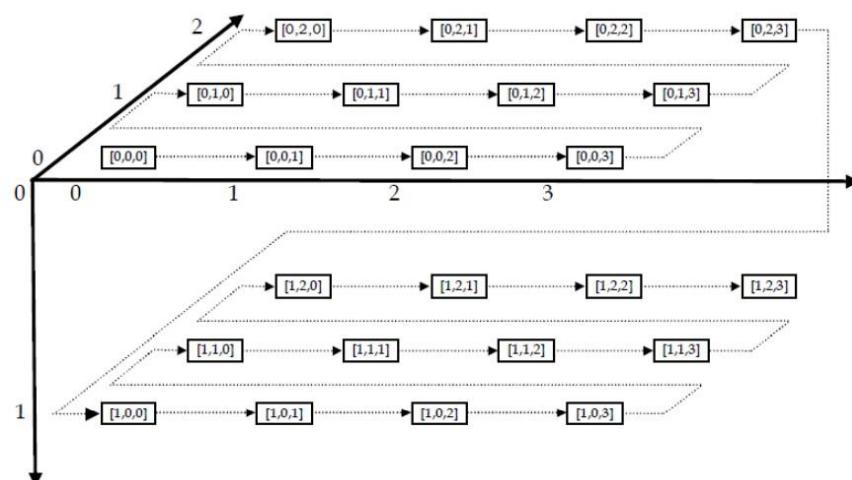
$$A_{i,j} = A_{i_1, j_1} + j - j_1 \cdot m + i - i_1 \cdot s \quad i_1 \leq i \leq i_u \text{ i } j_1 \leq j \leq j_u$$

gdje je A_{i_1, j_1} adresa prvog elementa, m broj redova u_1-i_1+1 , n broj kolona u_2-j_1+1 i s memorijska veličina jednog elementa

ALOKACIJA VIŠEDIMENZIONALNIH NIZOVA

- I u ovom slučaju je potrebno obaviti linearizaciju
- N-dimenzionalni niz $X[0..m_1, 0..m_2, \dots, 0..m_n]$
- Dobro bi bilo da prvo varira indeks najviše dimenzije niza, zatim manji, ...
- Ovo isto se moglo primijeniti i kod razmatranja dvodimenzionalnog niza, variramo prvo red ili kolonu, odnosno višu dimenziju niza
- Primjer niz $X[0..1, 0..2, 0..3]$
- Prva dimenzija ima veličinu dva $m_1=1-0+1=2$
- Druga dimenzija ima veličinu tri $m_2=2-0+1=3$
- Treća dimenzija ima veličinu četiri $m_3=3-0+1=4$

$X[0, \dots, 0, 0]$
$X[0, \dots, 0, 1]$
....
$X[0, \dots, 0, m_n]$
$X[0, \dots, 1, 0]$
$X[0, \dots, 1, 1]$
....
$X[0, \dots, 1, m_n]$
$X[0, \dots, 2, 0]$
$X[0, \dots, 2, 1]$
....
.....
$X[m_1, m_2, \dots, m_n]$



Poredak u memoriji

$A+0s$	$X[0,0,0]$	$A+8s$	$X[0,2,0]$	$A+16s$	$X[1,1,0]$
$A+1s$	$X[0,0,1]$	$A+9s$	$X[0,2,1]$	$A+17s$	$X[1,1,1]$
$A+2s$	$X[0,0,2]$	$A+10s$	$X[0,2,2]$	$A+18s$	$X[1,1,2]$
$A+3s$	$X[0,0,3]$	$A+11s$	$X[0,2,3]$	$A+19s$	$X[1,1,3]$
$A+4s$	$X[0,1,0]$	$A+12s$	$X[1,0,0]$	$A+20s$	$X[1,2,0]$
$A+5s$	$X[0,1,1]$	$A+13s$	$X[1,0,1]$	$A+21s$	$X[1,2,1]$
$A+6s$	$X[0,1,2]$	$A+14s$	$X[1,0,2]$	$A+22s$	$X[1,2,2]$
$A+7s$	$X[0,1,3]$	$A+15s$	$X[1,0,3]$	$A+23s$	$X[1,2,3]$

- Ako treba da se dođe do element X[i1, i2, i3] npr. i1=1, i2=2 i i3=2, prvo treba naći podudaranje prvog indeksa, odnosno preći preko svih elemenata na nultoj ravni (12) m2*m3=12 u ovom slučaju u ravni 0 i tada dolazimo do elementa X[1, 0, 0]
- Da bi došli do i2=2, treba preći dvije grupe od četiri elementa i tada se nalazimo na X[1, 2, 0]
- Zatim preći preko još dva elementa koja su određena indeksom i3
- Nakon svih ovih prelazaka dolazi se do X[1, 2, 2]
- Da bi se izvela adresna funkcija u cilju lociranja elementa X[i1, i2, i3] potrebno je:

$$A_{i_1, i_2, i_3} = A_{0,0,0} + i_1 \cdot m_2 \cdot m_3 \cdot s + i_2 \cdot m_3 + i_3$$
- Za gornji primjer

$$A_{1,2,2} = A_{0,0,0} + 1 \cdot 3 \cdot 4 \cdot s + 2 \cdot 4 + 3$$

ALOKACIJA VIŠEDIMENZIONALNIH NIZOVA - POOPĆENJE

- Višedimenzionalni nizovi n>3 – teško ih je zamisliti ali vrijedi ista logika
- Niz je X[i1, i2, ..., in] ($0 \leq i_j \leq m_j$ gdje je $j=1, 2, \dots, n$)
- Prvo treba naći podudaranja prvog elementa i1
- Za ovo treba prijeći preko i1 elemenata pri čemu svaka grupa ima $(m_2-1)(m_3-1)\dots(m_n-1)$ elemenata
- Zatim preko i2 grupa kojih ima $(m_3-1)(m_4-1)\dots(m_n-1)$... • Konačno adresna funkcija ima oblik

$$A_{i_1, \dots, i_n}$$

$$= A_{0, \dots, 0} + i_1 \cdot (m_2-1) \cdot (m_3-1) \dots (m_n-1) \cdot s + i_2 \cdot (m_3-1) \cdot (m_4-1) \dots (m_n-1) + \dots + i_{n-1} \cdot m_n - 1) + i_n$$

IMPLEMENTACIJA LISTE POMOĆU NIZOVA

- Lista se može definirati kao uređena sekvenca podatkovnih elemenata (čvorova)
- Pod uređenom sekvencom zapravo podrazumijevamo da svaki element ima svoju poziciju. (Isti tip)
- Operacije koje su definirane nad listom, kao apstraktnim tipom podataka, ne zavise o tome kojeg su tipa elementi.
- Na primjer, lista, kao apstraktan tip podataka, može sadržavati cjelobrojne podatke, znakovne podatke, zapise o finansijskim transakcijama ili neke druge jednostavnije ili kompleksnije tipove podataka
- Prazna lista - ne sadrži niti jedan element
- Dužina liste - označava broj elemenata u listi
- Čelo liste - označava početak liste
- Rep liste - označava kraj liste
- $(a_0, a_1, \dots, a_{n-1})$ gdje je $i=0, 1, \dots, n-1$
- Lista ima kapacitet[L] – označava maksimalni broj elemenata koje lista može sadržavati
- velicina[L] – označava stvarni broj elemenata u listi
- Lista se može povećavati (umetanjem elementa u listu) ili smanjivati (brisanjem elemenata iz liste)
- Tekuća lista se dijeli na dvije particije, koje su odvojene zamišljenom pregradom
- Tekuća pozicija u listi se predstavlja atributom tekuci[L]
- Za označavanje trenutne pozicije u konfiguraciji liste koristit će se vertikalna crta koja će prikazivati zamišljenu pregradu između dvije particije, lijeve i desne
- Npr. za 7 elementa 15, 27, 8, 43, 29, 12, 34
- (15, 27 | 8, 43, 29, 12, 34) – dva + pet elementa
- Postoji operacija kojom se na kraj liste direktno dodaje novi element, pri čemu tekuća pozicija ostaje nepromijenjena.
- Elementi liste se mogu čitati, (radi eventualne promjene vrijednosti elementa ili pak izvođenja neke druge operacije)

- Listi možemo pristupati, stvarati je ili je reinicijalizirati.
- Postoje operacije kojima se tekuća pozicija postavlja na početak liste, kraj liste, ili na prethodni

IMPLEMENTACIJA LISTE POMOĆU NIZOVA – OPERACIJE

IDI NA POCETAK (L)

1 tekuci[L] $\leftarrow 0$

IDI NA KRAJ (L)

1 tekuci[L] $\leftarrow \text{velicina}[L]$

IDI NA PRETHODNI (L)

1 if (tekuci[L] $\neq 0$) then

2 tekuci[L] $\leftarrow \text{tekuci}[L] - 1$

3 end_if

IDI NA SLIJEDECI (L)

1 if (tekuci[L] < velicina[L]) then

2 tekuci[L] \downarrow tekuci[L] +1

3 end_if

IDI NA POZICIJU (L, i)

1 if (i < 0 or i>velicina[L]) then

2 ERROR („Indeks nedozvoljenog raspona“)

3 end_if

4 tekuci[L] $\leftarrow i$

PRIMJER

IDI-NA-POCETAK (L) \rightarrow (| 15, 27, 8, 43, 29, 12, 34)

IDI-NA-KRAJ (L) \rightarrow (15, 27, 8, 43, 29, 12, 34 |)

IDI-NA-PRETHODNI (L) \rightarrow (15 | 27, 8, 43, 29, 12, 34)

IDI-NA-SLJEDECI (L) \rightarrow (15, 27, 8 | 43, 29, 12, 34)

IDI-NA-POZICIJU (L,5) \rightarrow (15, 27, 8, 43, 29 | 12, 34)

UMETANJE ELEMENATA U LISTU

- Ako u listi ima još slobodnih lokacija, funkcija pomjera sve elemente od tekuće pozicije do kraja liste za jedno mjesto udesno, a zatim tekućoj poziciji dodjeljuje novi element x.
- Veličina liste se povećava za 1, ako se operacija uspješno obavi vraća **true**, a ako ne vraća **false**
- Ako se nalazi na kraju izvodi se u vremenu O(1)
- Ako se nalazi na početku izvodi se u vremenu O(n) – treba pomjeriti sve elemente udesno
- U prosječnom slučaju za n/2 traje O(n)

```

UMETNI(L, x)
1 if (velicina[L] < kapacitet[L] ) then
2   for i <= velicina[L] downto tekuci[L] +1 do
2     L[i] <= L[i-1]
3   end_for
4   L[tekuci[L]] <= x
5   velicina[L] <= velicina[L] +1
6   return true
7 else
8   return false
9 end_if

```

- Za sljedeću konfiguraciju (15, 27 | 8, 43, 29), pozivanje funkcije umetni(L,20) će promijeniti konfiguraciju liste
- (15, 27 | 20, 8, 43, 29)
- Operacija dodavanja elementa u listu
- Nakon utvrđivanja da u listi ima slobodnih pozicija, na poziciju s indeksom **velicina[L]**, neposredno iza trenutno zadnjeg elementa, upisuje se novi element **x**, zatim se povećava atribut **velicina** za 1, te se vraća vrijednost **true**.
- Ako nema slobodnih pozicija funkcija vraća **false**.
- Operacija dodavanja se izvodi u konstantnom vremenu O(1).

DODAJ(x)

```

1 if (velicina[L] < kapacitet[L] ) then
2   L[tekuci[L]] <= x
3   velicina[L] <= velicina[L] +1
4   return true
5 else
6   return false
7 end_if

```

- Npr. funkcija dodaj(12) početnu konfiguraciju liste (15, 27 | 8, 43, 29), mijenja u (15, 27 | 8, 43, 29, 12), može se zapaziti da je funkcija DODAJ(x) skraćeni oblik funkcije UMETNI(L, x)

Operacija brisanja elementa iz liste

- Ako je indeks tekuće pozicije pozicioniran tako da je desna particija prazna, funkcija signalizira grešku.
- U suprotnom, element na tekućoj poziciji se pamti u privremenom objektu x, te se svi elementi od tekuće pozicije do kraja liste pomjeraju za jedno mjesto uljevo. Na kraju se atribut velicina umanjuje za 1, te funkcija vraća element koji se izbacuje.
- Najbolji slučaj pri izbacivanju je izbacivanje zadnjeg elementa, u kojoj nema pomjeranja elemenata, pa se operacija izvodi u konstantnom vremenu O(1).
- Najgori slučaj se pojavljuje kada se izbacuje prvi element. Tada je potrebno pomjeriti preostalih n-1 elemenata, te zaključujemo da se operacija izvodi u vremenu O(n). U prosječnom slučaju je potrebno pomjeriti (n-1)/2 elemenata, pa se u prosjeku operacija izbacivanja izvodi u linearном vremenu O(n).

IZBACI(L)

```
1 if (tekuci[L] > velicina[L] ) then
2     ERROR („Ništa za izbaciti“)
3 end_if
4 x ← L[tekuci]
5 for i ← tekuci[L] to velicina[L]-1 do
6     L[i] ← L[i+1]
7 end_for
8 velicina[L] ← velicina[L] -1
9 return x
```

- Npr. funkcija IZBACI(L) početnu konfiguraciju liste (15, 27 | 8, 43, 29), mijenja u (15, 27, 43, 29)

(Predavanje IV)

Povezane liste

- Povezane liste su dinamičke strukture podataka kod kojih se prostor za elemente alocira tek kada je to potrebno prilikom operacije umetanja novog elementa u listu
- Kod sekvensijalne implementacije linearne liste, susjedni elementi u logičkom poretku su ujedno i susjedni elementi u fizičkom poretku.
- S druge strane, kod ulančane implementacije, susjedni elementi u logičkom poretku mogu biti pohranjeni bilo gdje u memoriji.
- kod ulančane implementacije linearnih listi, logički poredak elemenata se ne može izvesti iz njihovog fizičkog poretna u memoriji – zbog ovoga se uvodi pojam **čvora (Node)**

Čvor (Node)

- Sastoji se od dva dijela:
- informacionog sadržaja (označen sa info) – ovo može biti bilo kojeg tipa da li elementarnog ili struktturnog ali zbog jednostavnosti implementacije koristiti će se int
- pokazivač na sljedeći čvor u poretku (označen sa sljedeći)



FUNKCIJE:

- GETNODE – alocira memorijski prostor za jedan čvor i vraća adresu tog prostora (pokazivač na čvor).
- Npr. $p \leftarrow \text{GETNODE}();$ - alocira se mem. prostor za čvor, vraća se adresa koja se pridružuje pokazivaču p
- FREENODE (p) – oslobađa se mem. prostor na koji pokazuje pokazivač p
- Povezanoj listi se pristupa preko pokazivača **pocetak**, koji pokazuje na prvi čvor. Ovaj pokazivač se naziva i čelo (**Head**). Ako pokazivač ne pokazuje niti na jedan čvor onda on ima vrijednost NULL (pocetak=NULL)

Jednostruko povezana lista

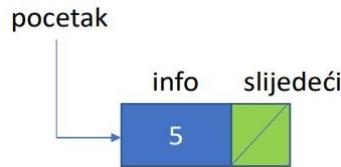
pocetak



ZADATAK:

- Kreirati povezanu listu od tri čvora (3 int vrijednosti 5, 12 i 8)
- Prvi čvor liste se kreira sa sekvencom

1 pocetak[L] \leftarrow GETNODE()
 2 info(pocetak[L]) \leftarrow 5
 3 slijedeci(pocetak[L]) \leftarrow NULL

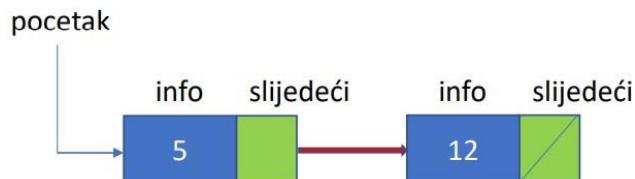


- Naredba 1. alocira prostor za prvi čvor
i pokazivač pocetak pridružuje adresu tog prostora
- Naredba 2. informacionom dijelu dodjeljuje vrijednost 5
- Naredba 3. u pokazivač slijedeci upisuje se NULL

ZADATAK:

- Dodavanje drugog čvora koji sadrži broj 12

1 slijedeci(pocetak[L]) \leftarrow GETNODE()
 2 info(slijedeci(pocetak[L])) \leftarrow 12
 3 slijedeci(slijedeci(pocetak[L])) \leftarrow NULL



- Naredba 1. alocira prostor za drugi čvor i pokazivaču, smještenom u polju slijedeci pridružuje adresu tog prostora
- Naredba 2. informacionom dijelu dodjeljuje 12
- Naredba 3. pokazivač slijedeci dobija vrijednost NULL. NAPOMENA: konstantno koristimo pokazivač pocetak koji pokazuje na prvi čvor

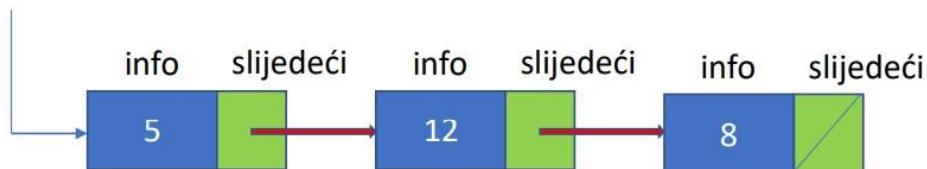
ZADATAK:

- Dodavanje trećeg čvora koji sadrži broj 8

1 slijedeci(slijedeci(pocetak[L])) \leftarrow GETNODE()
 2 info(slijedeci(slijedeci(pocetak[L]))) \leftarrow 8
 3 slijedeci(slijedeci(slijedeci(pocetak[L]))) \leftarrow NULL

- NAPOMENA: konstantno koristimo pokazivač pocetak koji pokazuje na prvi čvor. Osim toga, što je lista duža, potrebno je koristiti sve više pokazivača zapisanih u polju slijedeci, radi pristupa elementima liste.
- Povezane liste 5 info slijedeći 12 info slijedeći pocetak

pocetak



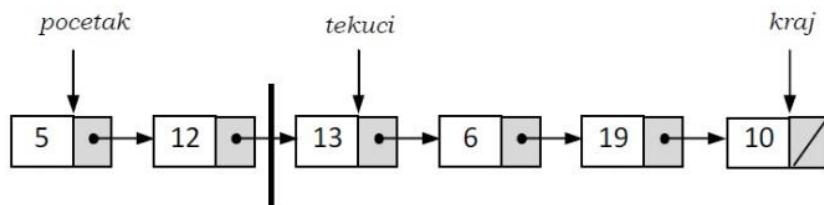
EFIKASNIJE KORIŠTENJE LISTI:

- Uvođenje pokazivača tekuci, kraj, i dužina desne i lijeve particije
- Kraj liste pokazivač kraj (u engleskoj literaturi tail) pokazuje kako ime kaže na kraj liste
- lduzina – sadrži cijelobrojnu vrijednost za lijevu particiju (broj elemenata koji se nalaze u lijevoj particiji)
- dduzina – sadrži cijelobrojnu vrijednost za desnu particiju (broj elemenata koji se nalaze u desnoj particiji)
- Pokazivač tekuci ima funkciju da pokazuje na trenutni čvor
- Problem efikasnosti pri definisanju čvora na koji pokazuje pokazivač tekuci

Slučaj I: Pokazivač tekuci pokazuje na prvi element desne particije

lduzina=2

dduzina=4

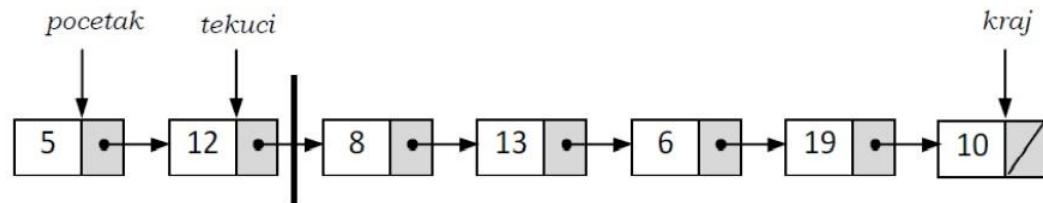


- Umetanje novog čvora koji ima informacionu vrijednost 8 u ovom slučaju se mora pokazivaču slijedeci čvora koji sadrži info 12 promijeniti pokazivač da pokazuje na novi čvor. Dalje referenciranje u ovom slučaju na početak otežava dalje referenciranje od novog čvora.

