

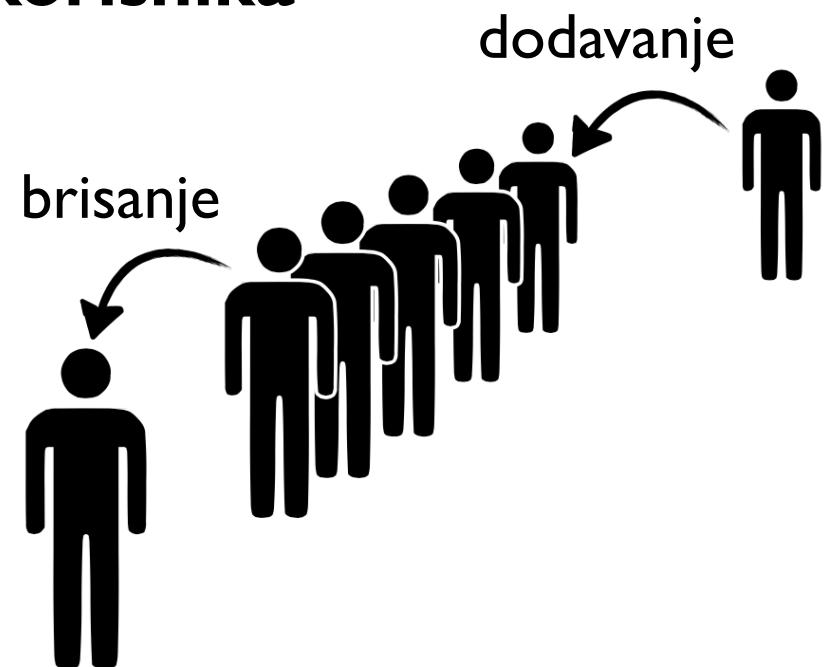
Red

Red

Red (engl. *queue*) je apstraktni tip podataka koji predstavlja kontejner u koji se **elementi dodaju ili iz koga se elementi brišu** po principu “**prvi stigao, prvi izašao**” (engl. *first in, first out - FIFO*)

Red realizuje situacije u kojima **više korisnika čeka servis sa jednog izvora**

Dodavanje novog elementa u red vrši se na **kraju reda**, dok se **operacije pristupa i uklanjanja elementa** obavljaju na **početku reda**



Red

Red ima linearnu strukturu i veoma široku praktičnu primenu (npr. funkcija bafera)

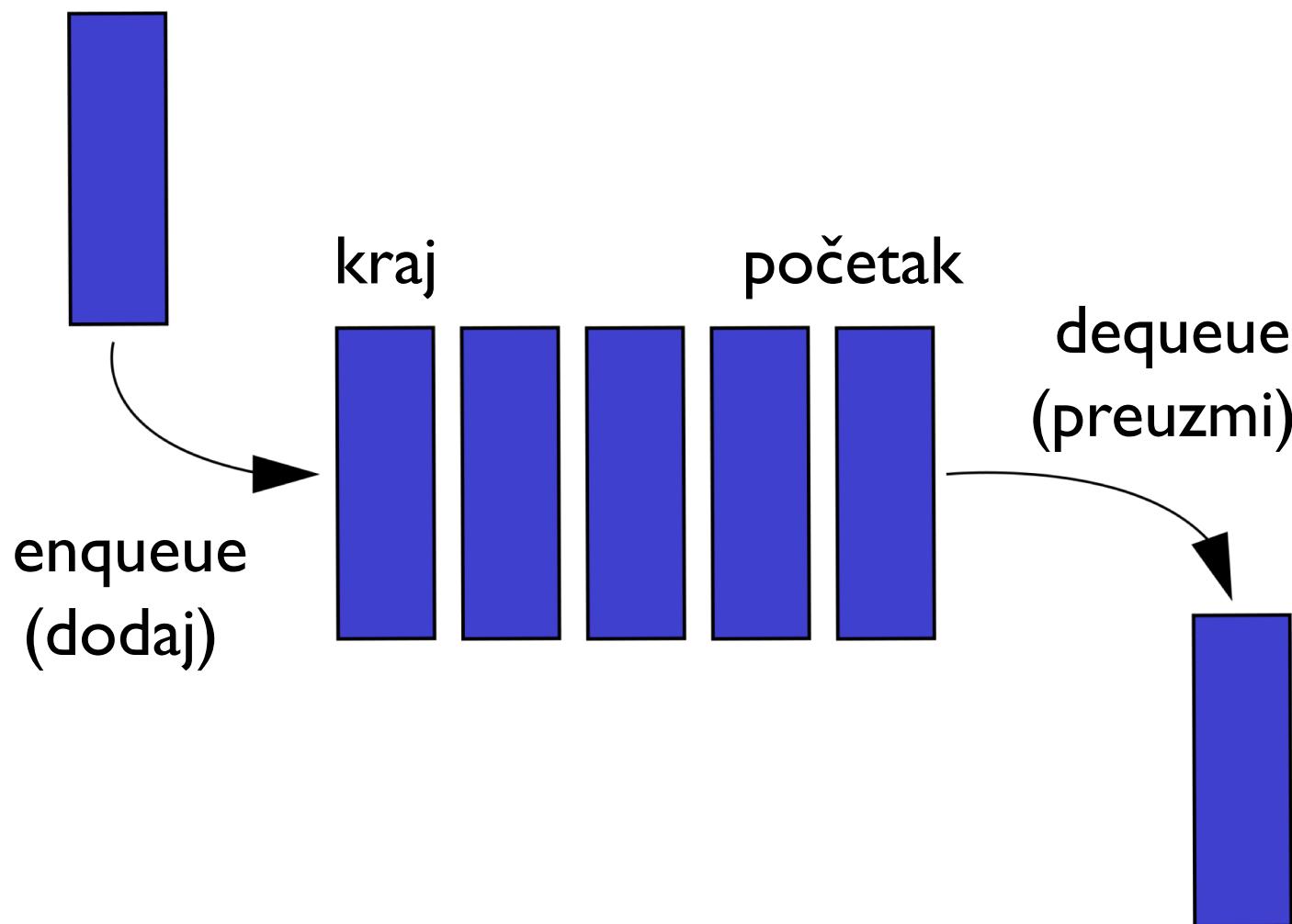
Posebne operacije za rad sa redom – **enqueue** (dodaj u red) i **dequeue** (preuzmi iz reda)

Realizacija reda može biti:

- **sekvencijalna (polje)**
 - Ako je n veličina polja, onda izračunavanjem indeksa po modulu n pretvaramo polje u krug
- **sregnuta (sregnuta lista)**
 - Prirodan izbor dvostruko sregnute liste, ali je moguća i realizacija sa jednostruko sregnutom uz dodavanje pokazivača na kraj

Red

Primer rada sa redom - operacije **enqueue** i **dequeue**:



Operacije sa redom

enqueue (dodaj) – dodavanje elementa na kraj reda

dequeue (preuzmi) – preuzimanje elementa sa početka reda

Ostale operacije:

peek ili front (vratiPocetak) – provera vrednosti na početku reda bez modifikacije stanja reda

rear (vratiKraj) – provera vrednosti na kraju reda bez modifikacije stanja reda

isEmpty (daLiJePrazan) – provera da li je red prazan

isFull (daLiJePun) – provera da li je red pun

size (vratiVelicinu) – vraća veličinu reda

Ciklični (kružni) red

Kod **cikličnog (kružnog, cirkularnog) reda** prilikom pristupa vraća se nazad na početak kada se dosegne kraj reda