Slučaj II: Pokazivač tekuci pokazuje na zadnji element lijeve particije i ovom slučaju umetanje elementa na početak desne particije neće biti problem

lduzina=2

dduzina=5



OPERACIJE U RADU SA POVEZANIM LISTAMA:

- pristup čvorovima u listi;
- umetanje novog čvora na tekuću poziciju;
- dodavanje novog čvora na kraj liste;
- brisanje čvora na tekućoj poziciji;
- pretraživanje liste.

PRISTUP ČVOROVIMA U LISTI:

Procedura

IDI-NA-SLIJEDECI(L)

```
1 if (druzina[L] ≠ 0) then
2   if (lruzina[L] = 0) then
3     tekuci[L] ← pocetak[L]
4   else
5     tekuci[L] ← slijedeci(tekuci[L])
6   end_if
7   lruzina[L] ← lruzina[L] + 1
8   druzina[L] ← druzina[L] - 1
9 end_if
```

- Ako je desna particija prazna onda se ne dešava ništa
- U suprotnom pokazivač tekuci se pomjera na početak ako nema ništa sa lijeve strane
- Ili na slijedeći čvor (5) nakon čega se lruzina povećava za 1, a desna dužina smanjuje na 1

Procedura

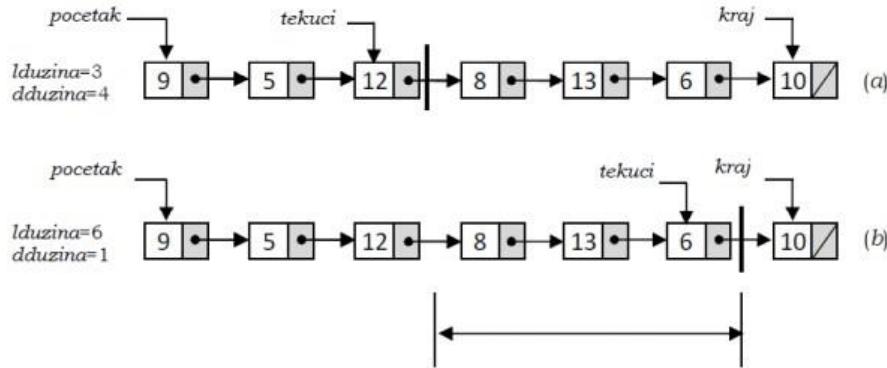
IDI-NA-PRETHODNI(L)

```
1 if (lruzina[L] ≠ 0) then
2   if (lruzina[L] = 1) then
3     tekuci[L] ← NULL
4   else
5     p ← pocetak[L]
6     while slijedeci(p) ≠ tekuci[L] do
7       p ← slijedeci[p]
8     end_while
9     tekuci[L] ← p
10    end_if
11    lruzina[L] ← lruzina[L] - 1
12    druzina[L] ← druzina[L] + 1
13 end_if
```

- Ako je lijeva particija prazna neće se izvršiti slijed operacija
- Ako nije mora se slijediti lanac pokazivača počevši od početka liste (5) pa sve do prethodnika (6-9)
- Na kraju se ažuriraju u linijama 11 i 12 atributi lruzina i druzina
- U drugoj liniji je specijalan slučaj kada lijeva particija sadrži jedan element (linija2) tekuci označava kraj odnosno NULL
- Prethodna procedura IDI-NA-POZICIJU ostvaruje pomjeranje pokazivača tekuci prema željenoj poziciji, tako da se slijedi lanac pokazivača od čela liste.
- Međutim, u nekim situacijama se elementima liste pristupa tako da se slijedeći element kojem se pristupa nalazi na nekoj poziciji iza elementa kojem se prethodno pristupilo. Tada vrijedi da je $i > lruzina[L]$.

Ova procedura će i u takvim slučajevima pristup i-tom elementu ostvariti tako da se lanac pokazivača slijedi od čela liste, bez obzira što bi bilo efikasnije lanac pokazivača slijediti od trenutne vrijednosti pokazivača tekuci. Na primjer, na slici je prikazana konfiguracija liste u kojoj je vrijednost lduzina=3.

- Prepostavimo da je potrebno pristupiti sedmom ($i=6$) elementu, tako da dužina lijeve particije iznosi lduzina=6. Procedura IDI-NA-POZICIJU će trebati šest koraka, bez obzira što se pristup sedmom elementu zapravo mogao ostvariti u tri koraka.



Procedura

IDI-NA-POZICIJU (L, i)

```

1  if (i < dduzina[L] + lduzina[L]) then
2      ERROR („Indeks izvan opsega“)
3  end_if
4  dduzina[L] <= dduzina[L] + lduzina[L] - i
5  lduzina[L] <= i
6  if (i = 0) then
7      tekuci[L] <= NULL
8  else
9      tekuci[L] <= pocetak[L]
10     for i <= 0 to i-1 do
11         tekuci[L] <= sljedeci(tekuci[L])
12     end_for
13 end_if

```

- Prvo se na temelju indeksa i , izračunavaju nove vrijednosti atributa dduzina i lduzina (linije 4 i 5), a zatim se, slijedeći lanac pokazivača od čela liste, pokazivač tekuci pozicionira na zadnji čvor u lijevoj particiji (linije 9-13).
- Procedura u obzir uzima i specijalan slučaj, koji se pojavljuje kada indeks $i=0$ te se pokazivač tekuci tada direktno postavlja na $\text{tekuci}=\text{NULL}$ (linije 6-7).
- Potrebno vrijeme za izvođenje ove operacije je reda $O(n)$.

Procedura – modificirana varijanta prelaza na poziciju

IDI-NA-POZICIJU-M (L, i)

```

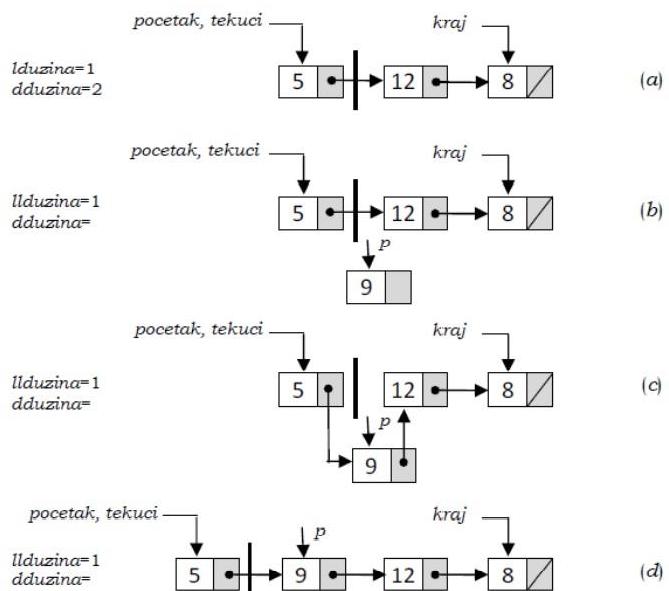
1  if (i < dduzina[L] + lduzina[L]) then
2      ERROR („Indeks izvan opsega“)
3  end_if
4  p := 0;
5  Id := lduzina[L]
6  dduzina[L] := dduzina[L]+lduzina[L]-i
7  lduzina[L] := i
8  if (i = 0) then
9      tekuci[L] := NULL
10   return
11 end_if
12 if (i ≥ Id) then
13     if (Id = 0) then
14         tekuci[L] := pocetak[L]
15     else
16         p := Id-1
17     end_if
18 else
19     tekuci[L] := pocetak[L]
20 end_if
21 for i := p to i-1 do
22     tekuci[L] := sljedeci(tekuci[L])
23 end_for

```

- Modificirana varijanta procedure IDI-NAPOZICIJU, koja koristi pokazivač tekuci i ne vraća se na čelo liste

Umetanje novog čvora x=9

- Nakon kreiranja novog čvora i upisivanja sadržaja 9, lista se povezuje tako da novi čvor pokazuje na čvor 12, koji je prethodno bio prvi element desne particije, dok čvor 5, koji je zadnji element lijeve particije, pokazuje na novi čvor 9.
- Na kraju je predstavljen samo drugačiji grafički prikaz prevezane liste, uz novu ažuriranu vrijednost atributa dduzina=3.



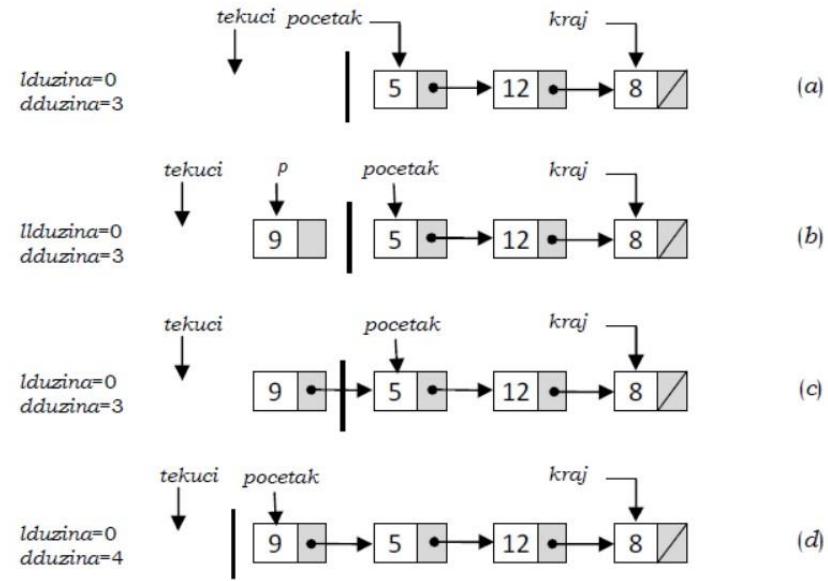
Procedura umetanja:

UMETNI (L, x)

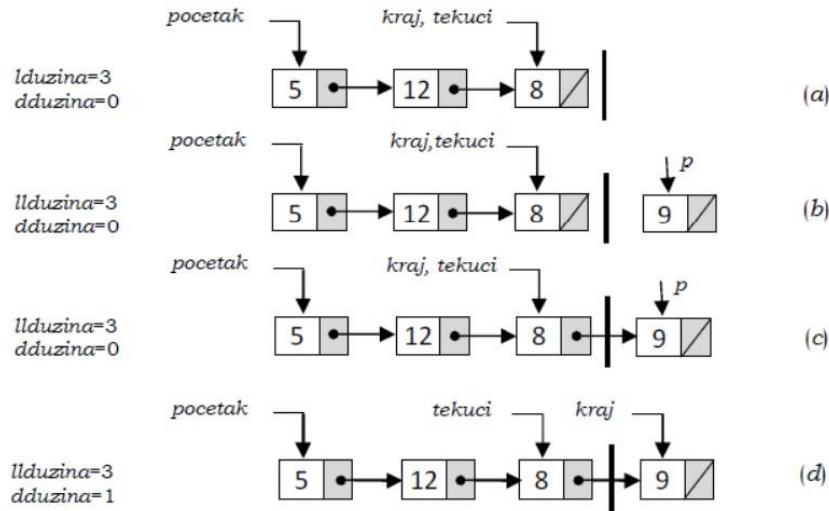
1. $p \leftarrow \text{GETNODE}()$
2. $\text{info } p \leftarrow x$
3. **if** ($\text{l}duzina[L] = 0$) **then**
4. $\text{slijedeci}(p) \leftarrow \text{pocetak}[L]$
5. $\text{pocetak}[L] \leftarrow p$
6. **if** ($\text{l}duzina[L] + dduzina[L] = 0$) **then**
7. $\text{kraj}[L] \leftarrow p$
8. **end_if**
9. **else**
10. $\text{slijedeci}(p) \leftarrow \text{slijedeci}(\text{tekuci}[L])$
11. $\text{slijedeci}(\text{tekuci}[L]) \leftarrow p$
12. **if** ($dduzina[L] = 0$) **then**
13. $\text{kraj}[L] \leftarrow \text{slijedeci}(\text{tekuci}[L])$
14. **end_if**
15. **end_if**
16. $dduzina[L] \leftarrow dduzina[L] + 1$

Umetanje elementa na bilo koju poziciju unutar liste

- Prvo se kreira novi čvor koji se smjesti novi element (1-2).
- Nadalje, u komponentu slijedeci novog čvora, je potrebno upisati adresu koja upućuje na čvor koji je trenutno na čelu desne particije (linija 10), dok se u komponentu slijedeci zadnjeg čvora lijeve particije (na kojeg pokazuje tekuci), treba upisati adresu koja upućuje na novi čvor koji se umeće (linija 11).
- Na kraju se atribut dduzina ažurira tako da se povećava za 1 (linija 16).
- **Ista procedura obrađuje i umetanje novog čvora na početak liste (linije 3-4), te u kojoj je potrebno ažurirati pokazivač pocetak (linija 5).**
- Takva situacija je ilustrirana na slici izvođenjem poziva procedure UMETNI(L, 9).
- Početno stanje je prikazano na slici a, dok je prvi korak (kreiranje novog čvora i upisivanje sadržaja $x=9$) prikazan na slici b. Prilikom povezivanja liste potrebno je jedino povezati novi čvor sa čvorom 5 (slika c).
- Dodatno je potrebno ažurirati pokazivač pocetak, tako da pokazuje na novi čvor 9, te je na kraju potrebno povećati atribut dduzina za 1, jer je novi element umetnut u desnu particiju (slika d).

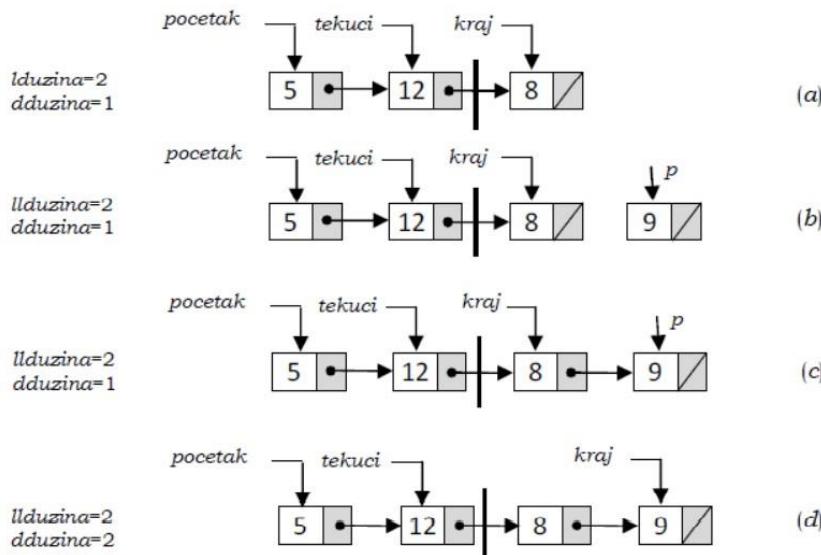


- Ista procedura obrađuje i umetanje novog čvora na kraj liste
- U tom slučaju je potrebno ažurirati pokazivač kraj (linije 12-13). Početno stanje liste je prikazano na slici a. Nakon kreiranja novog čvora i upisivanja elementa 9 (slika b), lista se povezuje tako da se u komponentu slijedeci zadnjeg čvora upisuje adresa novog čvora p (slika c).
- Pošto se testiranjem odgovarajućeg uvjeta utvrđuje da je desna particija prethodno bila prazna, ažurira se pokazivač kraj, tako da pokazuje na novi čvor, te se povećava atribut dduzina za 1 (slika d).



Dodavanje novog čvora x=9

- Operacija dodavanja je ilustrirana na slici izvođenjem poziva procedure DODAJ (L, 9), koja početnu konfiguraciju liste (5, 12 | 8) pretvara u konfiguraciju (5, 12 | 8, 9).
- Početno stanje je prikazano na slici a, dok je prvi korak (kreiranje novog čvora i upisivanje sadržaja x=9) prikazan na slici b.
- Prilikom povezivanja liste potrebno je jedino povezati novi čvor sa čvorom 8 (slika c). Međutim, dodatno je potrebno ažurirati pokazivač kraj, tako da pokazuje na novi čvor 9, te je na kraju potrebno povećati atribut dduzina za 1, jer je novi element umetnut u desnu particiju (slika d)



Procedura dodavanja:

DODAJ (L, x)

1. $p \leftarrow \text{GETNODE}()$
2. $\text{info } p \leftarrow x$
3. $\text{slijedeci}(p) \leftarrow \text{NULL}$
4. **if** ($\text{Iduzina}[L] + \text{dduzina}[L] = 0$) **then**
5. $\text{pocetak}[L] \leftarrow \text{kraj}[L] \leftarrow p$
6. **else**
7. $\text{kraj}[L] \leftarrow \text{slijedeci}(\text{kraj}[L]) \downarrow p$
8. **end_if**
9. $\text{dduzina}[L] \leftarrow \text{dduzina}[L] + 1$

- Prvo se kreira novi čvor pa u njega smješta novi element x, a zatim se u komponentu slijedeci upisuje NULL, jer se novi čvor dodaje na kraj (linije 1-3).
- Nadalje, u komponentu slijedeci zadnjeg čvora se upisuje adresa koja upućuje na novi čvor, a pokazivač kraj se ažurira tako da pokazuje na novi čvor (linija 7).
- Na kraju se povećava atribut dduzina za 1 (linija 9).
- Procedura DODAJ uzima u obzir i jedan specijalan slučaj, koji se pojavljuje u situaciji kada se novi čvor umeće u praznu listu, u kojem je dodatno potrebno ažurirati i pokazivač pocetak (linije 4-5).

Procedura izbacivanja bilo kojeg čvora:

IZBACI (L)

1. **if** ($\text{dduzina}[L] = 0$) **then**
2. ERROR(„Nema nista za izbaciti“)
3. **end_if**
4. **if** ($\text{Iduzina}[L] = 0$) **then**
5. $x \leftarrow \text{info}(\text{pocetak}[L])$
6. $p \leftarrow \text{pocetak}[L]$
7. $\text{pocetak}[L] \leftarrow \text{slijedeci}(p)$
8. **else**
9. $x \leftarrow \text{info}(\text{slijedeci}(\text{tekuci}[L]))$
10. $p \leftarrow \text{slijedeci}(\text{tekuci}[L])$
11. $\text{slijedeci}(\text{tekuci}[L]) \leftarrow \text{slijedeci}(p)$
12. **end_if**
13. **if** ($\text{dduzina}[L] = 1$) **then**
14. $\text{kraj}[L] \leftarrow \text{tekuci}[L]$
15. **end_if**
16. $\text{FREENODE}(p)$
17. $\text{dduzina}[L] \leftarrow \text{dduzina}[L] - 1$
18. **return** x

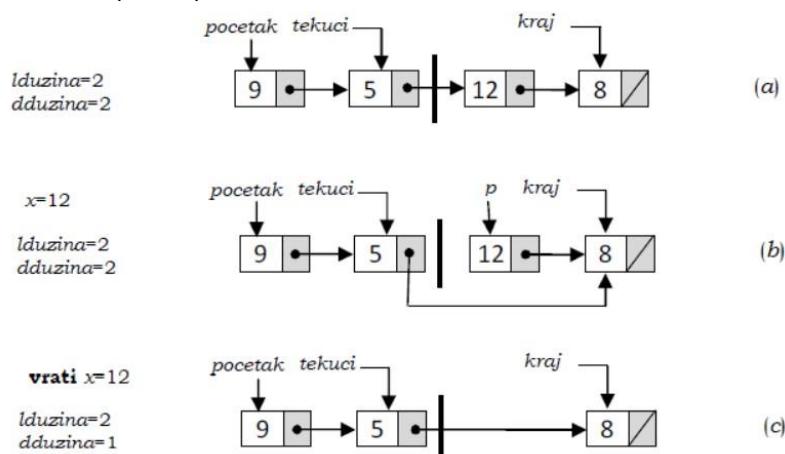
- Ako je desna participija prazna, funkcija prijavljuje grešku, kojom se signalizira da se nema šta izbaciti iz liste (linije 1-3). Inače, operacija izbacivanja u privremenom objektu x čuva informacioni sadržaj, a lista se povezuje tako da se adresa, iz komponente slijedeci čvora koji se briše, upisuje u komponentu

slijedeci čvora na koji pokazuje tekuci (linija 11), pri čemu se adresa čvora koji se briše čuva u pokazivaču p (linija 10), radi oslobođanja memorijskog prostora koji je taj čvor zauzimao (linija 16).

- Na kraju se ažurira dužina desne particije, tako da se atribut dduzina umanji za 1 (linija 17), te se vraća informacioni sadržaj čvora koji se briše (linija 18).

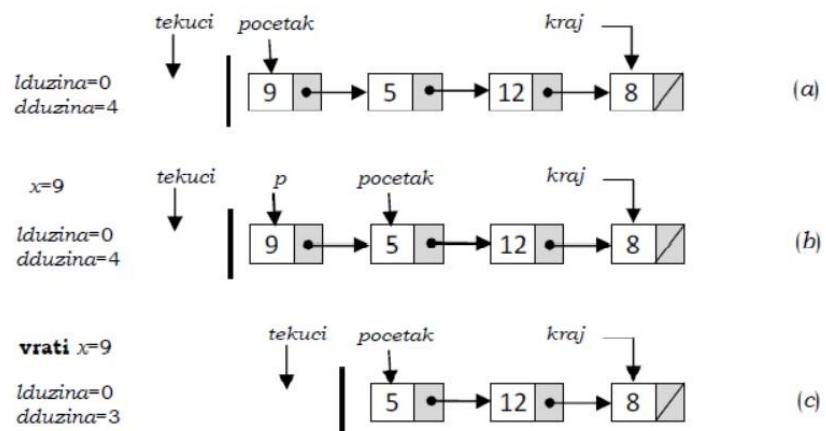
Izbacivanje čvora

- Procedura je ilustrirana na slici operacijom brisanja čvora sa sadržajem x=12, koja početnu konfiguraciju (9, 5 | 12, 8) pretvara u konfiguraciju (9, 5 | 8).
- Početno stanje liste je prikazano na slici a. Nakon upisivanja informacionog sadržaja 12 u privremenu varijablu x=12, lista se povezuje tako da se čvor, na koji pokazuje tekuci (prethodnik čvora 12), povezuje sa čvorom 8, koji je bio sljedbenik čvora 12.
- Na taj način se čvor 12 izbacuje iz logičkog poretka liste (slika b). Na kraju je još i fizički uklonjen čvor 12, tako što je oslobođen memorijski prostor koji je taj čvor zauzimao, te je dužina desne particije umanjena na vrijednost dduzina=1 (slika c).



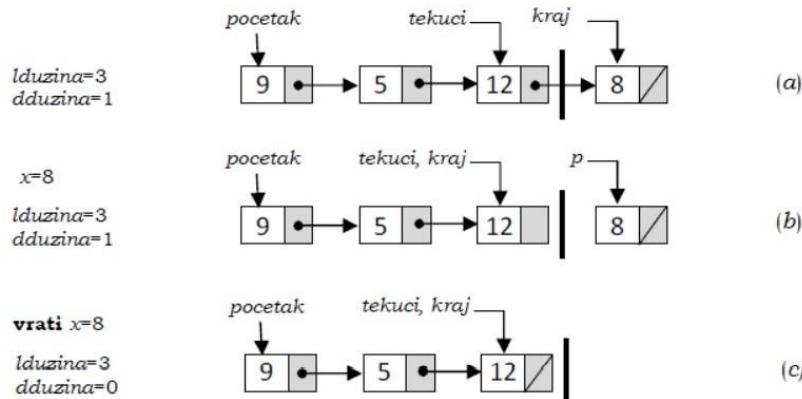
Izbacivanje čvora na početku liste

- Procedura IZBACI uzima u obzir i specijalan slučaj, koji nastaje kada se izbacuje element na početku liste, u kojem nije potrebno listu povezati, ali je potrebno sačuvati adresu čvora koji se briše i ažurirati pokazivač *pocetak*, tako da pokazuje na sljedbenika prvog čvora (linije 4-7). Ova situacija je ilustrirana na slici izvođenjem procedure IZBACI, koja početnu konfiguraciju liste (| 9, 5, 12, 8) pretvara u konfiguraciju (| 5, 12, 8).
- Početno stanje je prikazano na slici a, dok je ažuriranje pokazivača *pocetak* ilustrirano na slici b, čime je prvi element u logičkom smislu izbačen iz liste.
- Na kraju se čvor 9 i fizički uklanja iz liste, te se ažurira atribut *dduzina* na vrijednost *dduzina*=3 (slika c).



Izbacivanje čvora na kraju liste

- Konačno, procedura IZBACI uzima u obzir i specijalnu situaciju, koja nastaje kada se izbacuje zadnji čvor, u kojoj je potrebno ažurirati pokazivač kraj, tako da pokazuje na prethodnika (čvor na koji pokazuje tekuci) zadnjeg čvora (linije 13-15). Ovakva situacija je ilustrirana na slici izvođenjem procedure IZBACI, koja početnu konfiguraciju liste (9, 5, 12 | 8) pretvara u konfiguraciju (9, 15, 12 |).
- Početno stanje je prikazano na slici a, dok je ažuriranje pokazivača kraj ilustrirano na slici b, čime je zadnji element u logičkom smislu izbačen iz liste.
- Na kraju se čvor 8 i fizički uklanja iz liste, te se ažurira atribut dduzina na vrijednost dduzina=0 (slika c).



Procedura pretraživanje liste:

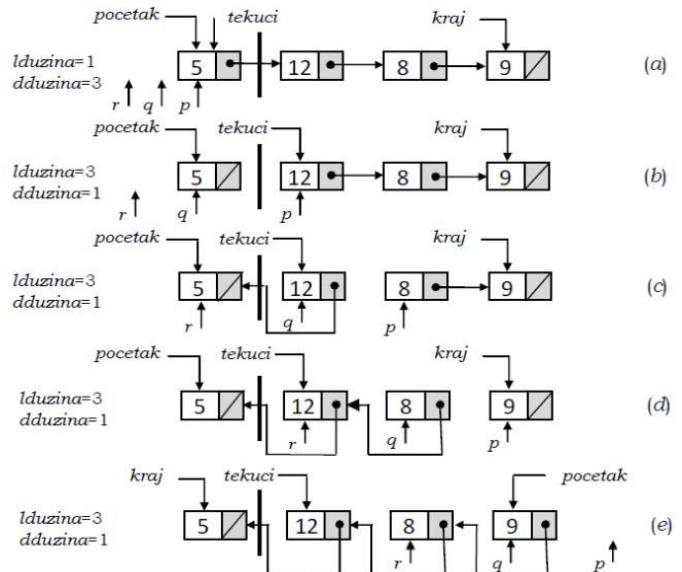
TRAZI (L, x)

- if** (lduzina[L]+dduzina[L] = 0) **then**
- ERROR**(„Lista je prazna“)
- end_if**
- p \leftarrow pocetak[L]
- while** ((p \neq NULL) **and** info(p) \neq x) **do**
- p \leftarrow slijedeci(p)
- end_while**
- return** p \neq NULL

- Funkcija vraća vrijednost true, ako je traženi element prisutan u listi, odnosno vrijednost false, ako traženi element nije prisutan u listi.
- Funkcija TRAZI, počevši od čvora na koji pokazuje pocetak, linearnim pretraživanjem pronalazi prvi čvor koji sadrži element x. Pretraživanje se obavlja sekvencijalno, počevši od čela liste (linija 4), i sljedeći lanac pokazivača sve dok se ne pronađe traženi informacioni sadržaj, ili dok se ne dođe do kraja liste (linije 5-7).
- Da bi se pretražila lista koja sadrži n čvorova, potrebno je u najgorem slučaju pretražiti čitavu listu, pa zaključujemo da je u najgorem slučaju vremenska složenost pretraživanja liste $O(n)$.

Obrtanje porekta čvorova u listi

- Obrtanje porekta čvorova u listi je ilustrirano na slici. Početno stanje povezane liste je prikazano na slici a, dok su koraci pri izvođenju operacije obrtanja prikazani na slikama b-e.
- Početna konfiguracija liste ($5 | 12, 8, 9$) se, nakon izvođenja operacije, pretvara u konfiguraciju ($9, 8, 12 | 5$).
- Prema tome, osim obrtanja porekta elemenata, također su međusobno razmijenjene i vrijednosti broja elemenata u lijevoj (lDUZINA) i desnoj (dDUZINA) particiji



Procedura obrtanja porekta u listi:

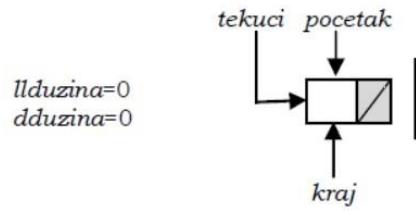
OBRNI-POREDAK (L)

1. $p \leftarrow \text{pocetak}[L]$
2. $q \leftarrow \text{NULL}$
3. **if** ($\text{lDUZINA}[L] = 0$) **then**
4. $\text{tekuci}[L] \leftarrow \text{pocetak}[L]$
5. **else if** ($\text{dDUZINA}[L] = 0$) **then**
6. $\text{tekuci}[L] \leftarrow \text{NULL}$
7. **else**
8. $\text{tekuci}[L] \leftarrow \text{slijedeci}(\text{tekuci}[L])$
9. **end_if**
10. $\text{dDUZINA}[L] \leftarrow \text{lDUZINA}[L]$
11. **while** ($p \neq \text{NULL}$) **do**
12. $r \leftarrow q$
13. $q \leftarrow p$
14. $p \leftarrow \text{slijedeci}(p)$
15. $\text{slijedeci}(q) \leftarrow r$
16. **end_while**
17. $\text{kraj}[L] \leftarrow \text{pocetak}[L]$
18. $\text{pocetak}[L] \leftarrow q$

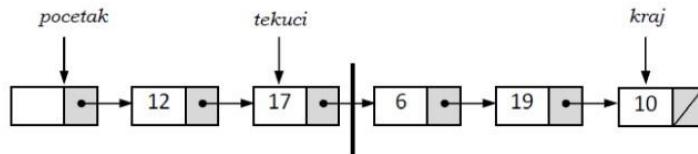
- Ažuriranje pokazivača tekuci, tako da pokazuje na svog sljedbenika (linija 8). Operacija OBRNI-POREDAK uzima u obzir i specijalan slučaj, koji se pojavljuje u situaciji kada je lijeva particija prazna (linije 3-4), kao i slučaj koji se pojavljuje kada je desna particija bila prazna (linije 5-6).
- Operacija koristi tri tekuća pokazivača (r, q i p), koji se slijedno pomjeraju od početka prema kraju liste, tako da p pokazuje na sljedbenika čvora na kojeg pokazuje q , dok r pokazuje na prethodnika čvora na kojeg pokazuje q . While petlja pomiče pokazivače (11-16).
- Na kraju se još pokazivač $kraj$ ažurira tako da pokazuje na prethodno čelo liste (linija 17), dok se pokazivač $pocetak$ postavlja na vrijednost tekućeg pokazivača q (linija 18).

Jednostruko povezane liste sa fiktivnim čvorom:

- Da se ne bi uzimali specijalni slučajevi kada je lista prazna što povećava kompleksnost operacija uvode se liste sa fiktivnim čvorom (staviti čvor na čelo liste)
- Informacioni sadržaj ovakvog čvora se zanemaruje
- Dakle, uvođenjem fiktivnog čvora se pojednostavljuju implementacije operacija, jer neće biti potrebno uzimati u obzir specijalne slučajeve koji nastaju kada je lista prazna, te kada je lijeva particija prazna, naravno uz povećanje memoriskog prostora
- Umetanje jednog čvora u listu sa fiktivnim čvorom, info $x=13$
- Ovom operacijom se početna konfiguracija liste $(12, 17 | 6, 19, 10)$ pretvara u konfiguraciju $(12, 17 | 13, 6, 19, 10)$. Kao i kod liste bez fiktivnog čvora, umetanje se obavlja na početak desne particije, a pokazivač tekuci pokazuje na zadnji element lijeve particije.

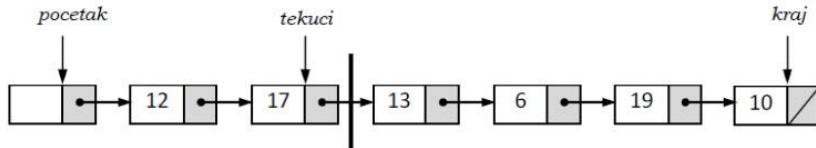


*l_duzina=2
d_duzina=3*



(a)

*l_duzina=2
d_duzina=4*



(b)

Procedura – jednostruka lista – fiktivan čvor

IDI-NA-SLIJEDECI(L)

- if** ($d_{d}uzina[L] \neq 0$) **then**
- $tekuci[L] \leftarrow slijedeci(tekuci[L])$
- $l_{d}uzina[L] \leftarrow l_{d}uzina[L] + 1$
- $d_{d}uzina[L] \leftarrow d_{d}uzina[L] - 1$
- end_if**

- Ako je desna particija prazna onda se ne dešava ništa
- U suprotnom pokazivač tekuci se pomjera na sljedeći čvor (5) nakon čega se $l_{d}uzina$ povećava za 1, a desna dužina smanjuje na 1

Procedura – jednostruka lista – fiktivan čvor

IDI-NA-PRETHODNI(L)

1. **if** ($l_{\text{duzina}}[L] \neq 0$) **then**
2. $p \leftarrow p_{\text{ocetak}}[L]$
3. **while** $s_{\text{lijedeci}}(p) \neq t_{\text{ekuci}}[L]$ **do**
4. $p \leftarrow s_{\text{lijedeci}}[p]$
5. **end_while**
6. $t_{\text{ekuci}}[L] \leftarrow p$
7. $l_{\text{duzina}}[L] \leftarrow l_{\text{duzina}}[L] - 1$
8. $d_{\text{duzina}}[L] \leftarrow d_{\text{duzina}}[L] + 1$
9. **end_if**

- Ako je lijeva participija prazna neće se izvršiti slijed operacija
- Ako nije mora se slijediti lanac pokazivača počevši od početka liste (2) pa sve do prethodnika (3-6)
- Na kraju se ažuriraju u linijama 7 i 8 atributi l_{duzina} i d_{duzina}

IDI-NA-POZICIJU (L, i)

- Ne uzima se u obzir slučaj kada je $i=0$

IDI-NA-POZICIJU (L, i)

- 1 **if** ($i < 0$ **and** $i > l_{\text{duzina}}[L]+d_{\text{duzina}}[L]$) **then**
- 2 ERROR („Indeks izvan dozvoljenog raspona“)
- 3 **end_if**
- 4 $d_{\text{duzina}}[L] \leftarrow d_{\text{duzina}}[L]+l_{\text{duzina}}[L]-i$
- 5 $l_{\text{duzina}}[L] \leftarrow i$
- 6 $t_{\text{ekuci}}[L] \leftarrow p_{\text{ocetak}}[L]$
- 7 **for** $j \leftarrow 0$ **to** $i-1$ **do**
- 8 $t_{\text{ekuci}}[L] \leftarrow s_{\text{lijedeci}}(t_{\text{ekuci}}[L])$
- 9 **end_for**

IDI-NA-POZICIJU-M (L, i)

- Ne uzima se u obzir slučaj kada je $i=0$

IDI-NA-POZICIJU-M (L, i)

- 1 **if** ($i < 0$ **and** $i > l_{\text{duzina}}[L]+d_{\text{duzina}}[L]$) **then**
- 2 ERROR („Indeks izvan dozvoljenog raspona“)
- 3 **end_if**
- 4 $d_{\text{duzina}}[L] \leftarrow d_{\text{duzina}}[L]+l_{\text{duzina}}[L]-i$
- 5 $l_{\text{duzina}}[L] \leftarrow i$
- 6 $p \leftarrow 0$
- 7 $l_d \leftarrow l_{\text{duzina}}[L]$
- 8 **if** ($i \geq l_d$) **then**
- 9 $p \leftarrow l_d$
- 10 **else**
- 11 $t_{\text{ekuci}}[L] \leftarrow p_{\text{ocetak}}[L]$
- 12 **end_if**
- 13 **for** $j \leftarrow p$ **to** $i-1$ **do**
- 14 $t_{\text{ekuci}}[L] \leftarrow s_{\text{lijedeci}}(t_{\text{ekuci}}[L])$
- 15 **end_for**

UMETNI (L, i)

- Zbog toga što nije potrebno posebno uzimati u obzir slučaj kada je lijeva particija liste prazna, procedura je pojednostavljena u odnosu na odgovarajuću proceduru kod implementacije liste bez fiktivnog čvora

UMETNI (L, x)

```
1 p ← GETNODE ()
2 info(p) ← x
3 sljedeci(p) ← sljedeci(tekuci[L])
4 sljedeci(tekuci[L]) ← p
5 if ( kraj[L] = tekuci[L] ) then
6   kraj[L] ← p
7 end_if
8 ddruzina[L]←ddruzina[L]+1
```

```
1. p ← GETNODE ()
2. info p ← x
3. if (lDUZINA[L] = 0) then
4.   sljedeci(p) ← pocetak[L]
5.   pocetak[L] ← p
6.   if (lDUZINA[L]+ddUZINA[L] = 0) then
7.     kraj[L] ← p
8.   end_if
9. else
10.   sljedeci(p) ← sljedeci(tekuci[L])
11.   sljedeci(tekuci[L]) ← p
12.   if (ddUZINA[L] = 0) then
13.     kraj[L] ← sljedeci(tekuci[L])
14.   end_if
15. end_if
16. ddUZINA[L] ← ddUZINA[L]+1
```

DODAJ (L, x)

- U ovom slučaju je operacija pojednostavljena, jer se nije trebao posebno tretirati slučaj kada je lista prazna

DODAJ (L, x)

```
1 p ← GETNODE ()
2 kraj[L] ← sljedeci(kraj[L]) ← p
3 ddruzina[L] ← ddruzina[L]+1
```

IZBACI (L)

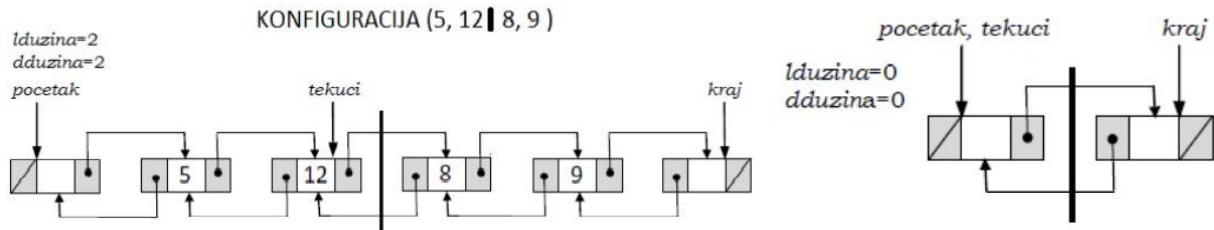
- Implementacija funkcije je pojednostavljena, jer se nije trebao posebno tretirati slučaj kada je lijeva particija liste prazna.
- Funkcija vraća informacioni sadržaj elementa koji se izbacuje

IZBACI (L)

```
1 if ( ddruzina[L] = 0 ) then
2   ERROR („Nista za izbaciti!“)
3 end_if
4 x ← info(sljedeci(tekuci[L]))
5 p ← sljedeci(tekuci[L])
6 sljedeci(tekuci[L]) ← sljedeci(p)
7 if (p = kraj[L]) then
8   kraj[L] ← tekuci[L]
9 end_if
10 FREENODE (p)
11 ddruzina[L] ← ddruzina[L]+1
12 return x
```

Dvostruko povezane liste:

- Neefikasnost jednostrukopovezane liste vodi do ovog rješenja zato što ako trebamo doći do prethodnika nekog čvora moramo preći od čela liste do tog prethodnika
- Jedan od načina kako možemo povećati efikasnost kretanja kroz povezanu listu je da svaki čvor pored pokazivača na sljedeći čvor (slijedeći), sadrži i pokazivač na prethodni čvor (prethodni). Na taj način ćemo dobiti dvostruko povezanu listu.
- Fiktivni čvor može biti na čelu ali i na začelju liste



Procedura

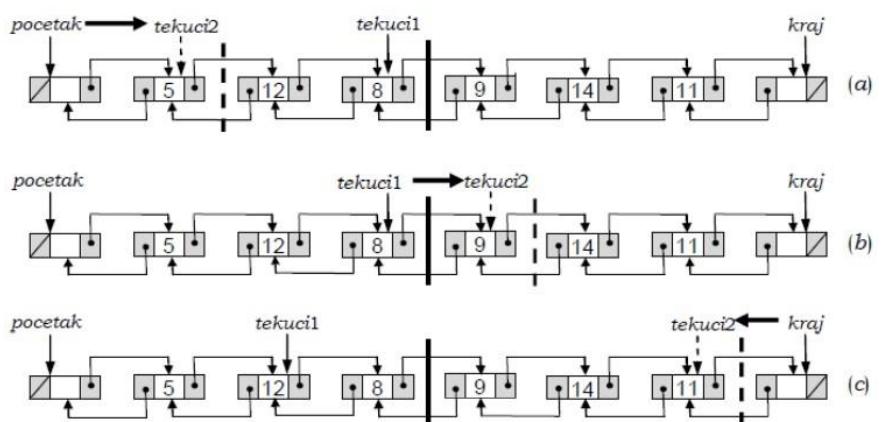
IDI-NA-PRETHODNI(L)

1. **if** ($l_duzina[L] \neq 0$) **then**
2. $tekuci[L] \leftarrow prethodni(tekuci[L])$
3. $l_duzina[L] \leftarrow l_duzina[L] - 1$
4. $d_duzina[L] \leftarrow d_duzina[L] + 1$
5. **end_if**

- Ako je lijeva particija prazna neće se izvršiti slijed operacija
- Ako nije mora se slijediti lanac pokazivača počevši od početka liste (2) pa sve do prethodnika (3-6)
- Na kraju se ažuriraju u linijama 7 i 8 atributi l_duzina i d_duzina

Pristup od početka, kraja ili od tekućeg čvora

- Na slici su ilustrirane tri moguće situacije za početnu konfiguraciju liste (5, 12, 8 | 9, 14, 11).
- Ako se, na primjer, pristupa poziciji $i=1$, tada se pristup najefikasnije ostvaruje u jednom koraku počevši od početnog čvora (slika a).
- S druge strane, ako se pristupa poziciji $i=4$, tada se pristup u jednom koraku ostvaruje iz tekućeg čvora (slika b).
- I konačno, ako se pristupa poziciji $i=6$, pristup samo u jednom koraku se ostvaruje iz krajnjeg čvora, koristeći vezu na prethodnika (c).