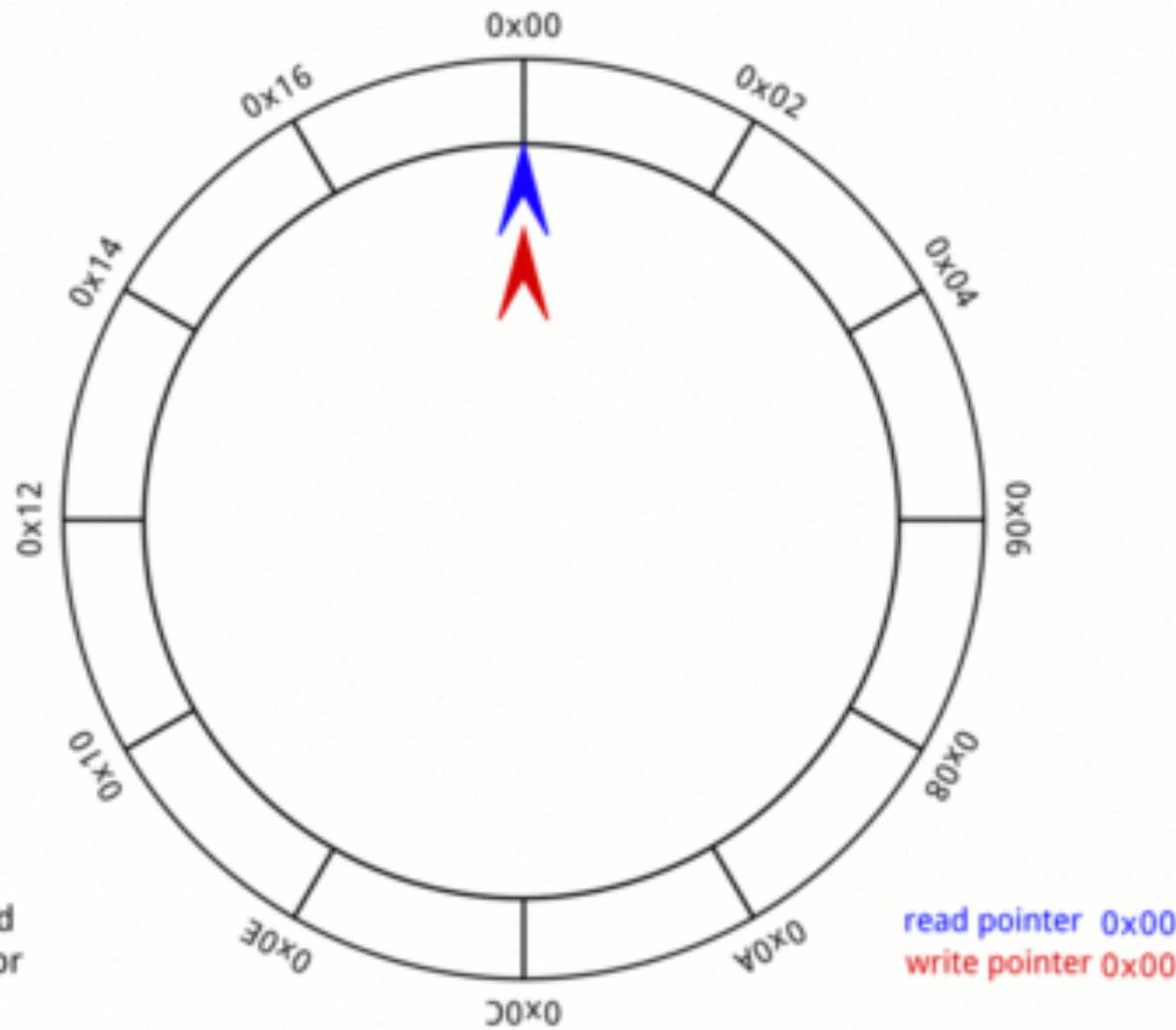
Primer kružnog reda je **cirkularni bafer** (engl. *circular or ring buffer*) za **tastaturu**, koji prima znakove brzinom kojom se kucaju, a procesu ih predaje na zahtev

Ako je proces zauzet nekim drugim poslom, ovi baferi omogućuju da se nastavi sa kucanjem, a da se ne izgubi sadržaj

Ako su **indeksi početka i kraja jednaki**, da li je **red pun ili prazan?**

Za razlikovanje ove dve situacije može se “žrtvovati” jedno mesto u redu - **indeks kraja se inicijalizuje na jedno mesto iza indeksa početka** i potom mu nije dozvoljeno da ga sustigne

Cirkularni bafer za tastaturu



Izvor: https://en.wikipedia.org/wiki/Circular_buffer#/media/File:Circular_Buffer_Animation.gif

Implementacija reda – primeri

Zadatak 1:

Napisati u jeziku C program kojim se red realizuje **sekvencijalno (u vidu polja)**. Implementirati operacije za inicijalizaciju reda, dodavanje i brisanje elemenata, pružanje informacija o prvom, odnosno poslednjem elementu u redu i proveru da li je red pun i prazan.

Zadatak 2:

Napisati u jeziku C program kojim se red realizuje **sregnuto (u vidu jednostruko sregnute liste)**. Implementirati operacije za inicijalizaciju reda, dodavanje i brisanje elemenata, proveru da li je red prazan, vraćanje informacije o veličini reda i prikaz sadržaja reda.

Vežba 1:

Izmeniti prethodne programe tako da korisnik može da bira željenu operaciju za rad sa redom. Omogućiti ponavljanje izabranih operacija sve dok korisnik ne odluči da izađe iz programa.

Sekvencijalni red – Zadatak I

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

typedef struct          // Struktura koja predstavlja red
{
    int* podaci;
    int pocetak, kraj, trenutnaVelicina;
    unsigned maxVelicina;
} tRed;

tRed* inicijalizacijaReda(unsigned);
int daLiJePun(tRed* );
int daLiJePrazan(tRed* );
void dodaj(tRed*, int);
int preuzmi(tRed* );
int vratiPocetak(tRed* );
int vratiKraj(tRed* );
```

Sekvencijalni red – Zadatak I

// Primer rada sa sekvencijalno implementiranim redom

```
int main()
{
    tRed* red = inicializacijaReda(50);

    dodaj(red, 1);
    dodaj(red, 2);
    dodaj(red, 3);
    dodaj(red, 4);
    printf("%d preuzeto iz reda!\n", preuzmi(red));
    printf("Element na pocetku reda je %d.\n", vratiPocetak(red));
    printf("Element na kraju reda je %d.\n", vratiKraj(red));

    return 0;
}
```

Sekvencijalni red – Zadatak I

```
tRed* inicijalizacijaReda(unsigned kapacitet) // Funkcija za kreiranje reda zeljenog kapaciteta
{
    tRed* red = (tRed*) malloc(sizeof(tRed));
    red->maxVelicina = kapacitet;
    red->pocetak = red->trenutnaVelicina = 0;
    red->kraj = kapacitet - 1;
    red->podaci = (int*)malloc(red->maxVelicina * sizeof(int));
    return red;
}

int daLiJePun(tRed* red)
{
    return (red->trenutnaVelicina == red->maxVelicina);
}

int daLiJePrazan(tRed* queue)
{
    return (red->trenutnaVelicina == 0);
}
```

Sekvencijalni red – Zadatak I