- IDI-NA-POZICIJU procedura na osnovu prethodno opisanog algoritma.
- Ispitivanjem odgovarajućih uvjeta se utvrđuje iz koje pozicije se u najmanjem broju koraka k može ostvariti pristup (linije 8-16).
- Nakon toga se korištenjem veza na prethodnike (linije 18-20) ili na sljedbenike ostvara pristup u k koraka (linije 22-24).

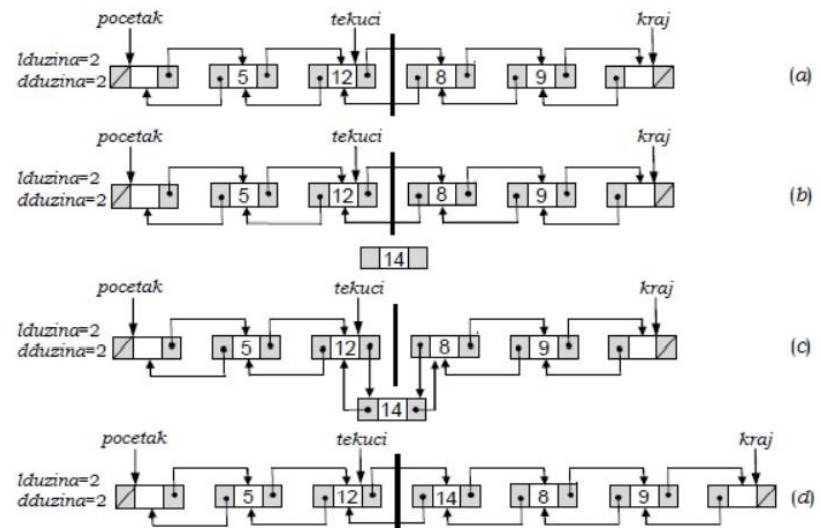
```

IDI-NA-POZICIJU-M ( $L, i$ )
1  if ( $i < 0$  or  $i \geq lduzina[L] + dduzina[L]$ ) then
2    ERROR („Indeks izvan dozvoljenog raspona“)
3  end_if
4   $ld \leftarrow lduzina[L]$ 
5   $dd \leftarrow dduzina[L]$ 
6   $dduzina[L] \leftarrow dduzina[L] + lduzina[L] - i$ 
7   $lduzina[L] \leftarrow i$ 
8  if ( $|i - ld| < |i - dd|$  and ( $i < (ld + dd)/2$ )) then
9     $k \leftarrow i$ 
10    $tekuci[L] \leftarrow pocetak[L]$ 
11  else if ( $|i - ld| \leq |ld + dd - i|$ ) then
12     $k \leftarrow i - ld$ 
13  else
14     $k \leftarrow i - ld - dd - 1$ 
15     $tekuci[L] \leftarrow kraj[L]$ 
16  end_if
17  if ( $k < 0$ ) then
18    for  $i \leftarrow 0$  to  $|k|$  do
19       $p \leftarrow prethodni(tekuci[L])$ 
20    end_for
21  else
22    for  $i \leftarrow 0$  to  $|k|$  do
23       $p \leftarrow sljedeci(tekuci[L])$ 
24    end_for
25  end_if

```

Umetanje čvora:

- Operacija kojom se početna konfiguracija liste $(5, 12 | 8, 9)$, umetanjem elementa 14, pretvara u konfiguraciju $(5, 12 | 14, 8, 9)$, je ilustrirana na slici.
- Slika a prikazuje početno stanje, dok je na slici b prikazano stanje nakon kreiranja novog čvora i upisivanja sadržaja 14.
- Stanje nakon povezivanja liste je prikazano na slici c, dok je na slici d, samo grafički drugačije prikazano stanje liste, uz ažuriranu dužinu desne particije $dduzina=3$



UMETNI (L , x) procedura umetanja.

- Nakon kreiranja novog čvora i upisivanja informacionog sadržaja (linije 1-2), lista se povezuje tako da novi čvor postaje sljedbenik tekućeg čvora (linije 3 i 5), te prethodnik bivšeg sljedbenika tekućeg čvora (linije 4 i 6). Na kraju se ažurira dužina desne particije (linija 7).

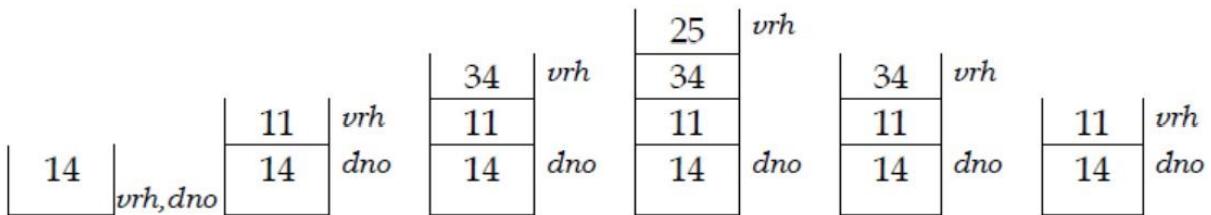
UMETNI (L , x)

```
1   $p \leftarrow \text{GETNODE}()$ 
2   $\text{info}(p) \leftarrow x$ 
3   $\text{prethodni}(p) \leftarrow \text{tekuci}[L]$ 
4   $\text{sljedeci}(p) \leftarrow \text{sljedeci}(\text{tekuci}[L])$ 
5   $\text{sljedeci}(\text{tekuci}[L]) \leftarrow p$ 
6   $\text{prethodni}(\text{sljedeci}(p)) \leftarrow p$ 
7   $dduzina[L] \leftarrow dduzina[L]+1$ 
```

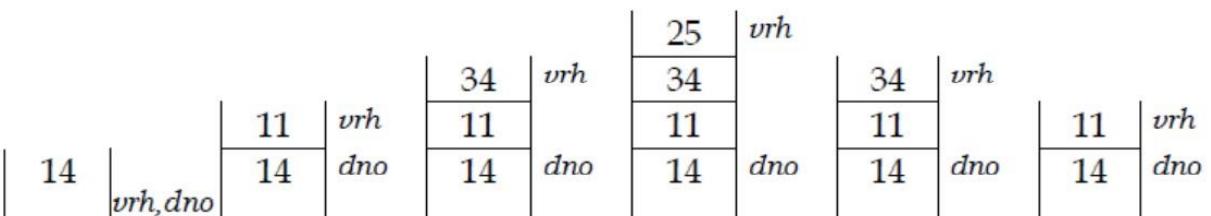
(Predavanje V)

Stek

- Stek je struktura podataka kod koje se elementi mogu umetati ili uklanjati samo na jednom kraju strukture. Taj kraj se naziva vrh steka (eng. top).
- Drugi kraj strukture podataka se naziva dno steka (eng. bottom).
- Umetanjem novog elementa na stek formira se novi vršni element.
- Pošto se sa steka uklanja element sa vrha steka, a to je element koji je zadnji umetnut, za stek možemo reći da ima LIFO (eng. Last In, First Out) organizaciju.
- Pretpostavimo da na prazan stek prvo stavljamo četiri cijelobrojna elementa sljedećim redoslijedom: 14, 11, 34, 25. Zatim, neka u radu sa stekom slijede dvije operacije skidanja sa steka. Niz operacija stavljanja na stek i skidanja sa steka je ilustriran na slici



- Općenito govoreći, koncept steka ne zahtijeva da elementi obavezno moraju biti istog tipa. Međutim, ako je riječ o sekvenčijalnoj implementaciji koncepta steka, onda su elementi steka istog tipa, što znači da je stek homogena struktura podataka



Tipične operacije u radu sa stekom su:

- stavljanje elemenata na stek (eng. push);
- skidanje elemenata sa steka (eng. pop);
- provjera da li je stek prazan;
- provjera da li je stek pun;
- čitanje vršnog elementa, bez njegovog skidanja.

IMPLEMENTACIJA STEKA POMOĆU NIZOVA:

- Sekvenčijalna implementacija steka, pri čemu se za smještanje elemenata steka koristi niz $S[0..n-1]$.
- Pretpostavlja se da niz S ima cijelobrojni atribut vrh , koji predstavlja indeks vrha steka. Prema tome, sadržaj steka u nekom trenutku je $S[0], S[1], \dots, S[vrh]$ – gdje je $S[0]$ element na dnu steka, a $S[vrh]$ je element na vrhu steka (kada je stek prazan $vrh=-1$)

Operacija umetanja novog elementa na stek:

STAVI-NA-STEK (S, x)

- Prvo se pozivom funkcije JE-LIPUN(S) provjerava da li je stek S pun. Ako je stek pun, novi element se ne može umetnuti, te se signalizira prekoračenje kapaciteta steka (linija 2).
- Inače, ako stek nije pun, povećava se oznaka vrha steka za 1, tako da pokazuje na prvu sljedeću slobodnu lokaciju, te se na tu lokaciju upisuje novi element x (linije 4-5)

STAVI-NA-STEK (S, x)

```

1 if (JE-LI-PUN( $S$ )) then
2   ERROR („Stek je pun!“)
3 else
4    $vrh[S] \leftarrow vrh[S]+1$ 
5    $S[vrh[S]] \leftarrow x$ 
6 end_if
-----
```

JE-LI-PUN (S)

- Slika prikazuje pun stek pod a i prazan stek pod b
- Funkcija provjerava da li je stek pun. Ako je stek pun vraća true odnosno na stek se ne može dodati element odnosno vraća false i na taj način osigurava da se izvrši komanda else u funkciji STAVI-NA-STEK

0	23
1	15
2	30
...	...
$n-1$	18

$vrh = n-1$
(a)

PUN STEK

0	
1	
2	
...	...
$n-1$	

$vrh = -1$
(b)

PRAZAN STEK

JE-LI-PUN(S)

```

1 if ( $vrh[S] = n-1$ ) then
2   return true
3 else
4   return false
5 end_if
-----
```

Operacija skidanja elementa na steka:

SKINI-SA-STEKA (S)

- Prvo se pozivom funkcije JE-LIPRAZAN(S) ispituje da li je stek S prazan. Ako je stek prazan, operacija skidanja sa steka se ne može izvršiti, pa se odgovarajućom porukom signalizira da je stek prazan (linije 1-2).
- U suprotnom, ako stek nije prazan, uzima se element sa vrha steka i funkcija SKINI-SA-STEKA vraća taj element, pri čemu se oznaka vrha steka smanjuje za 1 (linije 4-6). Treba primjetiti da memorija nije oslobođena

SKINI-SA-STEKA (S)

```

1 if (JE-LI-PRAZAN( $S$ )) then
2   ERROR („Stek je prazan!“)
3 else
4    $x \leftarrow S[vrh[S]]$ 
5    $vrh[S] \leftarrow vrh[S]-1$ 
6   return  $x$ 
7 end_if
-----
```

Operacija umetanja novog elementa na stek:

JE-LI-PRAZAN (S)

- Funkcija provjerava da li je stek prazan. Ako je stek prazan vraća true odnosno sa steka se ne može skinuti element odnosno vraća false i na taj način osigurava da se izvrši komanda else u funkciji SKINI-SA-STEKA

```
-----  
JE-LI-PRAZAN (S)  
1 if (vrh[S] = -1) then  
2   return true  
3 else  
4   return false  
5 end_if
```

Operacija čitanja elementa na vrhu steka:

ELEMENT-NA-VRHU (S)

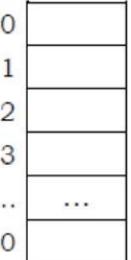
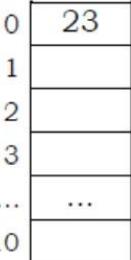
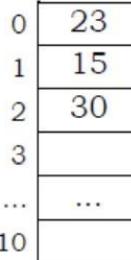
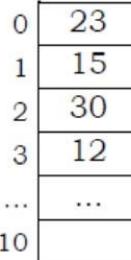
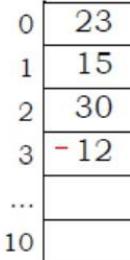
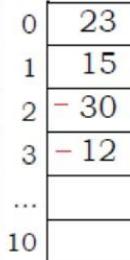
- Funkcija čita vrijednost elementa sa steka bez njegovog skidanja

```
-----  
ELEMENT-NA-VRHU (S)  
1 return S[vrh[S]]
```

Niz operacija umetanja i skidanja elementa steka:

- Niz operacija dodavanja elemenata (1-5 slika)
- Dvije operacije skidanja elemenata (6 i 7 slika)
- Primjetiti 6 i 7 sliku da su elementi iznad vrijednosti steka u memoriji
- Dodavanje jednog elementa

STAVI-NA-STEK (S, 23)	STAVI-NA-STEK (S, 15)	STAVI-NA-STEK (S, 30)	STAVI-NA-STEK (S, 12)	SKINI-SA-STEKA (S)	SKINI-SA-STEKA (S)	STAVI-NA-STEK (S, 17)
-----------------------	-----------------------	-----------------------	-----------------------	--------------------	--------------------	-----------------------

						
0	0	0	0	0	0	0
1	1	1	1	1	1	1
2	2	2	2	2	2	2
3	3	3	3	3	3	3
...
10	10	10	10	10	10	10

$vrh[S] = -1$ $vrh[S] = 0$ $vrh[S] = 1$ $vrh[S] = 2$ $vrh[S] = 3$ $vrh[S] = 2$ $vrh[S] = 1$ $vrh[S] = 2$

- Kod sekvenčnog steka se može primjetiti da je potrebno unaprijed definisati prostor steka koji je fiksne veličine.
- Ako trebamo koristiti stek kao strukturu podataka u područjima primjene u kojima ne možemo predvidjeti dinamiku rasta steka, onda postoji mogućnost prekoračenja rezerviranog memorijskog prostora. • Alociranje potrebnog prostora na maksimalnu očekivanu veličinu, za posljedicu ima neefikasno korištenje memorijskih resursa.
- Problem efikasnijeg korištenja memorijskih resursa kod implementacije steka se može riješiti implementacijom steka pomoću povezanih listi

Operacija umetanja novog elementa na stek pomoću liste:

STAVI-NA-STEK (S, x)

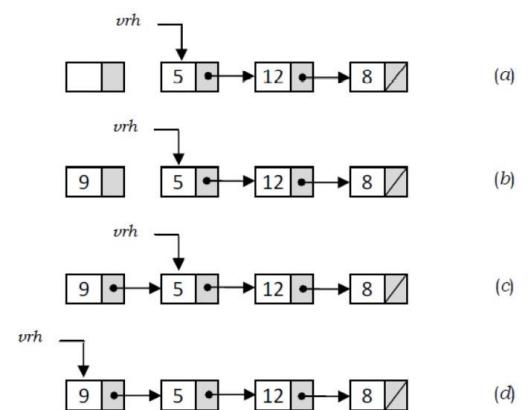
- U prvom koraku se alocira prostor za novi čvor (linija 1), a zatim se u informacioni dio upisuje novi element x (linija 2).
- Na kraju se novi čvor povezuje sa prethodnim čelom liste, dok se pokazivač vrh ažurira tako da pokazuje na novi čvor (linije 3-4)

STAVI-NA-STEK (S, x)

- 1 $p \leftarrow \text{GETNODE}()$
 - 2 $\text{info}(p) \leftarrow x$
 - 3 $\text{sljedeci}(p) \leftarrow \text{vrh}[S]$
 - 4 $\text{vrh}[S] \leftarrow p$
-

STAVI-NA-STEK (S, 9)

- Primjer stavljanja novog elementa na stek implementiran povezanom listom je ilustriran na slici ubacivanjem novog elementa x=9.
- Početno stanje steka i prvi korak alokacije prostora za novi čvor su prikazani na slici a. Zatim se u novi čvor upisuje element x=9 (slika b), te se taj čvor povezuje sa prethodnim čelom liste, na kojeg pokazuje pokazivač vrh (slika c).
- U zadnjem koraku se pokazivač vrh ažurira tako da pokazuje na novokreirani čvor 9 (slika d).



Operacija skidanja elementa na steka:

SKINI-SA-STEKA (S)

- Ako pokazivač vrh ima vrijednost NIL, signalizira se da je stek prazan (linije 1- 3).
- U suprotnom, u privremenim objekat x se upisuje informacioni sadržaj elementa na čelu liste (linija 4), te se u privremenom pokazivaču p čuva adresa trenutnog čela liste (linija 5).
- Nakon toga se ažurira pokazivač vrh, tako da pokazuje na sljedbenika čvora na čelu (linija 6), čime se u logičkom smislu element sa vrha steka izbacuje iz liste.
- Na kraju je potrebno taj čvor i fizički ukloniti (linija 7), te vratiti informacioni sadržaj skinutog elementa (linija 8).

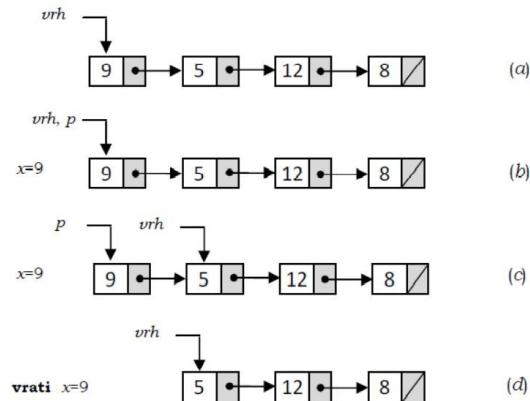
SKINI-SA-STEKA (S)

- 1 **if** ($\text{vrh}[S] = \text{NIL}$) **then**
 - 2 **ERROR** („Stek je prazan!“)
 - 3 **end_if**
 - 4 $x \leftarrow \text{info}(\text{vrh}[S])$
 - 5 $p \leftarrow \text{vrh}[S]$
 - 6 $\text{vrh}[S] \leftarrow \text{sljedeci}(\text{vrh}[S])$
 - 7 **FREENODE** (p)
 - 8 **return** x
-

Operacija skidanja novog elementa sa steka pomoću liste:

SKINI-SA-STEKA (S)

- Operacija skidanja elementa sa vrha steka je ilustrirana na slici.
- Početno stanje je prikazano na slici a. Privremeni pokazivač p se postavlja na čelo liste (slika b), a pokazivač vrh se ažurira da pokazuje na njegovog sljedbenika (slika c).
- Na kraju se oslobađa prostor koji je zauzimao element $x=9$, te se taj element vraća, budući da je prethodno pohranjen u objektu x (slika d).



Operacija čitanja elementa na vrhu steka:

ELEMENT-NA-VRHU (S)

- Funkcija čita vrijednost elementa sa steka bez njegovog skidanja

DODATNA NAPOMENA:

- Ulančana implementacija steka koristi dodatni pokazivač za svaki element steka.
- Međutim, ako se ima na umu da je memorijski prostor koji je potreban za pohranjivanje informacionog sadržaja u većini primjena u relativnom smislu znatno veći od potrebnog prostora za smještanje pokazivača, onda dodatni pokazivač ne predstavlja neko bitnije povećanje ukupno zahtijevanog memorijskog prostora, u odnosu na sekvencijalnu implementaciju.

ELEMENT-NA-VRHU (S)
1 return info(vrh[S])

Primjer primjene stekova za evaluaciju postfiksnih izraza

- U svakodnevnom životu za prikaz aritmetičkih izraza koristi se tzv. infiksnu notaciju. Na primjer, zbroj brojeva 7 i 4 se prikazuje kao $7+4$. Operator $+$ je binarni operator, jer zahtijeva dva operanda: 7 i 4.
- U infiksnoj notaciji, binarni operator se smješta između operanada.
- Iako je infiksna notacija široko rasprostranjena, ipak, takav način zapisa aritmetičkih izraza nije najprikladniji za obradu u različitim računarskim primjenama
- Na primjer, ako se posmatra sljedeći izraz $7+4*3$ (operator $*$ prikazuje množenje), te ako se zanemari konvencija o prioritetima operatora, postavlja se pitanje koja od dviju prisutnih operacija se izvršava prvo.
- Da li se prvo izvršava operacija zbrajanja ili se prvo izvršava operacija množenja?
- Ako se uzme u obzir prihvaćena konvencija, prema kojima operacije množenja i dijeljenja imaju veći prioritet u odnosu na operacije zbrajanja i oduzimanja, onda se prethodni izraz svodi na $7+12$.
- u računarskim sistemima, za primjenu prethodno spomenutog pravila o prioritetima, potrebno je to pravilo ugraditi u aritmetički sistem, kao dodatno pravilo.
- Također potrebno je uvesti dodatnu notaciju u obliku zagrada

- Na primjer, ako se želi prvo zbrojiti 7 i 4, a zatim rezultat zbrajanja pomnožiti sa 3, tada je potrebno prethodni izraz prikazati kao $(7+4)*3$.
- Primjenom zagrada se samo definira pripadnost operanda 4 operatoru sabiranja, a ne operatoru množenja
- Općenito, pripadnost operanda operatoru možemo nedvosmisleno definirati, tako da se operator zajedno sa svojim operandima okruži zgradama.
- Takav oblik zapisa se naziva izraz sa potpunim zgradama. Primjer izraza sa potpunim zgradama: $((24/6)*(7-2))+15$.
- U aritmetičkim izrazima sa potpunim zgradama, redoslijed izvršavanja operatora je po opadajućem nivou ugnježđavanja.
- Prvo se izvršavaju operatori sa najvećim nivoom, nakon toga slijede operatori sa sljedećim manjim nivoom, a na kraju se izvršavaju operatori s nivoom 1.
- U gornjem izrazu, postoje dva podizraza koji imaju nivo 3. To su podizrazi: $(24/6)$ i $(7-2)$. U takvima situacijama, kada više operatora ima isti nivo, izvršavanje operatora se obavlja s lijeve strane, prema desnoj strani. To znači da će se u gornjem aritmetičkom izrazu prvo izračunati izraz $(24/6)$, a zatim izraz $(7-2)$.
- Nakon toga se prelazi na nivo 2, na kojem se nalazi operator $*$, odnosno sljedeći podizraz: $((24/6)*(7-2))$.
- Na nivou 1 se nalazi operator $+$, pa se izračunava izraz $((24/6)*(7-2))+15$, što na kraju daje konačan rezultat. Kod izraza sa potpunim zgradama, nije potrebno definirati prioritet operatora
- Međutim, upravo definiranjem prioriteta operatora se smanjuje broj zagrada.
- Kao primjer, možemo navesti da je operacijama množenja i dijeljenja pridružen veći prioritet, u odnosu na operacije zbrajanja i oduzimanja. Ako se neki operand nalazi između dva operatora sa različitim prioritetom, tada se operand pridružuje operatoru sa definiranim većim prioritetom.
- Na primjer, u izrazu $24/6+15$, operand 6 je pridružen operatoru dijeljenja. U zapisu sa potpunim zgradama, prethodni izraz ima oblik $((24/6)+15)$.
- Ako se želi operand 6 pridružiti operatoru $+$, onda bi zapis prethodnog izraza trebao biti $24/(6+15)$.
- Kod izraza sa nepotpunim zgradama, redoslijed izvršavanja operatora je takav da se prvo izvršavaju operatori na višim nivoima ugnježđavanja zagrada, a zatim operatori na nižim nivoima ugnježđavanja zagrada.
- U situaciji kada ima više operatora u okviru istog para zagrada, izvršavanje operatora je definirano prioritetom, te ako više susjednih operatora ima isti prioritet, redoslijed izvršavanja je definiran smjerom grupiranja.
- Na primjer, za izraz sa nepotpunim zgradama $24/6*(7-2*3)+15$, redoslijed izvršavanja operatora bi bio:

1. $2*3$
2. $7-2*3$
3. $24/6$
4. $24/6*(7-2*3)$
5. $24/6*(7-2*3)+15$

- Prema tome, možemo zaključiti da korištenje infiksne notacije zahtijeva uvođenje dva dodatna koncepta.
- Prvi koncept se odnosi na prioritet operatora, dok se drugi koncept odnosi na upotrebu zagrada, pomoću kojih se može odrediti redoslijed izvršavanja operacija.
- Osim toga, prevodiocu je potrebno da u više navrata ispituje izraz s lijeva udesno, da bi odredio redoslijed izvršavanja operacija.

- Ovaj nedostatak infiksne notacije se može izbjegći upotrebom prikaza aritmetičkih izraza sa drugačijim rasporedom operanada i operatora. Jedan od tih načina prikaza je tzv. postfiksna notacija ili reverzna poljska notacija (eng. reverse Polish notation).
 - Prema postfiksnoj notaciji, kao što i samo ime sugerira, operator se nalazi iza operanada.
 - Na primjer, izraz za zbroj $7 + 4$, napisan u infiksnoj notaciji, bi u postfiksnoj notaciji imao oblik $7\ 4\ +$.
 - Dakle, aritmetički izraz u infiksnoj notaciji pretvaramo u izraz zapisan u postfiksnoj notaciji, tako što se operator sa mjesto između dva operanda premjesti iza drugog operanda.
 - Kod nešto složenijih izraza, najjednostavniji način pretvaranja je da prvo izraz u infiksnoj notaciji zapišemo u obliku sa potpunim zagradama. Zatim, svaki operator prebacimo na mjesto tom operatoru pripadajuće desne zgrade, te na kraju uklonimo sve zgrade iz izraza).
 - Na primjer, ako imamo izraz u infiksnom zapisu
 - $15 + 24/(6-3)+(8-4)*5$, prvo ćemo taj izraz pretvoriti u izraz sa potpunim zagradama
 - $((15+(24/(6-3)))+((8-4)*5))$
 - $((15(24(6\ 3\ -\)\ /\ ((8\ 4\ -\ 5\ *\)\ +\))\ +\)$
 - $15\ 24\ 6\ 3\ -\ /\ 8\ 4\ -\ 5\ *\ +$
- POSTFIKSNA NOTACIJA

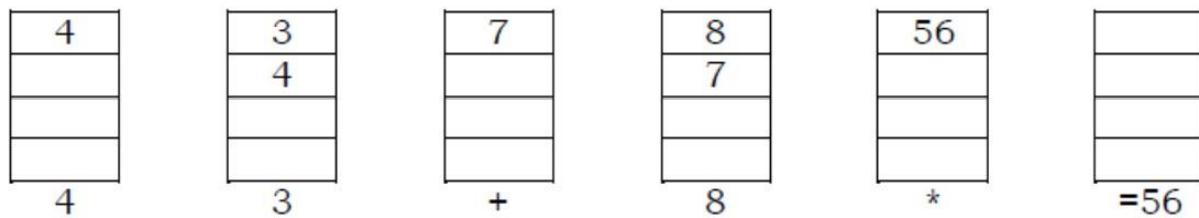
Stekovi u postfiksnoj notaciji – algoritam

1. Inicijalizirati prazan stek S
 - (a) alocirati memoriski prostor za n elemenata u nizu $S[0..n-1]$;
 - (b) inicijalizirati vrh steka, $vrh[S]=-1$.
2. Sve dok u ulaznom postfiksnom izrazu ima simbola, čitati sljedeći simbol x slijeva na desno
 - (a) Ako je pročitani simbol x operand staviti ga na stek S (STAVI-NA-STEK (S, x));
 - (b) Inače, ako je simbol x binarni operator:
 - i. uzeti sa steka dva operanda u varijable a i b ($a \leftarrow SKINI-SA-STEKA(S)$,
 - $b \leftarrow SKINI-SA-STEKA(S)$),
 - ii. primijeniti operator x ($rez \leftarrow a \ x \ b$)
 - iii. rezultat rez staviti na stek (STAVI-NA-STEK(S, rez)).
3. Nakon čitanja svih simbola, sa steka uzeti rezultat ($rez \leftarrow SKINI-SA-STEKA(S)$)
4. Ispitati da li je stek prazan, nakon čitanja rezultata u varijablu rez .
 - (a) ako je stek nakon čitanja rezultata ostao prazan, onda algoritam kao rezultat vraća vrijednost rez .
 - (b) inače, ako stek nije prazan, algoritam signalizira da je ulazni postfiksni izraz nepravilan.

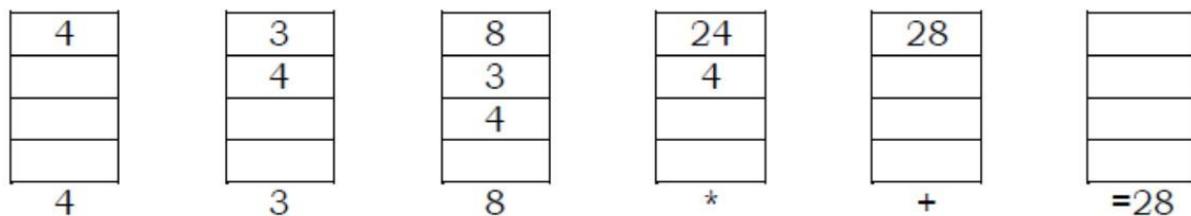
Primjer primjene stekova za evaluaciju postfiksnih izraza

- Na primjer, treba razmotriti sljedeći postfiksni izraz: $4\ 3\ +\ 8\ *$.
- Na početku se prvo pročita 4 i stavlja se na stek. Zatim se čita 3 i također se stavlja na stek.
- Treći pročitani element je operator $(+)$, pa se sa steka uzimaju $(3$ i $4)$ i primjenjuje se operator $+$, što rezultira brojem 7 , koji se u skladu s algoritmom stavlja na stek.
- Sljedeći pročitani broj je 8 , tako da se stavlja na stek. Zadnji pročitani el. je operator $(*)$, što znači da se sa steka trebaju uzeti dva el. $(8$ i $7)$, da bi se primijenio pročitani operator $(*)$. Rezultat primjene

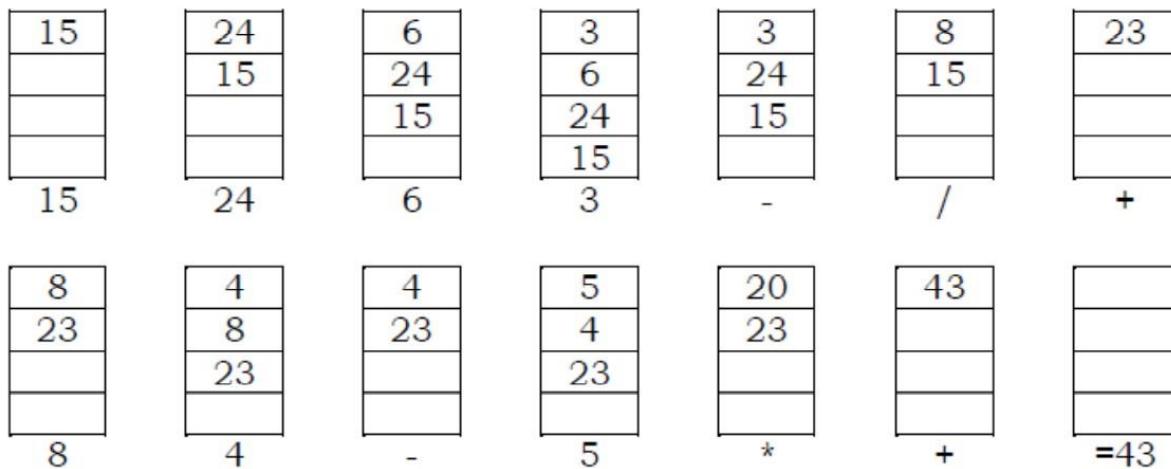
operatora je broj 56, koji se stavlja na stek. Nakon čitanja svih simbola u ulaznom izrazu, u sljedećem koraku se sa steka uzima rezultat (56). Pošto je stek ostao prazan, algoritam vraća rezultat 56.



- Još jedan primjer postfiksni izraz: $4 \ 3 \ 8 \ * \ +$.



- Polazni primjer: $15 \ 24 \ 6 \ 3 \ - \ / \ + \ 8 \ 4 \ - \ 5 \ * \ +$



NAPOMENA

- Na kraju treba ostati jedan element koji predstavlja rezultat.
- Nakon čitanja rezultata, stek bi trebao ostati prazan, ako je postfiksni izraz pravilan.
- U suprotnom, ako nakon uzimanja rezulatata, stek nije prazan, to znači da postfiksni izraz nije pravilan i da sadrži više operanada, nego što to broj operatora zahtijeva.
- Osim toga, greška zbog neispravnosti postfiksног izraza se može pojaviti i u slučajevima kada je u izrazu pročitan znak koji predstavlja operator, pa se pri uzimanju operanada sa steka pojavljuje greška zbog toga što je stek prazan.
- Ovakva situacija se pojavljuje u slučajevima kada pročitani operator nema dovoljno operanada

(Predavanje VI)

Redovi

- Red (eng. queue) je struktura podataka koja je slična steku.
- Za razliku od steka, red ima dva pristupna kraja.
- Na jednom kraju strukture, koji se naziva začelje (eng. tail), se vrši umetanje novih elemenata, dok se izbacivanje elemenata iz reda vrši na drugom kraju strukture, koji se naziva čelo (eng. head).
- Pristup na dva kraja omogućuje da se elementi iz reda uklanaju istim redoslijedom u kojem su i umetnuti. Element koji je ranije umetnut u strukturu će biti ranije i uklonjen iz strukture. To znači da se iz reda uklanja onaj element koji je u red umetnut najranije. Ovakva organizacija pristupa elementima strukture se naziva FIFO (eng. First In, First Out) – „prvi unutra, prvi van“.

Operacija umetanja i izbacivanja iz reda:

- STAVI-U-RED
- IZVADI-IZ-REDA
- Možemo se primjetiti da se prvo umeću tri elementa (6, 15 i 7) na začelje reda, a zatim se operacijom izbacivanja uklanja element 6 sa čela reda.
- Nakon toga, slijedi još jedna operacija umetanja (18), te jedna operacija izbacivanja. Prema tome, sukladno operacijama umetanja i izbacivanja elemenata, mijenjaju se: broj elemenata u redu, čelo reda (pocetak) i začelje reda (kraj).
- Zato možemo reći da je red dinamička struktura podataka. Osim toga, elementi reda su obično istog tipa, što znači da je red i homogena struktura podataka

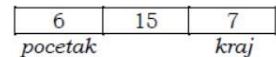
STAVI-U-RED ($Q, 6$)



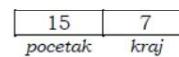
STAVI-U-RED ($Q, 15$)



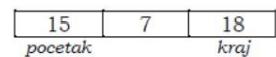
STAVI-U-RED ($Q, 7$)



IZVADI-IZ-REDA (Q)



STAVI-U-RED ($Q, 18$)



IZVADI-IZ-REDA (Q)



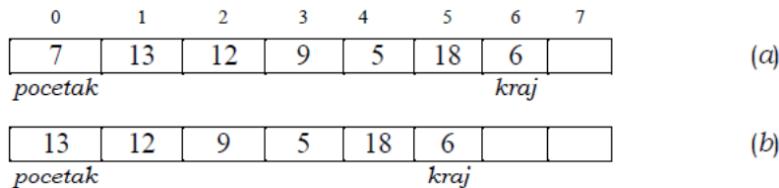
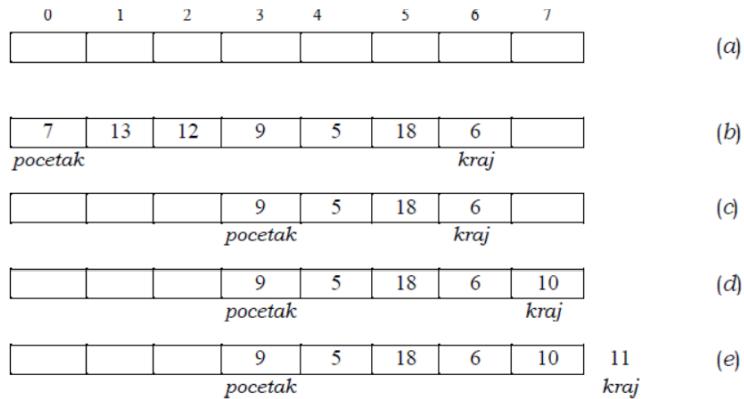
Tipične operacije u radu sa redom su:

- umetanje novih elemenata;
- izbacivanje prvog elementa u redu;
- provjera da li je red prazan;
- provjera da li je red pun;
- čitanje prvog elementa u redu bez njegovog brisanja;
- brisanje čitavog reda.

Implementacija reda pomoću nizova.

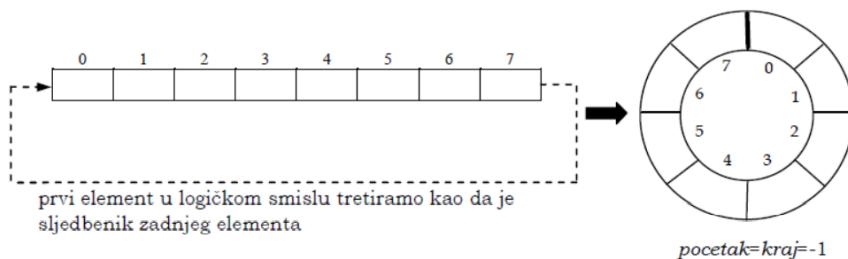
- Najčešće se koriste sekvencijalna i ulančana implementacija. Kod sekvencijalne implementacije koristimo nizove, u koje smještamo elemente reda, te dvije cijelobrojne oznake, koje predstavljaju pozicije čela i začelja.
- Konačna dužina niza, slično kao i kod steka, može prouzrokovati određene probleme

- Neka se za implementaciju reda koristi niz kapaciteta 8 elemenata (slika a). Počevši od praznog reda, pretpostavimo da se izvode operacije sljedećim redoslijedom: sedam umetanja elemenata 7, 13, 12, 9, 5, 18 i 6 (slika b); tri izbacivanja elemenata 7, 13 i 12 (slika c); i umetanje elementa 10 (slika d))
 - Pretpostavimo da se u ovom stanju reda pojavi operacija umetanja elementa 11. Oznaka začelja kraj se nalazi na zadnjoj poziciji niza (slika d), tako da nije moguće oznaku začelja pomjeriti iza gornje granice niza, da bi se umetnuo novi element, iako u nizu postoje tri prazna mesta na pozicijama s indeksima 0, 1 i 2 (slika e))
- Napomenimo da je čak moguća i situacija da je red potpuno ispraznjen operacijama izbacivanja, a u red se ne mogu umetnuti novi elementi, jer je oznaka začelja kraj prekoračila granicu niza.
- Jedno od mogućih rješenja je pomjeranje svih preostalih elemenata u nizu za jednu poziciju ulijevo, nakon svakog izbacivanja elementa iz reda, tako da bi oznaka čela pocetak uvijek pokazivala na prvu poziciju u nizu.
- Na primjer, na slici je prikazana situacija u kojoj se iz reda koji u početnom stanju ima 7 elemenata (slika a), nakon izbacivanja elementa 7 sa čela reda pomjeraju svi preostali elementi za jedno mjesto ulijevo (slika b).
- U ovom pristupu je zapravo oznaka za čelo reda početak nepotrebna, jer je ta oznaka uvijek jednaka indeksu prve pozicije)

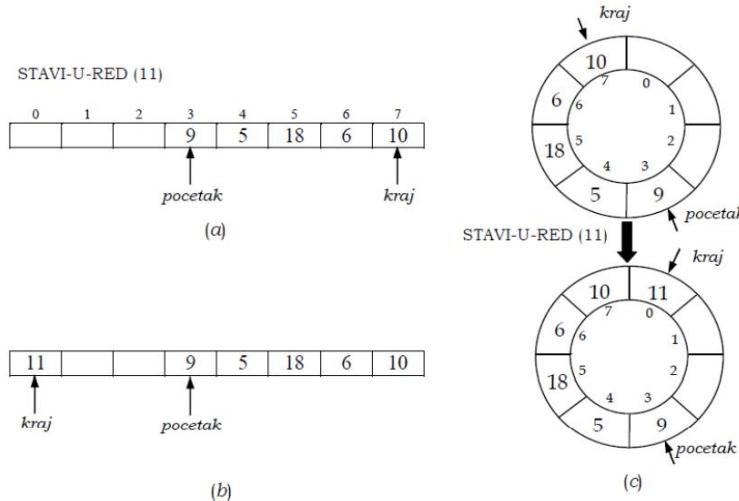


Implementacija reda cirkularnim nizom.