```
void dodaj(tRed* red, int element)
{
    if (daLiJePun(red))
        return;
    red->kraj = (red->kraj + 1) % red->maxVelicina;
    red->podaci[red->kraj] = element;
    red->trenutnaVelicina = red->trenutnaVelicina + 1;
    printf("%d dodat u red!\n", element);
}

int preuzmi(tRed* red)
{
    if (daLiJePrazan(red))
        return INT_MIN;
    int element = red->podaci[red->pocetak];
    red->pocetak = (red->pocetak + 1) % red->maxVelicina;
    red->trenutnaVelicina = red->trenutnaVelicina - 1;
    return element;
}
```

Sekvencijalni red – Zadatak I

```
int vratiPocetak(tRed* red)
{
    if (daLiJePrazan(red))
        return INT_MIN;
    return red->podaci[red->pocetak];
}
```

```
int vratiKraj(tRed* red)
{
    if (daLiJePrazan(red))
        return INT_MIN;
    return red->podaci[red->kraj];
}
```

Spregnuti red – Zadatak 2

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

```
typedef struct cvor
{
    int podatak;
    struct cvor *sledeci;
} tCvor;
```

```
typedef struct
{
    tCvor *pocetak, *kraj;
} tRed;
```

```
tCvor* kreirajCvor(int);
tRed* kreirajRed();
int daLijePrazan(tRed* );
void dodaj(tRed*, int);
tCvor* preuzmi(tRed* );
```

Spregnuti red – Zadatak 2

```
int main()
{
    tRed *red = kreirajRed();

    dodaj(red, 1);
    dodaj(red, 2);
    dodaj(red, 3);
    dodaj(red, 4);
    dodaj(red, 5);

    tCvor *n = preuzmi(red);
    if (n != NULL)
        printf("%d preuzeto iz reda!\n", n->podatak);
    return 0;
}
```

Spregnuti red – Zadatak 2

```
tCvor* kreirajCvor(int x)
{
    tCvor *noviCvor = (tCvor*)malloc(sizeof(tCvor));
    noviCvor->podatak = x;
    noviCvor->sledeci = NULL;
    return noviCvor;
}

tRed* kreirajRed()
{
    tRed *red = (tRed*)malloc(sizeof(tRed));
    red->pocetak = red->kraj = NULL;
    return red;
}

int daLiJePrazan(tRed* red)
{
    return (red->pocetak == NULL);
}
```

Spregnuti red – Zadatak 2

```
void dodaj(tRed* red, int x)
{
    // Kreiraj novi cvor
    tCvor *tekuci = kreirajCvor(x);
    // Ako je red prazan, onda je novi cvor i pocetak i kraj
    if (daLijePrazan(red))
    {
        red->pocetak = red->kraj = tekuci;
        printf("%d dodat u red!\n", x);
        return;
    }
    // Dodaj novi cvor na kraj i azuriraj kraj
    red->kraj->sledeci = tekuci;
    red->kraj = tekuci;
    printf("%d dodat u red!\n", x);
    return;
}
```

Spregnuti red – Zadatak 2

```
tCvor* preuzmi(tRed* red)
{
    if (daLiJePrazan(red))
        return NULL;
    // Sacuvaj trenutni pocetak i pomeri se na novi pocetak
    tCvor *tekuci = red->pocetak;
    red->pocetak = red->pocetak->sledeci;

    // Ako pocetak postane NULL, onda i kraj treba da bude NULL
    if (red->pocetak == NULL)
        red->kraj = NULL;
    return tekuci;
}
```

Primene reda

- Programska simulacija bilo koje situacije iz realnog života u kojoj postoji koncept reda
 - kase u prodavnicama, bioskopima ili pozorištima, kontola leta, izvršavanje finansijskih transakcija...
- Obrada zahteva klijentskih računara od strane servera na Web-u
- Red čekanja procesa za procesorsko vreme ili za pristup printeru

Vrste redova

Običan red (FIFO) (engl. queue)

- necikličan
- cikličan

Dek (engl. double-ended queue)

Red sa prioritetom (engl. priority queue)

Dek sa prioritetom (engl. priority dequue)

Dek

Dek (engl. *dequeue* – *double-ended queue*) je apstraktni tip podataka koji predstavlja kontejner u koji se **elementi mogu dodavati ili iz koga se elementi mogu brisati i na početku i na kraju reda** – red sa dva početka/kraja

Dek predstavlja **uopštenje steka i reda** u smislu načina pristupa, tj. dodavanja i uklanjanja elemenata

Najčešće se **implementira** pomoću **dvostruko spregnute liste** – (engl. *head/tail linked list*)

Operacije sa dekom

insertFirst (dodajPrvi) – dodavanje elementa na početak deka

removeFirst (preuzmiPrvi) – preuzimanje elementa sa početka deka

insertLast (dodajPoslednji) – dodavanje elementa na kraj deka

removeLast (preuzmiPoslednji) – preuzimanje elementa sa kraja deka

Ostale operacije:

first (vratiPrvi) – provera vrednosti na početku deka bez modifikacije deka

last (vratiPoslednji) – provera vrednosti na kraju deka bez modifikacije deka

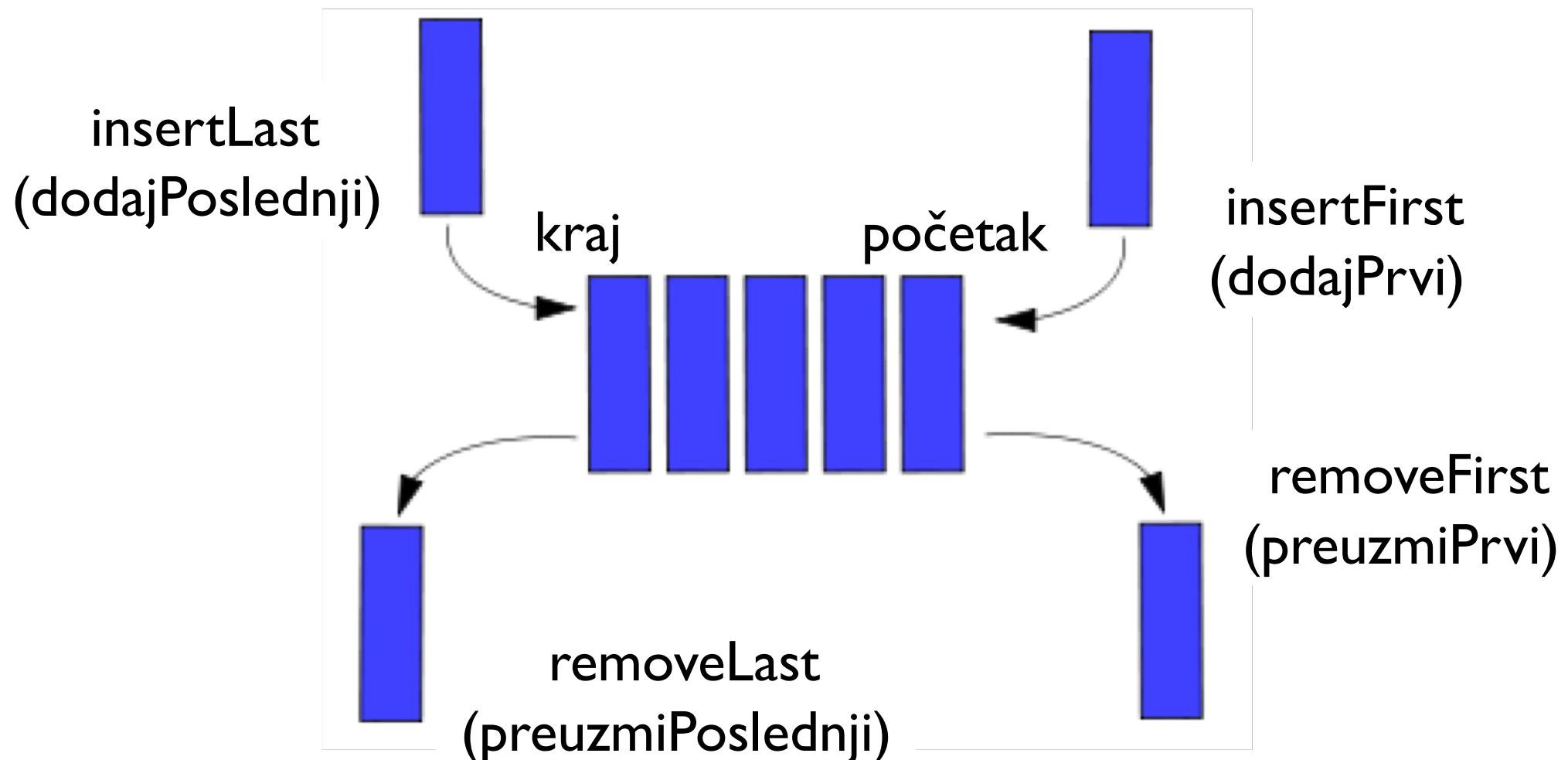
isEmpty (daLiJePrazan) – provera da li je dek prazan

isFull (daLiJePun) – provera da li je dek pun

size (vratiVeličinu) – vraća veličinu deka

Dek

Primer rada sa dekom:



Red sa prioritetom

Red sa prioritetom (engl. *priority queue*) je **proširenje apstraktnog tipa podataka red** kod koga su **elementima pridruženi odgovarajući prioriteti**.

Elementi se uklanjuju sa početka reda

Elementi su **uređeni** na osnovu vrednosti polja **prioriteta** tako da se oni **sa najvišim prioritetom nalaze na početku reda**

Ako dva elementa imaju **isti prioritet**, onda se “opslužuju” u skladu sa njihovim **redosledom u običnom redu**

Obično se implementiraju pomoću **gomila** (engl. heap)

Operacije kod reda sa prioritetom

insertWithPriority (dodajSaPrioritetom) – dodaj element sa pridruženim prioritetom u red

**pullHighestPriorityElement
(preuzmiElementSaNajvisimPrioritetom)** – preuzmi element sa najvišim prioritetom iz reda