- Puno efikasnije rješenje za prethodno opisane situacije je implementacija reda pomoću tzv. cirkularnog niza.
- Kod cirkularnog niza se prvi element, u logičkom smislu, smatra sljedbenikom zadnjeg elementa, bez obzira što su ta dva elementa u fizičkom smislu odvojeni.
- Logička struktura cirkularnog niza, kod kojeg prvi element slijedi iza zadnjeg elementa, je prikazana na slici, za red Q koji ima dužinu jednaku 8.



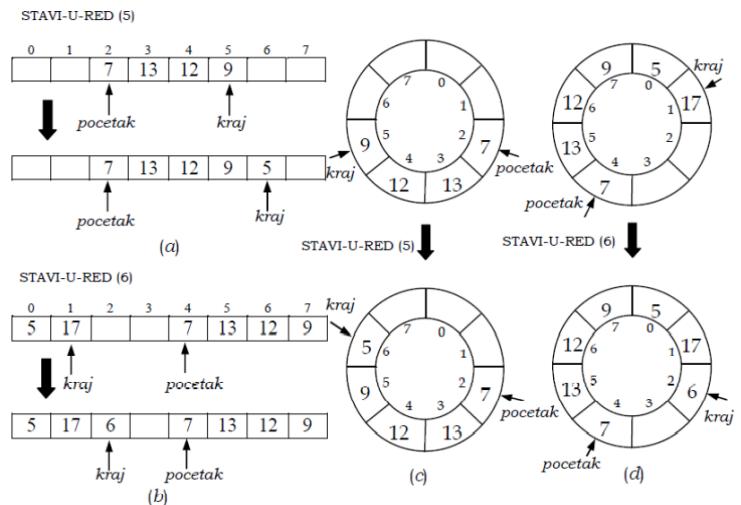
- Element Q[0], u logičkom smislu, slijedi iza zadnjeg elementa u nizu Q[7], iako su u fizičkoj implementaciji ti elementi razdvojeni.
- Operacija inicijalizacije reda Q maksimalne veličine, pored rezerviranja potrebnog memorijskog prostora za niz date veličine, također uključuje i postavljanje oznaka čela i začelja reda na neku vrijednost, koja indicira prazan red, kao na primjer pocetak = kraj = -1



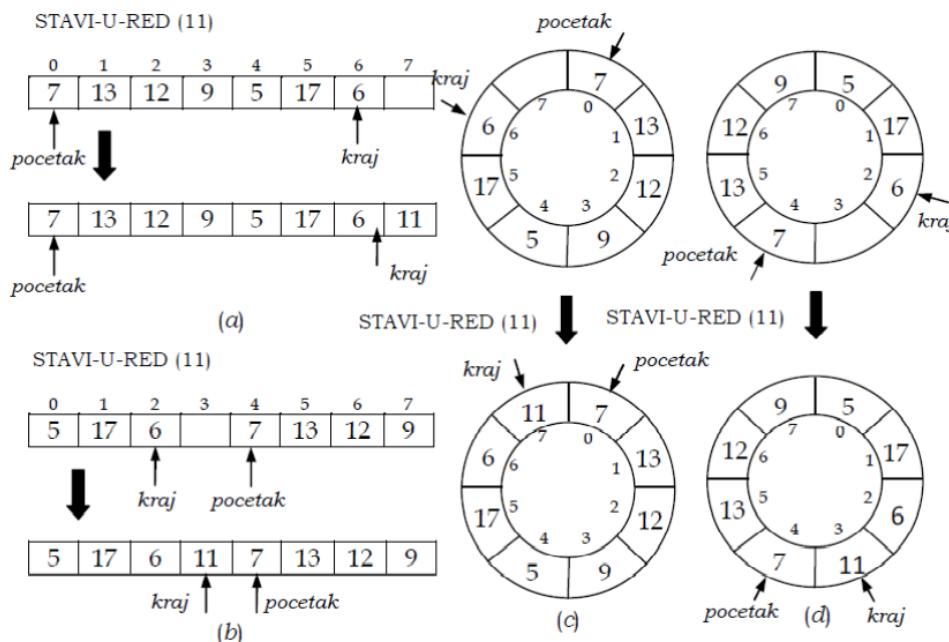
- Slika prikazuje red Q, koji u početnom stanju sadrži elemente: 9, 5, 18, 6 i 10.
- Na slici b je prikazano stanje u redu, nakon umetanja elementa 11.
- Prema tome, iako je zadnja pozicija u nizu bila popunjena, novi element 11 je umetnut na prvu poziciju u nizu, jer je ta pozicija bila slobodna.
- Način umetanja elementa 11 je također ilustriran i na slici c, ali uz cirkularni grafički prikaz niza. Bitno je istaknuti da pri umetanju elementa 11, nije bilo pomjeranja prisutnih elemenata u nizu, pa je ovakva organizacija mnogo efikasnija

Primjer umetanja elemenata u red sa cirkularnim nizom.

- U ovom primjeru, red u početnom stanju sadrži 4 elementa, od pozicije pocetak=2 do pozicije kraj=5 (slika a).
- U red se pozivom procedure STAVI-U-RED (Q, 5) umeće novi element 5. Oznaka kraj se povećava za 1 i poprima vrijednost kraj=6, te se na tu lokaciju upisuje novi element. U ovoj situaciji vrijedi kraj>pocetak.
- Već smo prethodno, na primjeru prikazanom na slici (prethodni slajd), vidjeli da je u stanju reda u kojem također vrijedi kraj>pocetak, nakon umetanja elementa 11, oznaka kraj poprimila vrijednost kraj=0 i niz je prešao u stanje u kojem vrijedi pocetak>kraj.



- Drugo tipično stanje pri umetanju novih elemenata karakterizira sljedeći odnos oznaka za čelo i začelje reda: $pocetak > kraj$.
- Ovo stanje je ilustrirano primjerom umetanja novog elementa na slici b.
- Red u početnom stanju sadrži 6 elemenata, od pozicije $pocetak=4$ do pozicije $kraj=1$.
- U red Q se pozivom procedure STAVI-U-RED(Q, 6) umeće novi element 6. Oznaka kraj se povećava za 1 i poprima vrijednost $kraj=2$, te se na tu lokaciju upisuje element 6.
- Prethodno opisane dvije tipične situacije su ilustrirane i uz cirkularni grafički prikaz niza na slikama c i d.
- Sljedeća tipična situacija koju je bitno razmotriti je situacija kada se umetanjem novog elementa zauzimaju sve dostupne pozicije u nizu, te red poprima status punog reda.
- Ovakva situacija je ilustrirana pomoću dva primjera na slici. U prvom primjeru (slika a), red je skoro popunjen i sadrži 7 elemenata, na pozicijama od čela reda $pocetak=0$ do začelja reda $kraj=6$.
- U red Q se pozivom procedure STAVI-U-RED(Q, 11) umeće novi element 11 i red postaje popunjen (slika a). Vrijednosti oznaka za čelo i začelje u ovom stanju popunjenosti su: $pocetak=0$ i $kraj=7$
- Provjera mora uključivati :
- $pocetak[Q] = 0$ and $kraj[Q] = \text{duzina}[Q]-1$



- U drugom primjeru (slika b), red u početnom stanju sadrži 7 elemenata, od čela reda $pocetak=4$ do začelja reda $kraj=2$, te prema tome ima samo jedno slobodno mjesto. I u ovom primjeru, pozivom procedure STAVI-U-RED (Q, 11) se umeće novi element 11, i red također postaje popunjen.
- Vrijednosti oznaka za čelo i začelje u ovom stanju popunjenosti su: $pocetak=4$ i $kraj=3$.
- Provjera mora uključivati :
- $pocetak[Q]=0$ and $kraj[Q]=\text{duzina}[Q]-1$
- or $pocetak[Q] = kraj[Q]+1$ ILI $pocetak[Q] = (\text{kraj}[Q]+1) \bmod \text{duzina}[Q]$

Primjer procedure STAVI-U-RED

- Funkcija JE-LI-PUN prikazana je na slici.
- Ako je ispunjen kriterij za identifikaciju punog reda, prethodno definiran izrazom, funkcija vraća logičku vrijednost true.
- U suprotnom, u redu postoje slobodne lokacije, pa funkcija vraća logičku vrijednost false.

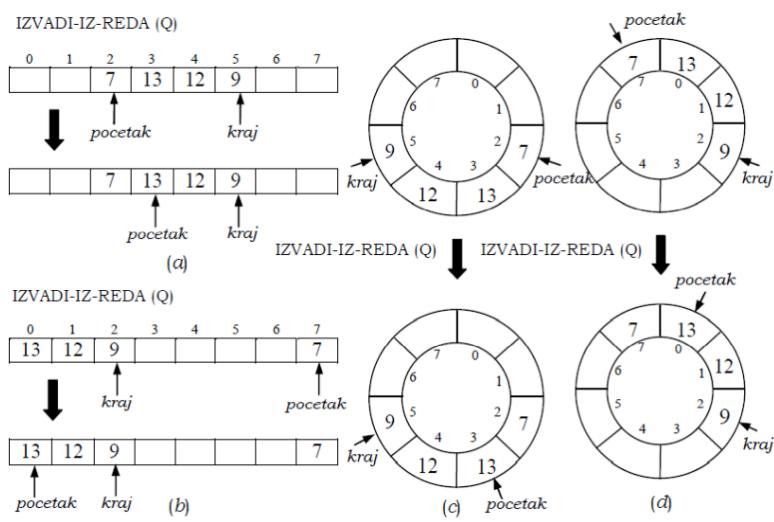
```
JE-LI-PUN (Q)
1 if (pocetak[Q] = (kraj[Q]+1) mod duzina[Q]) then
2   return true
3 else
4   return false
5 end_if
```

```
STAVI-U-RED (Q, x)
1 if (JE-LI-PUN(Q)) then
2   ERROR („Red je pun!“)
3 else
4   kraj[Q]  $\leftarrow$  (kraj[Q]+1) mod duzina[Q]
5   Q[kraj[Q]]  $\leftarrow$  x
6   if (pocetak[Q] = -1) then
7     pocetak[Q]  $\leftarrow$  0
8   end_if
9 end_if
```

- Operacija umetanja novog elementa x u red Q je opisana funkcijom STAVI-U-RED.
- Prvo se pozivom funkcije JE-LI-PUN(Q) provjerava da li je red Q pun (linija 1). Ako je red pun, novi element se ne može umetnuti, te se signalizira prekoračenje kapaciteta reda (linija 2).
- Nadalje, ako red nije pun, povećava se pokazivač kraj za 1, tako da pokazuje na prvu sljedeću slobodnu lokaciju u nizu, te se na tu lokaciju upisuje novi element x (linije 4 i 5).
- Na kraju, procedura STAVI-U-RED još provjerava da li se umetanje obavlja u prazan red, jer se u toj situaciji treba, pored oznake kraj, ažurirati i oznaka pocetak.
- Naime, ako je red bio prazan (pocetak=-1), onda se postavlja i oznaka čela reda pocetak na vrijednost 0 (linije 6-8).

Funkcija izbacivanja elemenata iz reda

- Operacija izbacivanja je implementirana tako da se prvo provjerava da li je red prazan. Ako red nije prazan, oznaka čela pocetak se povećava za 1, tako da pokazuje na sljedeći element, koji na taj način postaje prvi element reda. Na slici je ilustrirana operacija izbacivanja, kojom se iz reda koji sadrži 4 elementa, na pozicijama od pocetak=2 do kraj=5, izbacuje element 7, tako da se oznaka čela reda ažurira na vrijednost pocetak=3.
- Izbačeni element Q[2]=7 fizički ostaje u nizu, ali se nalazi izvan logičkog raspona reda Q[3..5].
- Na slici b je ilustriran još jedan mogući raspored elemenata u nizu, prije i nakon brisanja elementa iz reda, pri čemu je oznaka čela reda pocetak, nakon izbacivanja, poprimila vrijednost pocetak=0.



- Situacije prikazane na slikama a i b su također ilustrirane uz cirkularni grafički prikaz niza i na slikama c i d, respektivno

Funkcija JE-LI-PRAZAN.

- Ako cjelobrojna oznaka čela reda pocetak ima vrijednost -1, onda je red Q prazan, pa funkcija vraća logičku vrijednost true (linije 1-2).
- U suprotnom, cjelobrojna oznaka pocetak ima neku vrijednost u rasponu 0 do duzina[Q]-1, pa funkcija vraća logičku vrijednost false (linija 4).

```
JE-LI-PRAZAN (Q)
1 if (pocetak[Q] = -1) then
2   return true
3 else
4   return false
5 end_if
```

IZVADI-IZ-REDA

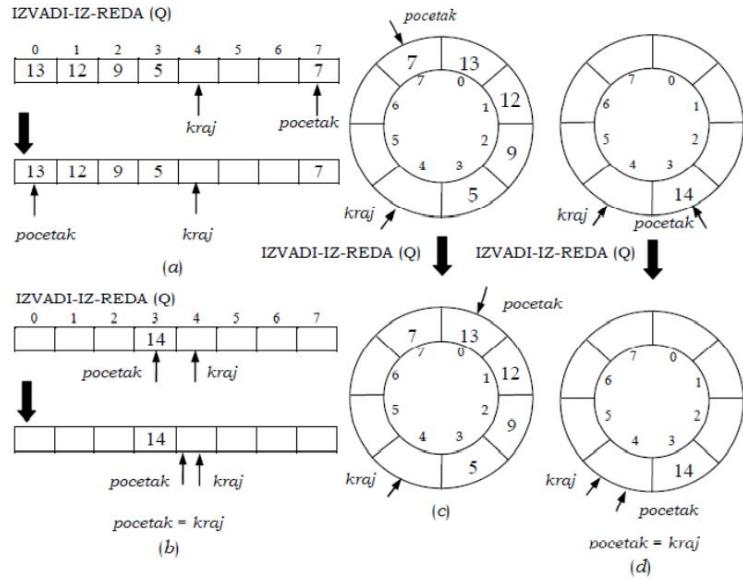
- Prvo se pozivom funkcije JE-LI-PRAZAN(Q) ispituje da li je red Q prazan. Ako je red prazan, operacija izbacivanja se ne može izvršiti, pa funkcija prijavljuje grešku kojom se signalizira da je red prazan (linije 1-2).
- Nadalje, ako red nije prazan, uzima se element x sa čela reda (linija 4), a zatim se ažurira oznaka pocetak i eventualno oznaka kraj.
- Naime, ako je prethodno red sadržavao samo jedan element, što se utvrđuje ispitivanjem uvjeta pocetak[Q] = kraj[Q], onda se ažuriraju i oznaka pocetak i oznaka kraj, tako da poprimaju vrijednost -1 (linije 5-6). S druge strane, ako je red prije operacije izbacivanja sadržavao više od jednog elementa, tada se ažurira samo oznaka pocetak, tako da joj se dodijeli vrijednost cirkularnog sljedbenika (linija 8). Na kraju, funkcija IZVADI-IZ-REDA vraća element x (linija 10)

```
IZVADI-IZ-REDA (Q)
1 if (JE-LI-PRAZAN (Q)) then
2   ERROR („Red je prazan!“)
3 else
4   x ← Q[pocetak[Q]]
5   if (pocetak[Q] = kraj[Q]) then
6     pocetak[Q] ← kraj[Q] ← -1
7   else
8     pocetak[Q] ← (pocetak[Q]+1) mod duzina[Q]
9   end_if
10  return x
11 end_if
```

Modificirana varijanta implementacije reda cirkularnim nizom

- Da bi se u određenoj mjeri pojednostavile a time učinile i nešto efikasnijim operacije umetanja i izbacivanja, moguće je organizaciju cirkularnog niza modificirati tako da se usvoji pravilo da oznaka kraj ne pokazuje na začelje reda, nego na jednu poziciju iza njega

- U prvom primjeru (slika a), red u početnom stanju sadrži 5 elemenata na pozicijama od pocetak=7 do kraj-1=3.
- Pozivom funkcije IZVADI-IZ-REDA(Q) se ažurira oznaka pocetak na vrijednost pocetak=0, te se element 7 u logičkom smislu izbacuje iz reda.
- U drugom primjeru (slika b), red u početnom stanju sadrži samo jedan element na poziciji pocetak=3. Oznaka kraj pokazuje na sljedeću praznu poziciju i ima vrijednost kraj=4. Pozivom funkcije IZVADI-IZ-REDA se ažurira oznaka pocetak na vrijednost pocetak=4, te se element 14 u logičkom smislu izbacuje iz reda, te red postaje prazan.
- Prema tome, možemo zaključiti da će logički uvjet, kojim ćemo ispitivati da li je red prazan, imati sljedeći oblik:
- $\text{pocetak}[Q] = \text{kraj}[Q]$.



Modificirana verzija operacija izbacivanja iz reda

- Modificirana verzija operacije kojom ispitujemo da li je red prazan je opisana funkcijom JE-LIPRAZAN-M(Q).
- Ako je ispunjen kriterij za identifikaciju praznog reda, prethodno definiran izrazom, funkcija vraća logičku vrijednost true (linije 1-2).
- U suprotnom, funkcija vraća logičku vrijednost false (linija 4).
- Prvo se pozivom funkcije JE-LI-PRAZAN-M (Q) provjerava da li je red Q prazan.
- Ako je red prazan, funkcija prijavljuje odgovarajuću grešku (linije 1-2). U suprotnom, u objekat x se upisuje element sa čela reda, povećava se oznaka čela reda za 1, te funkcija vraća element x koji se izbacuje (linije 4-6).
- Ova varijanta izbacivanja iz reda je nešto efikasnija u odnosu na prvu varijantu, jer nije potrebno dodatno provjeravati da li red izbacivanjem elementa ostaje prazan. Naime, u prvoj varijanti su oznake pocetak i kraj, nakon utvrđivanja da red ostaje prazan, trebale biti ponovno postavljene na -1. S druge strane, u ovoj drugoj varijanti je kriterij za identifikaciju praznog reda definiran odgovarajućim uvjetom (vidjeti izraz 6.5), pa nije potrebno oznake pocetak i kraj postavljati na vrijednost -1.