Ostale operacije:

peek (proveri) – provera vrednosti na početku reda bez modifikacije, u ovom kontekstu obično se zove **findMax** ili **findMin**

isEmpty (daLiJePrazan) – provera da li je red sa prioriteom prazan

Primene reda sa prioritetom

Upravljanje ograničenim resursima – dodela procesorskog vremena procesima u višeprocesnim operativnim sistemima, propusni opseg kod rutera

Sortiranje pomoću gomile (engl. *heapsort*) – algoritam za sortiranje zasnovan na dodavanju elemenata u red sa prioritetom

Dijkstrin algoritam za pronalaženje najkraćeg puta između para čvorova u grafu

Huffman-ovo kodiranje – metod za kompresiju podataka

Nelinearne strukture podataka

Vrste apstraktnih tipova podataka

- **lista** (engl. *list*)
- **stek** (engl. *stack*)
- **red** (engl. *queue*)
- **red sa prioritetom** (engl. *priority queue*)
- **dek** (engl. *double-ended queue – deque*)
- **dek sa prioritetom** (engl. *priority deque*)
- **stablo** (engl. *tree*)
- **graf** (engl. *graph*)
- **mapa, multimapa** (engl. *map, multimap*)
- **skup, multiskup** (engl. *set, multiset*)

Strukture podataka

Linearne strukture podataka

- polje (niz)
- spregnuta (povezana) lista
- strukture sa posebnim pravilima pristupa:
 - stek
 - red
 - dek

Nelinearne strukture podataka

- stablo
- graf

Nelinearne strukture podataka

Nelinearne strukture podataka su strukture u kojima elementi nisu organizovani sekvensijalno (mogu imati više od jednog prethodnika i jednog sledbenika)

Elementi u nelinearnim strukturama podataka mogu biti povezani sa većim brojem drugih elemenata čime se odražava poseban odnos između tih elemenata

U opštem slučaju, svi elementi u nekoj nelinearnoj strukturi podataka ne mogu se obići u jednom prolazu

Dva najznačnija primera nelinearnih struktura podataka su **stablo** i **graf**

Stablo

Stablo

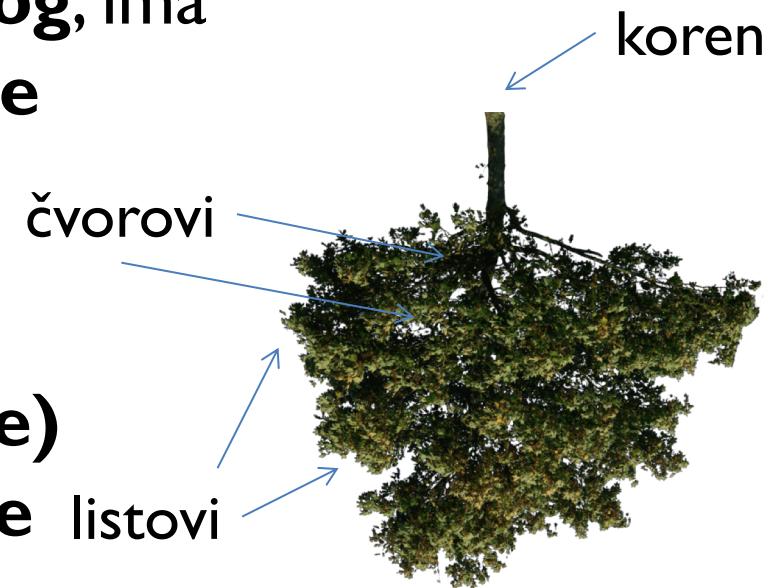
Stablo (engl. tree) je **apstraktni tip podataka i struktura podataka** koja **implementira istoimeni ATP**, u vidu koje se predstavlja **hijerarhijski odnos između elemenata** koji su u **relaciji roditelj-dete** (nadređeni-podređeni), elementi se obično pamte u **čvorovima** (engl. node)

Svaki element u stablu, **osim vršnog**, ima **jednog roditelja** i **nula ili više dece**

Vršni element je **koren stabla** (engl. root)

Razlikujemo **interne (neterminalne)**

i **eksterne (terminalne) elemente**



Stablo - primene

Primene stabla:

Organizacione šeme

Fajl sistemi

Grafički korisnički interfejsi

(engl. *Graphical User Interface - GUI*)

Baze podataka

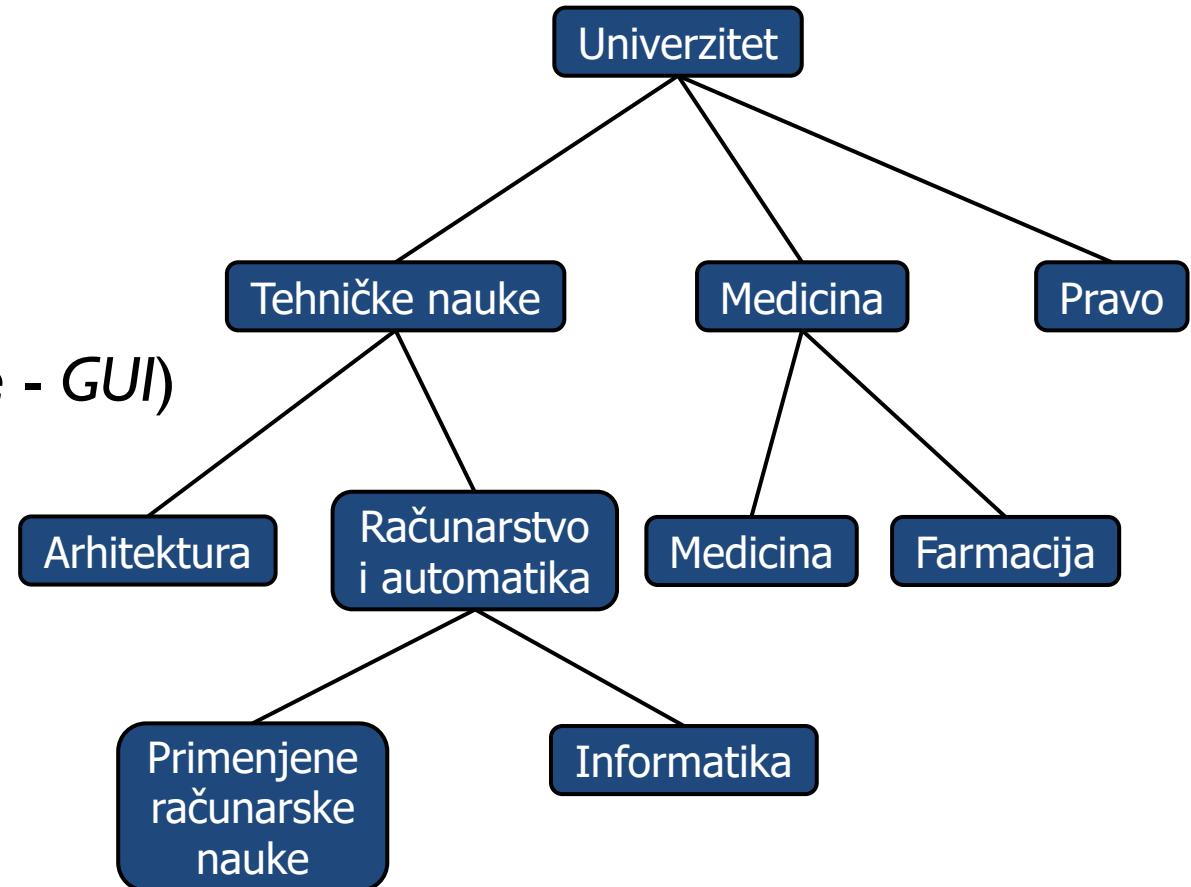
Programski prevodioci

Web sajtovi

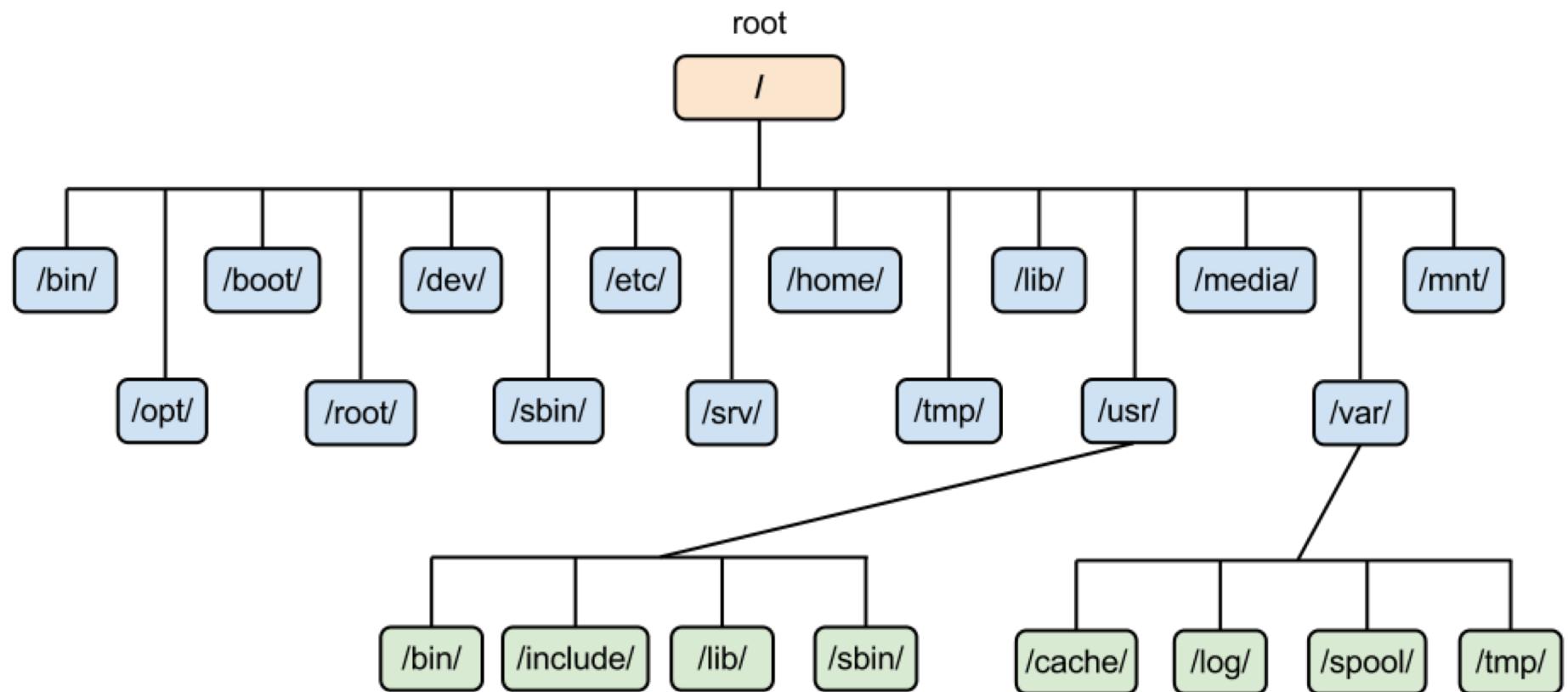
Okruženja za programiranje

(engl. *Integrated Development Environment – IDE*)

...



Primer stabla – Linux fajl sistem



Izvor: <https://freedompenguin.com/articles/how-to/learning-the-linux-file-system/>

Stablo – matematička definicija

Stablo S je konačan neprazan skup elemenata

$$S = \{k\} \cup S_1 \cup S_2 \cup \dots \cup S_n$$

Sa sledećim svojstvima:

1. Elemenat k je **koren** stabla
2. Ostali elementi su podeljeni u $n \geq 0$ podskupova S_1, S_2, \dots, S_n od kojih je **svaki stablo**

S_1, S_2, \dots, S_n su **podstabla** stabla S

Rekurzivna definicija!

Specijalni slučaj $n = 0$ – svako stablo ima barem jedan element – koren stabla

Stablo – apstraktni tip i struktura

Posmatrano kao **apstraktni tip podataka**, **stablo** ima **vrednost (podatak)** i **decu**, gde su **deca opet stabla** – **rekurzivni pojam**

Vrednost stabla se interpretira kao podatak zapisan u korenu, a **deca stabla** kao podstabla koja su deca korena

Posmatrano kao **struktura podataka**, **stablo** je **grupa čvorova** u kojoj **svaki čvor ima vrednost i listu pokazivača na druge čvorove** (njegovu decu)

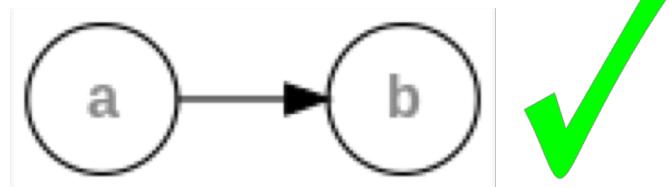
Ograničenja: dva pokazivača “na dole” ne smeju pokazivati na isti čvor, može biti samo jedan koren, ciklusi ne smeju postojati

Čvorovi u stablu mogu da čuvaju i pokazivače na sledeći/prethodni element kao i na svog roditelja

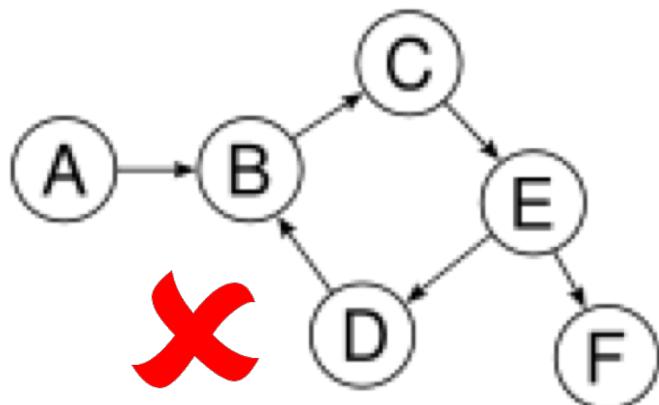
Stabla se najčešće **realizuju** u vidu **pokazivača na koren**

Stablo

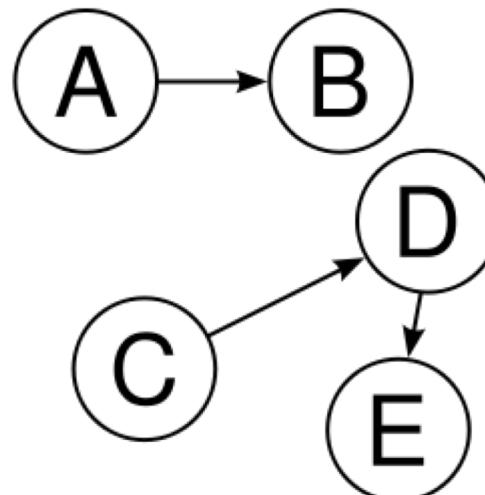
Stablo je digraf (engl. *digraph* – *directed graph*) tj. **usmereni graf** koji se sastoji od **čvorova** i **potega** i ne sme sadržati **cikluse**



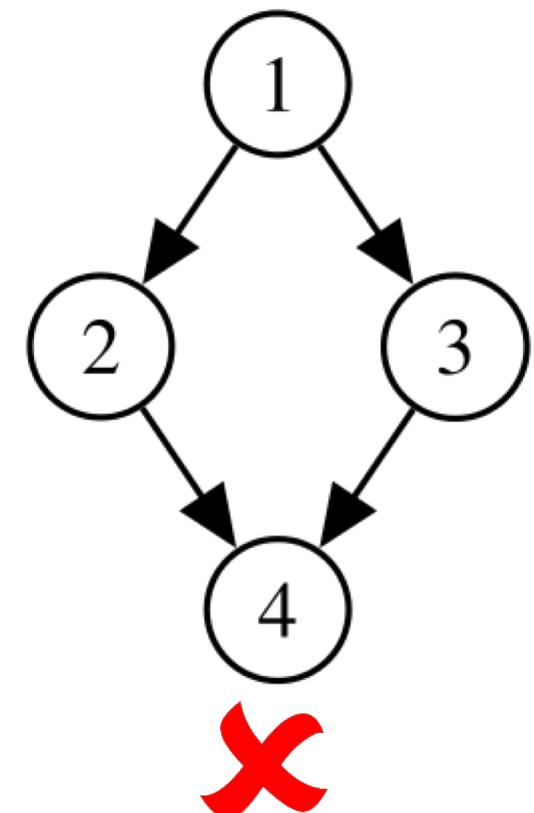
Spregnuta lista je trivijalan slučaj stabla!



Ciklusi su zabranjeni!



Može postojati
samo jedan koren!



Svaki čvor ima samo
jednog roditelja!

Stablo - terminologija

Koren (engl. *root*) – čvor bez roditelja (A)

Braća/sestre (engl. *siblings*) – čvorovi koji imaju zajedničkog roditelja

Interni (neterminalni) čvor (engl. *internal node*) – čvor sa najmanje jednim detetom (A, B, C, F)

Eksterni (terminalni) čvor (engl. *external node*) – list (engl. *leaf*) – čvor bez dece (E, I, J, K, G, H, D)

Preci (engl. *ancestors*) **čvora** – roditelj, pra-roditelj, pra-pra-roditelj itd.

Potomci (engl. *descendants*) **čvora** – dete, unuk, praunuk, itd.

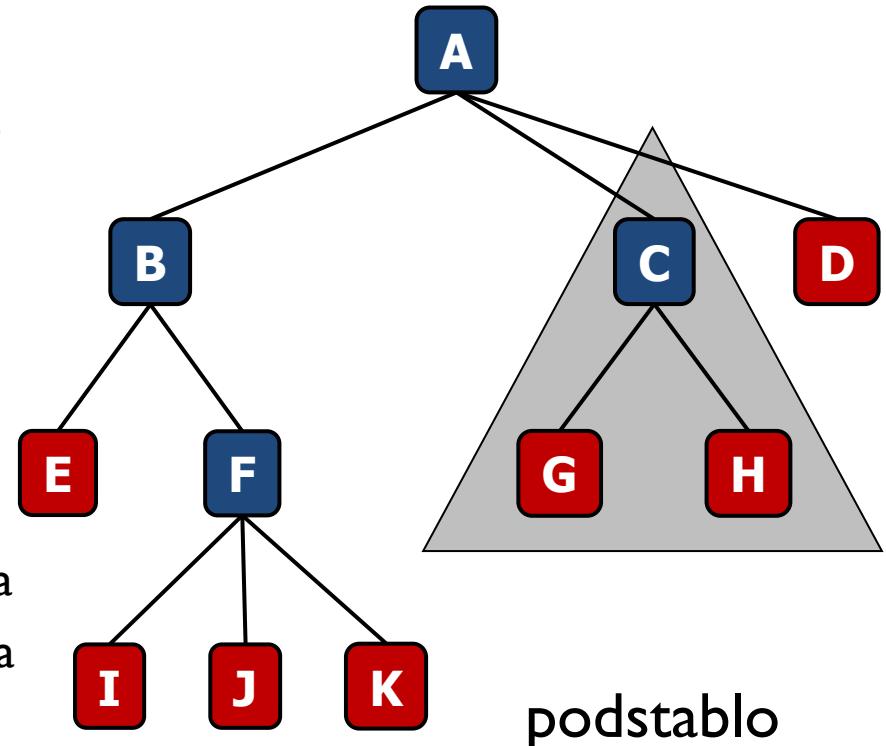