JE-LI-PRAZAN-M (Q)

```

1 if (JE-LI-PRAZAN-M ( $Q$ )) then
2   return true
3 else
4   return false
5 end_if

```

IZVADI-IZ-REDA-M (Q)

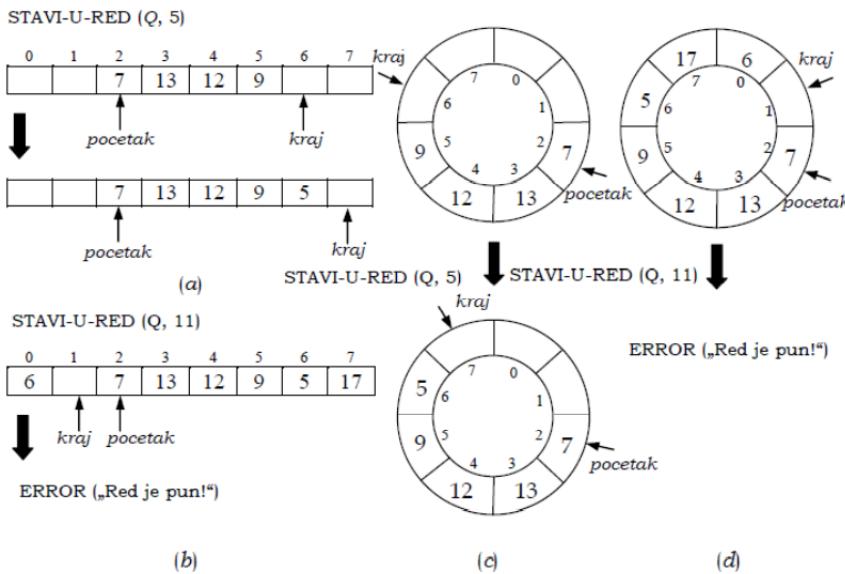
```

1 if (JE-LI-PRAZAN-M ( $Q$ )) then
2   ERROR („Red je prazan!“)
3 else
4    $x \leftarrow Q[\text{pocetak}[Q]]$ 
5    $\text{pocetak}[Q] \leftarrow (\text{pocetak}[Q]+1) \bmod \text{duzina}[Q]$ 
6   return  $x$ 
7 end_if

```

Modificirana verzija operacija umetanja u red

- U prvom primjeru (slika a), red u početnom stanju sadrži 4 elementa, od pozicije pocetak=2 do pozicije kraj-1=5.
- Pozivom procedure STAVI-U-RED(Q, 5) se prvo na poziciji kraj=6 upisuje novi element 5, a zatim se oznaka kraj ažurira da pokazuje na sljedeću praznu poziciju.



- Drugi primjer, prikazan na slici b, predstavlja situaciju u kojoj red u početnom stanju ima 7 elemenata na pozicijama od pocetak=2 do kraj=1.
- Prema tome, u nizu je preostala još jedna slobodna lokacija, upravo sa indeksom kraj=1.
- Razmotrimo sada stanje reda pri umetanju novog elementa na jedinu preostalu slobodnu poziciju. Pozivom procedure STAVI-U-RED (Q, 11) bi se element 11 trebao upisati na poziciju sa indeksom kraj=1, a oznaka kraj bi se trebala ažurirati na vrijednost kraj=2.
- To bi značilo da kriterij za identifikaciju punog reda ima sljedeći oblik:
- $pocetak[Q] = kraj[Q]$.
- Prema tome, iako je u nizu ostala slobodna još jedna pozicija s indeksom kraj=1, u ovoj modificiranoj verziji implementacije neće biti dozvoljeno umetanje novog elementa, jer bi tada red prešao u stanje kojem vrijedi $pocetak[Q]=kraj[Q]$, što bi značilo da bismo imali isti kriterij i za identifikaciju punog reda i za identifikaciju praznog reda.
- Na slici c je ilustrirana situacija sa slike a, uz grafički cirkularni prikaz niza. Također, koristeći grafički cirkularni prikaz na slici d je ilustrirano stanje punog reda sa slike b.
- Možemo zaključiti da će kriterij za identifikaciju punog reda, u modificiranoj verziji, uključivati ispitivanje da li je oznaka kraj prethodnik oznaci pocetak u smjeru kretanja kazaljke na satu (slika d).
- Odnosno, kriterij za identifikaciju punog reda, u modificiranoj varijanti, možemo zapisati na sljedeći način:
- $pocetak[Q] = (kraj[Q]+1) \bmod \text{duzina}[Q]$.

JE-LI-PUN-M

- Modificirana verzija operacije kojom ispitujemo da li je red pun je opisana funkcijom **JE-LI-PUN-M**.
- Ako je ispunjen kriterij za identifikaciju punog reda, prethodno definiranim izrazom, funkcija vraća logičku vrijednost true (linije 1-2).
- U suprotnom, funkcija vraća logičku vrijednost false (linija 4).

JE-LI-PUN-M (Q)

```
1 if (pocetak[Q] = (kraj[Q]+1) mod duzina[Q] ) then
2   return true
3 else
4   return false
5 end_if
```

STAVI-U-RED-M

- Jedna od razlika ove operacije umetanja, u odnosu na prvu varijantu, je u tome da se oznaka kraj povećava za 1 nakon upisivanja novog elementa u red Q na poziciju kraj (linija 5).
- Dodatno istaknimo da je i operacija umetanja nešto jednostavnija u odnosu na prvu varijantu, jer nije potrebno dodatno provjeravati da li se umetanje obavlja u prazan red.

STAVI-U-RED-M (Q, x)

```
1 if (JE-LI-PUN-M(Q)) then
2   ERROR („Red je pun!“)
3 else
4   Q[kraj[Q]] ← x
5   kraj[Q] ← (kraj[Q]+1) mod duzina[Q]
6 end_if
```

Implementacija reda povezanom listom

- Slično kao kod steka, red se također može implementirati koristeći povezane liste.
- Prvi čvor liste predstavlja čelo reda, dok zadnji element u listi predstavlja začelje reda.

Implementacija reda povezanom listom – umetanje elementa

STAVI-U-RED

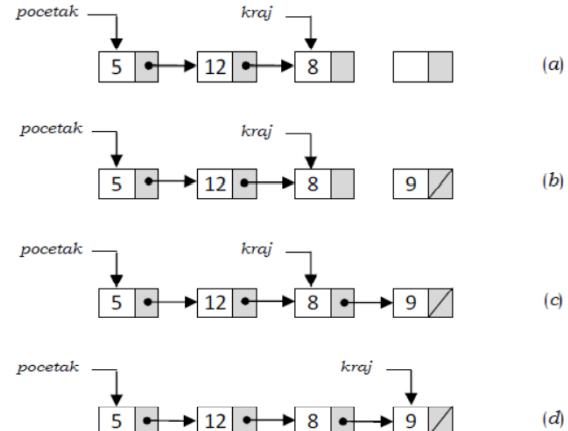
- Ovom operacijom se prvo alocira prostor za novi čvor (linija 1). Zatim se u informacioni dio novog čvora upiše sadržaj x (linija 2), dok se u pokazivački dio novog čvora upiše vrijednost NULL (linija 3).
- Ako je lista bila prazna (linija 4), tada se ažuriraju pokazivači pocetak i kraj tako da pokazuju na jedini čvor u listi (linija 5). Inače,
- Ako lista nije bila prazna, povezuju se zadnji čvor u listi na kojeg pokazuje pokazivač kraj i novi čvor na kojeg pokazuje pokazivač p (linija 7).
- Pošto se novi čvor umeće na kraj liste, potrebno je na kraju ažurirati pokazivač kraj, tako da pokazuje na novi čvor (linija 8).

STAVI-U-RED (Q, x)

```

1   $p \leftarrow \text{GETNODE}()$ 
2   $\text{info}(p) \leftarrow x$ 
3   $\text{sljedeci}(p) \leftarrow \text{NIL}$ 
4  if ( $\text{pocetak}[Q] = \text{NIL}$ ) then
5     $\text{pocetak}[Q] \leftarrow \text{kraj}[Q] \leftarrow p$ 
6  else
7     $\text{sljedeci}[\text{kraj}[Q]] \leftarrow p$ 
8     $\text{kraj}[Q] \leftarrow \text{sljedeci}[\text{kraj}[Q]]$ 
9  end_if

```



- Operacija umetanja novog čvora u red implementiran povezanom listom je ilustrirana na slici umetanjem elementa x=9.
- Prvo se alocira memorijski prostor za novi čvor (slika a). Zatim se u komponentu novog čvora info upisuje element x=9, dok se u komponentu sljedeci upisuje vrijednost NULL, jer se novi čvor dodaje na kraj liste (slika b).
- U sljedećem koraku, novi čvor se povezuje sa ostatkom liste, tako što se u komponentu sljedeci zadnjeg čvora upisuje pokazivač na novi čvor (slika c).
- U zadnjem koraku vrijednost pokazivača kraj se ažurira, tako da mu se dodjeljuje adresa alociranog prostora za novi čvor (slika d).

Implementacija reda povezanom listom – izbacivanje elementa

IZVADI-IZ-REDA

- Operacija pored uklanjanja prvog čvora na koji pokazuje pokazivač pocetak, također vraća informacioni sadržaj x tog čvora (linija 12).
- Isto tako, operacija ažurira pokazivač pocetak, tako da pokazuje na sljedbenika prvog čvora (linija 9).
- Ako je u redu bio samo jedan element, tada se njegovim izbacivanjem generira prazna lista, pa pokazivač pocetak i kraj poprimaju vrijednost NIL (linije 6-7).

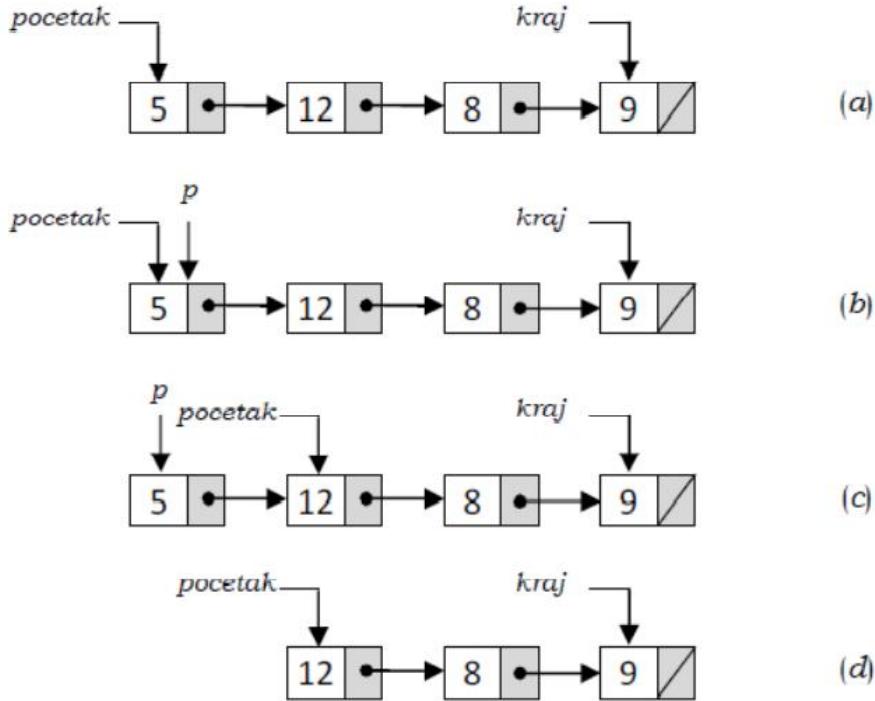
IZVADI-IZ-REDA (Q)

```

1  if ( $\text{pocetak}[Q] = \text{NIL}$ ) then
2    ERROR („Red je prazan!“)
3  end_if
4   $x \leftarrow \text{info}[\text{pocetak}[Q]]$ 
5   $p \leftarrow \text{pocetak}[Q]$ 
6  if ( $\text{pocetak}[Q] = \text{kraj}[Q]$ ) then
7     $\text{pocetak}[Q] \leftarrow \text{kraj}[Q] \leftarrow \text{NIL}$ 
8  else
9     $\text{pocetak}[Q] \leftarrow \text{sljedeci}[\text{pocetak}[Q]]$ 
10 end_if
11  $\text{FREENODE}(p)$ 
12 return  $x$ 

```

- U prvom koraku se u privremenim objekatima pohranjuje informacioni sadržaj čvora na čelu reda ($x=5$), te se pokazivač p dodjeljuje vrijednost pokazivača pocetak, koji pokazuje na prvi čvor liste (slika b).
- U sljedećem koraku se pokazivač pocetak ažurira i dodjeljuje mu se adresa drugog čvora u redu (slika c), čime se prvi čvor, u logičkom smislu, izbacuje iz reda.
- Na kraju se i fizički uklanja prvi čvor i vraća se sadržaj prethodno pohranjen u privremenom objektu x (slika 6.12d).



(Predavanje VII)

Algoritmi pretraživanja

- Pod pretraživanjem podrazumijevamo utvrđivanje da li se zadata tražena vrijednost pojavljuje u nekom skupu vrijednosti.
- U ovom poglavlju će biti opisani osnovni algoritmi pretraživanja, a to su algoritmi sekvenčnog i binarnog pretraživanja.
- Složeniji algoritmi se temelje na ovim algoritmima

Sekvencijalno pretraživanje

- Sekvencijalno pretraživanje je najjednostavniji algoritam pretraživanja.
- Algoritmi pretraživanja podrazumijevaju pronalaženje neke zadate vrijednosti.
- Niz se pretražuje sve dok se zadata tražena vrijednost ne pronađe, ili dok algoritam kao svoj izlaz ne odredi da tražena vrijednost nije u nizu
- Na primjer, pretpostavimo da su u nizu jednostavnii podaci cijelobrojnog tipa, onda se tražena vrijednost uzastopno uspoređuje sa elementima iz niza, sve dok se ne pronađe podudaranje tih vrijednosti, ili dok se ne ispitaju svi elementi u nizu.
- Kod kompleksnijih tipova podataka obično se neki atribut definira kao ključni atribut. Tada se vrši poređenje tražene vrijednosti samo sa vrijednostima ključnog atributa, dok se ostali atributi kompleksnijih tipova za potrebe pretraživanja ignoriraju.
- Na primjer, neki niz može sadržavati podatke o radnicima neke kompanije. Radnici mogu biti prikazani pomoću složenog tipa koji uključuje atribute kao što su: prezime radnika, ime radnika, adresa, telefonski broj, radno mjesto, itd.
- Nadalje, za potrebe pretraživanja može se definirati i šifra radnika koja je jedinstvena za svakog radnika u nizu. Šifra radnika se može koristiti kao ključni atribut, pa se kod pretraživanja kao tražena vrijednost tada koristi neka konkretna šifra radnika
- Sekvencijalni algoritam pretraživanja ostvaruje pronalaženjem traženog ključa tako da se uspoređuje jedna po jedna vrijednost ključa u nizu sa traženim ključem, počevši od prvog elementa u nizu, sve dok se traženi ključ ne pronađe ili dok se ne dođe do kraja niza.
- Algoritam kao znak uspješnog pretraživanja može vratiti indeks pronađenog elementa u nizu, ili neku predefinisanu vrijednost, ako je pretraživanje bilo neuspješno.

SEKVENC-PRETRAZI (K, k)

```
1   $i \leftarrow 0$ 
2  while ( $i \leq n-1$ ) do
3      if ( $K[i] = k$ ) then
4          return  $i$ 
5      else
6           $i \leftarrow i+1$ 
7      end_if
8  end_while
9  return -1
```

SEKVENC-PRETRAZI-M (K, k)

```
1   $K[n] \leftarrow k$ 
2   $i \leftarrow 0$ 
3  while ( $K[i] \neq k$ ) do
4       $i \leftarrow i+1$ 
5  end_while
6  if ( $i \neq n$ ) then
7      return  $i$ 
8  end_if
9  return -1
```

- Prepostavimo da su ključevi organizirani u niz $K[0..n-1]$, koji sadrži n elemenata.
- Algoritam sekvencijalnog pretraživanja je opisan funkcijom SEKVENCPRETRAZI koja je prikazana na slici.
- Argumenti funkcije su niz K i traženi ključ k . Funkcija vraća poziciju traženog ključa u nizu ako je pretraživanje završilo uspješno (pronalaženjem ključa), ili vrijednost -1 ako traženi ključ nije prisutan u nizu K .
- Ako niz K sadrži više traženih ključeva, funkcija će vratiti najmanji indeks lokacije sa traženim ključem k .
- Algoritam sekvencijalnog pretraživanja SEKVENCPRETRAZI se može do određene mjere učiniti efikasnijim, ako se on modificira na način kako je to opisano na slici u obliku funkcije SEKVENC-PRETRAZI-M.
- Na početku algoritamskog procesa se na n -tu poziciju u nizu kao novi element dodaje traženi ključ (linija 1).
- Na taj način se postiže da se u nizu mora pronaći traženi ključ na nekoj od pozicija $0..n$. Ako je traženi ključ pronađen na zadnjoj (n -toj) poziciji u nizu ($i=n$), to zapravo predstavlja neuspješno pretraživanje, pa funkcija vraća -1 (linija 9).
- S druge strane, ako je traženi ključ pronađen na nekoj drugoj poziciji ($i \neq n$), to znači da je on prisutan u nizu, te funkcija vraća indeks pozicije u nizu na kojoj je ključ pronađen (linije 6-8).
- Prednost modificiranog sekvencijalnog algoritma pretraživanja ogleda se u činjenici da nije potrebno poslije svake usporedbe provjeravati da li je varijabla i izašla iz raspona $0..n-1$, kao kod običnog algoritma sekvencijalnog pretraživanja.
- Na taj način se dvostruko smanjuje broj operacija poređenja, što modificirani algoritam čini u određenoj mjeri efikasnijim.

Efikasnost sekvencijalnog pretraživanja.

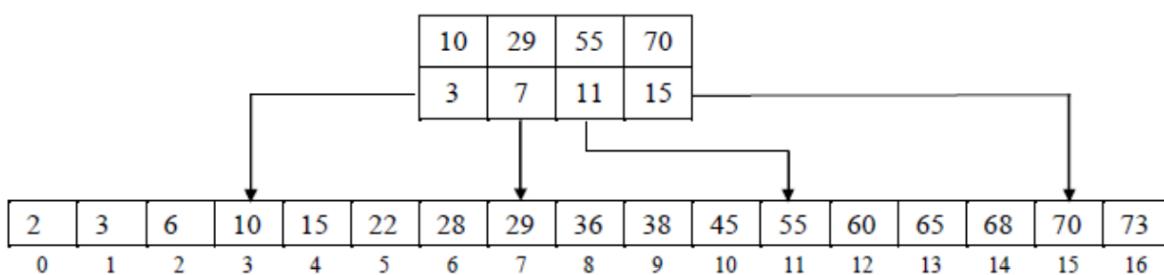
- Tri mjere za efikasnost sekvencijalnog pretraživanja:
 - najbolji,
 - najgori i
 - prosječni slučaj.
- Najbolji slučaj je situacija u kojoj je traženi ključ pozicioniran na prvom mjestu u nizu, jer je za njegovo pronalaženje potrebna samo jedna operacija usporedbe, bez obzira na dužinu niza n .
- S druge strane, najgori slučaj je situacija u kojoj je traženi ključ pozicioniran na zadnjem mjestu u nizu ili ako je pretraživanje neuspješno, tj. ako traženi element nije uopće prisutan u nizu.
- Tada je potrebno vrijednost traženog ključa usporediti sa vrijednostima svih ključeva u nizu, što znači da je potrebno n operacija usporedbi.
- Prosječan slučaj se dobija izračunavanjem prosječnog broja potrebnih operacija usporedbi tako da se razmotre sve moguće situacije pretraživanja, uzimajući u obzir sve moguće klase ulaza.
- Ako prepostavimo da su svi ključevi jednakovjedno vjerojatni kao traženi ključevi, lako intuitivno možemo zaključiti da će u prosjeku biti potrebno pretražiti oko pola niza.
- To znači da možemo procijeniti da će biti potrebno oko $n/2$ operacija usporedbi. Ili, u cilju formalnijeg i preciznijeg razmatranja prosječnog slučaja kod sekvencijalnog pretraživanja, možemo reći da broj usporedbi kod uspješnog pretraživanja ovisi o poziciji ključa u listi.
- Uz pretpostavku da je traženi ključ na nekoj i -toj poziciji, tada je kod uspješnog pretraživanja potrebno i operacija usporedbi. Ako još prepostavimo da su svi ključevi jednakovjedno vjerojatni, onda je pri uspješnom pretraživanju ukupan broj operacija poređenja $T(n)$.

$$T(n) = \frac{1}{n} \sum_{i=1}^n i = \frac{1 + 2 + \dots + n}{n} = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Imajući u vidu da je stupanj rasta broja ostalih operacija koje se izvršavaju u sklopu while petlje jednak stupnju rasta operacija usporedbi, možemo zaključiti da je vremenska složenost algoritma sekvencijalnog pretraživanja $O(n)$, što nadalje znači da efikasnost ovog algoritma u prosječnom slučaju nije dobra, pa je ovaj algoritam prihvatljiv samo za male vrijednosti n

Upotreba pomoćnih indeksnih struktura

- Efikasnost sekvencijalnog pretraživanja se može povećati kod sortiranih nizova. Jedan od načina je korištenje pomoćne indeksne strukture. Primjer povećanja efikasnosti pretraživanja sortiranog niza primjenom pomoćne indeksne strukture je ilustriran na slici.