Dubina (engl. *depth*) **čvora** – broj potega do korena

Visina (engl. *height*) **stabla** – maksimalni broj potega od korena do listova u stablu (3)

Stepen (engl. *degree*) **čvora** – broj njegove dece

Stepen stabla – maksimalni step nekog njegovog čvora

Podstablo (engl. *subtree*) – stablo sastavljeno od nekog čvora i njegovih potomaka



Stablo - terminologija

Stablo može biti:

- **neuređeno** (engl. *unordered*)
- **uređeno** (engl. *ordered*)

Stablo je uređeno ako postoji linearno uređenje definisano za decu svakog čvora

- bitno je relativno uređenje podstabala u svakom čvoru
- deca se identifikuju kao prvo, drugo, treće, itd.

Stablo - operacije

Opšte (engl. generic) operacije:

- **size (veličina)** – vraća veličinu stabla
- **isEmpty (jePrazno)** – provera da li je stablo prazno

Pristupne (engl. accessor) operacije:

- **root (koren)** – vraća poziciju korena
- **parent (roditelj)** – vraća poziciju roditelja
- **children (deca)** – vraća iterator kojim se mogu obići potomci čvora

Stablo - operacije

Operacije upita (engl. *query*):

- **isInternal (jeInterni)** – provera da li je čvor interni
- **isExternal (jeEksterni)** – provera da li je čvor eksterni
- **isRoot (jeKoren)** – provera da li je čvor koren

Operacije ažuriranja (engl. *update*):

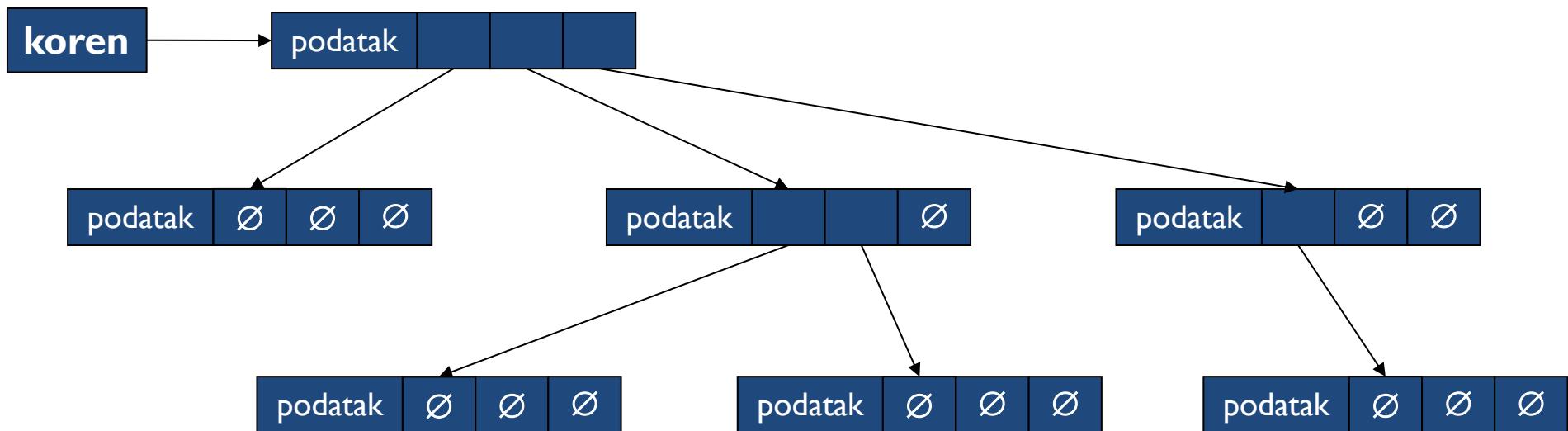
- **swapElements (zameniMestaPodacima)** – zamena vrednosti podataka između dva čvora
- **replaceElement (zameniVrednostPodatka)** – zamena vrednosti podatka u nekom čvoru

Dodatne operacije ažuriranja definišu se u skladu sa strukturom podataka kojom se implementira ATP stablo

Stablo – primer spregnute realizacije

Svaki čvor u stablu sadrži:

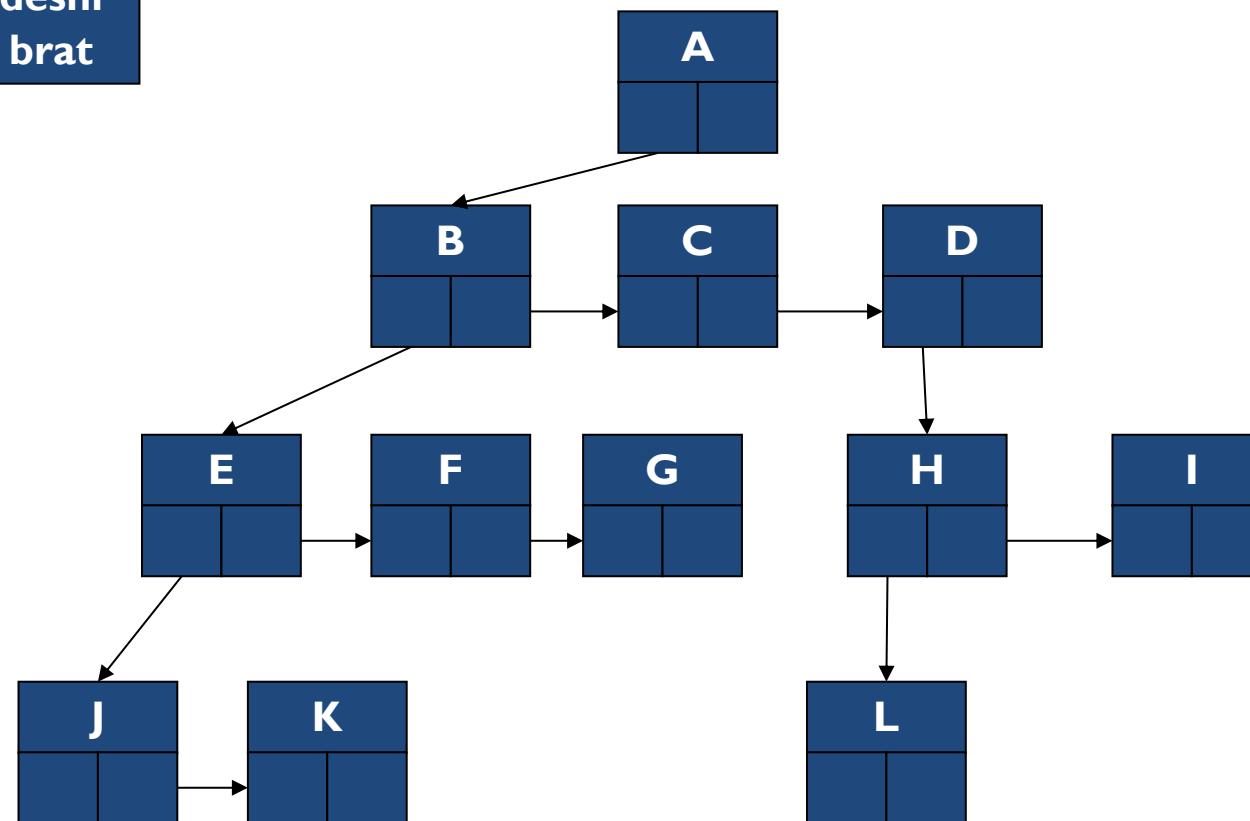
- polje **podatak** – korisne informacije
- polja **deca** – pokazivači na njegovu decu (\emptyset oznaka za NULL)



Stablo – primer spregnute realizacije

podatak	
levo dete	desni brat

Reprezentacija krajnje levo dete – desni brat



Stablo - obilazak

Obilazak (engl. *traversal*) stabla predstavlja postupak pristupanja njegovim **čvorovima na sistematski način**

Dva glavna načina obilaska su:

- **preorder (sa vrha ka dnu)**
- **postorder (sa dna ka vrhu)**

Priroda ovih postupaka je **rekurzivna**

Preorder (sa vrha ka dnu):

- poseti koren
- obidi u preorder redosledu decu (podstabla)

Postorder (sa dna ka vrhu):

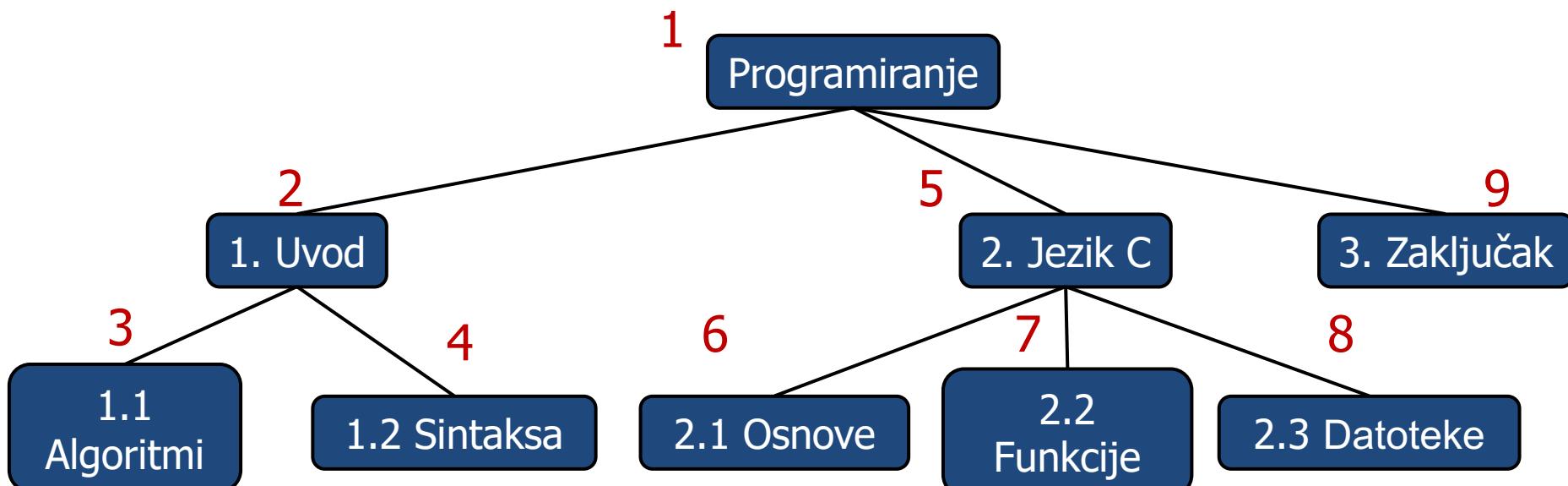
- obidi u postorder redosledu decu (podstabla)
- poseti koren

Stablo - preorder obilazak

Kod obilaska sa vrha ka dnu, čvor se posećuje pre posete njegovim potomcima