- Pomoćna indeksna struktura sadrži određeni broj indeksa pozicije iz niza i pripadajuće vrijednosti ključeva. Prilikom pretraživanja prvo se pretražuje indeksna struktura, da bi se pronašle dvije susjedne vrijednosti od kojih je jedna manja, a druga veća od traženog ključa. Nakon toga, pretraživanje se nastavlja samo u onom dijelu niza koji je definiran donjom i gornjom granicom iz indeksne strukture ili se odmah pronađe ključ u indeksnoj strukturi.
- Ako, na primjer, prepostavimo da je traženi ključ 38, algoritam će u indeksnoj strukturi identificirati donju granicu d=29 i gornju granicu g=55, odnosno algoritam će suziti pretraživanje niza na pozicije u rasponu 8..10. Dakle, potrebne su tri operacije poređenja da se utvrdi raspon pozicija u nizu 8..10, te dodatne dvije operacije poređenja da se pronađe traženi ključ 38.
- Na taj način je potrebno ukupno 4 operacije usporedbi, dok bi za sekvencijalno pretraživanje bez korištenja indeksne strukture bilo potrebno 10 operacija usporedbi.

Upotreba indeksnih struktura sa dva i više nivoa

- Ako nizovi imaju izuzetno veliki broj elemenata, onda se radi dodatnog povećanja efikasnosti mogu eventualno koristiti i indeksne strukture na dva hijerarhijska nivoa.
- Tada se indeksna struktura na višem hijerarhijskom nivou koristi da indeksira sadržaj indeksne strukture na nižem hijerarhijskom nivou, dok indeksna struktura na nižem nivou indeksira sadržaj elemenata niza

Poboljšanje efikasnosti pretraživanja preuređivanjem niza

- Spomenimo još da se efikasnost sekvencijalnog pretraživanja može poboljšati i u slučajevima kada su vjerojatnosti traženja pojedinih ključeva nejednake i unaprijed poznate
- Tada se ključevi u nizu mogu rasporediti tako da se ključevi sa većim vjerojatnoćama pretraživanja stavlju na pozicije bliže početku niza, dok se ključevi sa manjim vjerojatnoćama pretraživanja stavlju na pozicije bliže kraju niza
- Ako vjerojatnoće nisu unaprijed poznate, jedna od mogućnosti bi mogla biti mjerjenje frekvencije traženja pojedinih ključeva da bi se niz mogao dinamički preuređivati u skladu s izmijerenim frekvencijama. Međutim, brojanje frekvencija i dinamičko preuređivanje niza zahtijeva povećanje potrebnih operacija, što prethodno opisani pristup čini neprikladnim
- Jedan od mogućih pristupa bi mogao biti da pronađeni ključ zamijeni poziciju sa ključem koji je pozicioniran neposredno ispred njega.
- Kod ovog pristupa efikasnost je dobra i pri preuređivanju povezanih listi, kao i pri preuređivanju nizova.
- Drugi mogući pristup bi mogao biti da se pronađeni ključ odmah pozicionira na prvo mjesto, a svi ključevi koji su prije bili pozicionirani ispred pronađenog ključa bi se pomjerili za jedno mjesto ka kraju niza. Važno je istaknuti da je ovaj pristup efikasnije primijeniti na preuređivanju povezanih listi, nego na preuređivanju nizova

Binarno pretraživanje - uvod

- Ključni nedostatak sekvencijalnog pretraživanja zasniva se činjenici da se pri svakoj neuspješnoj operaciji poređenja odbacuje samo jedan element niza koji se pretražuje.
- Ovaj nedostatak je čak prisutan i kod nizova koji su sortirani po ključevima, iako kod njih informaciju o usporedbi traženog ključa i ključa u nizu pretraživanja možemo puno bolje iskoristiti,
- Ako se traženi ključ uspoređuje sa nekim ključem na poziciji i u nizu pretraživanja koji je sortiran u rastućem poretku, te ako se utvrdi da je traženi ključ veći od ključa na poziciji i, onda lako zaključujemo da je traženi ključ veći i od svih ostalih ključeva na pozicijama 1 do i-1.
- To znači, da se svi ključevi na pozicijama 1 do i-1 mogu odbaciti u nastavku algoritma pretraživanja
- Najefikasnije je vršiti poređenje na polovini intervala niza pretraživanja, te nakon odbacivanja pola elemenata niza ponovno vršiti polovljenje preostalog niza, sve dok se ne pronađe traženi ključ ili dok se ne utvrdi da traženi ključ nije prisutan u nizu.
- Ovakav pristup se naziva binarno pretraživanje
- Uvedimo oznake:
 - k - traženi ključ;
 - K - sortirani niz pretraživanja (u rastućem poretku);
 - vrh - zadnja pozicija u preostalom dijelu niza nakon polovljenja;
 - dno - prva pozicija u preostalom dijelu niza nakon polovljenja;
 - srednji - srednja pozicija u nizu
- Varijabla srednji predstavlja indeks središnjeg ključa u nizu ($K[srednji]$), koji zapravo dijeli niz na dvije polovine: lijevu polovinu koja se sastoji od elemenata sa indeksima isrednji.
- Preostali dio niza K, koji se dobija polovljenjem niza iz prethodne iteracije, je definiran cjelobrojnim varijablama dno i vrh
- U svakoj iteraciji algoritma se izračunava indeks središnjeg ključa $srednji=(vrh+dno)/2$, te se provjerava da li je središnji ključ $K[srednji]$ jednak ključu k kojeg tražimo

- Naravno, ova provjera najčešće pokazuje da središnji ključ nije traženi ključ, posebno u početnim iteracijama, pa se algoritam nastavlja tako da se na temelju usporedbe središnjeg ključa $K[\text{srednji}]$ i traženog ključa k odbacuje ili lijeva ili desna polovina niza
- Ako je ispunjeno $K[\text{srednji}] > k$, onda se odbacuje desna polovina niza. Inače, odbacuje se lijeva polovina niza. Nadalje, ako se odbacuje desna polovina niza, onda se ažurira varijabla vrh, tako da joj se pridruži vrijednost $\text{vrh} = \text{srednji} - 1$. S druge pak strane, ako se odbacuje lijeva polovina niza, onda se ažurira vrijednost varijable dno, tako da joj se pridruži vrijednost $\text{dno} = \text{srednji} + 1$

$k=22$																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	6	10	15	22	28	29	36	38	45	55	60	65	68	70	73

$\text{srednji} = 8, K[8] = 36, K[8] > k \Rightarrow \text{vrh} = 7$

0	1	2	3	4	5	6	7
2	3	6	10	15	22	28	29

(b)

$\text{srednji} = 4, K[4] = 15, K[4] < k \Rightarrow \text{dno} = 5$

5	6	7
22	28	29

(c)

$\text{srednji} = 6, K[6] = 28, K[6] > k \Rightarrow \text{vrh} = 5$

5
22

(d)

- Na slici a je prikazan niz nad kojim se obavlja binarno pretraživanje, pri čemu je traženi ključ $k=22$, dok su početne vrijednosti varijabli $\text{dno}=0$ i $\text{vrh}=16$, jer je u prvoj iteraciji preostali niz jednak zapravo početnom nizu.
- U prvoj iteraciji se izračunava indeks središnjeg elementa $\text{srednji}=8$, te se uspoređuje vrijednost elementa $K[8]=36$ sa traženim ključem $k=22$. Pošto je $36 > 22$, to znači da se treba odbaciti desna polovina niza, odnosno potrebno je ažurirati varijablu $\text{vrh} = \text{srednji} - 1 = 7$.
- Dakle, preostali dio niza u sljedećoj iteraciji će biti $K[0..7]$, jer je $\text{dno}=0$ i $\text{vrh}=7$.

$k=22$																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	3	6	10	15	22	28	29	36	38	45	55	60	65	68	70	73

$\text{srednji} = 8, K[8] = 36, K[8] > k \Rightarrow \text{vrh} = 7$

0	1	2	3	4	5	6	7
2	3	6	10	15	22	28	29

dno = 0; vrh = 7
dno + vrh / 2 = 3
k[3]=10 < 22
dno = srednji + 1 = 4

(b)

$\text{srednji} = 4, K[4] = 15, K[4] < k \Rightarrow \text{dno} = 5$

4	5	6	7
15	22	28	29

dno = 4; vrh = 7
dno + vrh / 2 = 5
k[5]=22 kraj

(c)

$\text{srednji} = 6, K[6] = 28, K[6] > k \Rightarrow \text{vrh} = 5$

5
22

(d)

- Preostali dio niza u ovoj iteraciji je prikazan na slici b. Ponovno se izračunava indeks središnjeg elementa srednji=3. Sada je središnji element K[3]=10 manji od traženog ključa k=22, pa je potrebno odbaciti lijevu polovicu niza sa indeksima lokacija i <= 3.
- Taj dio niza se odbacuje tako što se ažurira varijabla dno=srednji+1=4. Prema tome, preostali dio niza će biti K[4..7], jer je dno=4 i vrh=7.
- Na slici je prikazana funkcija BINPRETRAZI koja opisuje algoritam binarnog pretraživanja, koji kao ulaz uzima niz ključeva K i traženi ključ k, a kao izlaz daje indeks pozicije u nizu na kojoj je ključ k pronađen.
- Ako traženi ključ k nije pronađen, funkcija vraća -1

BIN-PRETRAZI (K, k)

```

1  dno ← 0
2  vrh ← duzina[K]-1
3  while (vrh ≥ dno) do
4      srednji ← (dno+vrh)/2
5      if (K[srednji] = k) then
6          return srednji
7      else if (K[srednji] > k) then
8          vrh ← srednji-1
9      else
10         dno ← srednji+1
11     end_if
12 end_while
13 return -1

```

BIN-PRETRAZI-M (K, k)

```

1  dno ← 0
2  vrh ← duzina[K]-1
3  while (vrh ≥ dno) do
4      srednji ← (dno+vrh)/2
5      if (K[srednji] < k) then
6          dno ← srednji+1
7      else
8          vrh ← srednji
9      end_if
10 end_while
11 if (vrh = -1) then
12     return -1
13 end_if
14 if (K[vrh] = k) then
15     return vrh
16 else
17     return -1
18 end_if

```

- Moguće je oblikovati i nešto drugačiji algoritam binarnog pretraživanja kod kojeg se u svakoj iteraciji, pri određivanju srednjeg elementa i polovljena intervala, ne provjerava da li je traženi ključ pronađen.
- Tek na samom kraju izvršavanja algoritma, kada u nizu ostane samo jedan element, provjerava se da li je taj element traženi ključ. Ova druga varijanta binarnog pretraživanja je prikazana na slici u obliku funkcije BIN-PRETRAZI-M
- Iako funkcija BIN-PRETRAZI-M na prvi pogled izgleda manje efikasna u odnosu na funkciju BIN-PRETRAZI, ipak u većini slučajeva je ustvari suprotno.
- Naime, funkcija BINPRETRAZI u svakoj iteraciji polovljena intervala ima dva poređenja, za razliku od funkcije BIN-PRETRAZI-M, kod koje se u svakoj iteraciji polovljena intervala obavlja jedna operacija usporedbe, te dodatna usporedba na kraju, kojom se provjerava da li je preostali element traženi ključ.
- Međutim, kod nizova sa većim brojem elemenata u većini slučajeva pretraživanja traženi ključ neće biti pronađen sve dok se niz polovljenjem ne reducira na svega nekoliko elemenata, a često i na samo jedan element.
- To znači, da će funkcija BINPRETRAZI u većini slučajeva izvršiti operacije usporedbi koje neće pronaći traženi ključ, što u određenoj mjeri ovu funkciju čini zapravo neefikasnijom u odnosu na funkciju BINPRETRAZI-M.

Efikasnost binarnog pretraživanja:

- Analizu efikasnosti binarnog pretraživanja možemo pojednostaviti tako da prepostavimo da nizovi imaju dužinu jednaku potencijama broja 2, tj. da nizovi imaju dužine 2, 4, 8, 16, 32, itd.
- Nadalje, procjenu potrebnih operacija usporedbi ćemo napraviti za drugu varijantu binarnog pretraživanja, koja je opisana funkcijom BIN-PRETRAZI-M. Na primjer, za niz dužine 8 će biti potrebna 3 polovljenja intervala, što znači da će biti potrebno ukupno 4 operacije usporedbi (3 operacije u koracima polovljenja i jedna završna operacija usporedbe).
- Za dužine nizova koje nisu jednake nijednoj potenciji broja 2, možemo procijeniti da će broj operacija usporedbi biti vrlo blizu broju operacija usporedbi za onaj niz čija je dužina najbliža dužini dotičnog niza i jednaka je potenciji broja 2.
- Na primjer, za niz dužine 1000, možemo procijeniti da će biti potrebno 10 operacija usporedbi da se niz polovi, sve dok u njemu ne ostane samo jedan element, te još jedna dodatna operacija usporedbe da se utvrdi da li je jedini preostali element traženi ključ
- Dakle, možemo procijeniti da će biti potrebno 11 operacija usporedbi da bi se pronašao neki ključ u nizu dužine 1000
- Općenito, za niz koji ima dužinu n jednaku $n=2^i$, možemo zaključiti da će biti potrebna $i+1$ operacija usporedbi. Odnosno, možemo pisati da je ukupan broj operacija usporedbi

$$T(n): T(n) = \log_2 n + 1$$

- Isto tako, umjesto analize kojom dobijamo broj potrebnih koraka usporedbe, možemo za određivanje vremenske složenosti binarnog pretraživanja procijeniti broj potrebnih koraka polovljenja intervala u ovisnosti od dužine n. Budući da se interval pretraživanja u svakom sljedećem koraku polovi, možemo pisati da poslije nekog i -tog koraka sljedeći izraz daje maksimalan broj elemenata koji su preostali u intervalu pretraživanja

$$\text{broj preostalih elemenata} = \frac{n}{2^i}$$

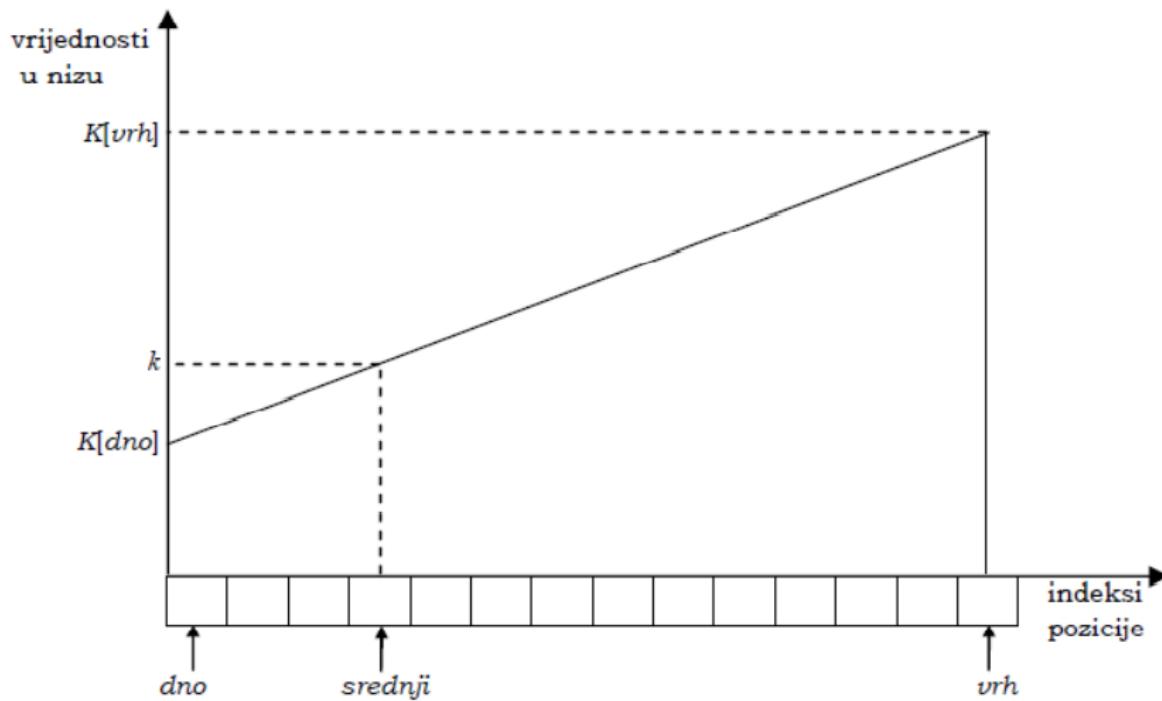
- Za obje prethodno opisane varijante binarnog pretraživanja možemo reći da algoritam sigurno završava kada preostane samo jedan element u intervalu pretraživanja.
- Dakle, nakon izjednačavanja izraza 8.2 sa 1, za maksimalan ukupan broj iteracija dobijamo

$$i = \log_2 n$$

- Imajući u vidu da je stupanj rasta broja ostalih operacija (operacije dodjele, aritmetičke operacije) jednak stupnju rasta operacija usporedbi, zaključujemo da je vremenska složenost binarnog pretraživanja reda $O(\log n)$.

Interpolacijsko pretraživanje - uvod

- Ako su vrijednosti ključeva u sortiranom nizu ravnomjerno raspoređene, onda se efikasnost algoritma binarnog pretraživanja može dodatno poboljšati korištenjem tzv. interpolacijskog pretraživanja
- Naime, umjesto da se u svakom koraku traženi ključ poredi sa elementom koji je pozicioniran na sredini intervala na koji je usmjeren pretraživanje, opravdano je traženi ključ porebiti sa elementom koji je bliži vrijednosti traženog ključa, koji se u općem slučaju ne nalazi na sredini intervala.
- Na primjer, ako je vrijednost traženog ključa k bliža vrijednosti ključa na donjoj granici intervala ($K[dno]$) od vrijednosti ključa na gornjoj granici intervala ($K[vrh]$), tada je efikasnije pretraživanje traženog ključa nastaviti u blizini donje granice intervala
- Prema tome, algoritam binarnog pretraživanja možemo modificirati tako da se pozicija polovljenja intervala ne određuje kao sredina intervala, nego se ta pozicija interpolira na način kako je to ilustrirano na slici



- Iz sličnosti truoglova se može napisati

$$\frac{srednji-dno}{k-K[dno]} = \frac{vrh-dno}{K[vrh]-K[dno]}$$

- Slijedi

$$srednji = dno + \frac{(k - K[dno])vrh - dno}{K[vrh] - K[dno]}$$

- Ako je traženi ključ jednako udaljen od vrijednosti ključeva na obje granice intervala, algoritam interpolacijskog pretraživanja će preploviti interval na isti način kao i obično binarno pretraživanje.
- Algoritam interpolacijskog pretraživanja je isti kao i prethodno opisane varijante binarnog pretraživanja u obliku funkcija BINPRETRAZI i BINPRETRAZI-M, s jedinom razlikom da se pozicija središnjeg elementa srednji ne određuje kao što je to opisano naredbom u liniji 4
- Istaknimo još jednom da je interpolacijsko pretraživanje efikasnije u odnosu na binarno pretraživanje samo ako su vrijednosti ključeva u nizu ravnomjerno raspoređene.
- U suprotnom, efikasnost interpolacijskog pretraživanja je bitno smanjena. Isto tako, potrebno je uzeti u obzir da obično binarno pretraživanje ima jednostavniju cjelobrojnu aritmetiku, koja uključuje samo indeks pozicija, dok interpolacijsko pretraživanje ima nešto složeniju aritmetiku, koja pored indeksa pozicija uključuje i vrijednosti ključeva, što u određenoj mjeri smanjuje vremensku efikasnost dotičnog algoritma