```
algoritam preorder(v)
  poseti(v)
  za svako dete w čvora v
    preorder (w)
```

Primer: prikaz strukture dokumenta

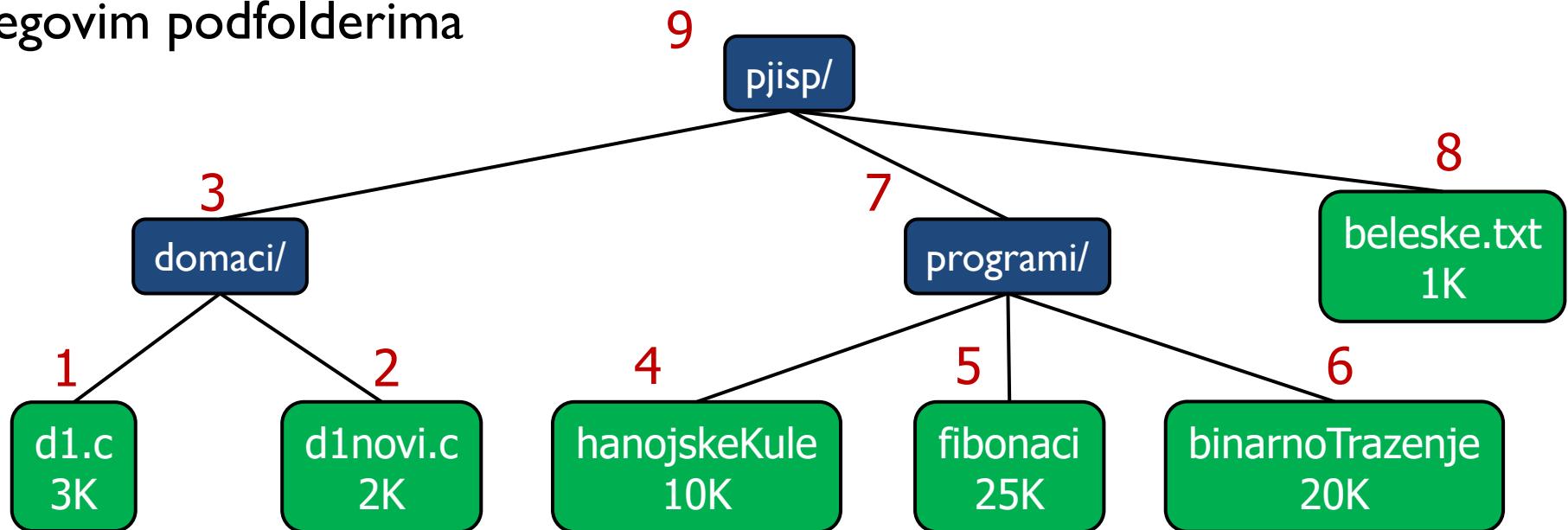


Stablo - postorder obilazak

Kod obilaska sa dna ka vrhu, čvor se posećuje nakon posete potomaka

```
algoritam postorder(v)
za svako dete w čvora v
    postorder (w)
    poseti(v)
```

Primer: izračunavanje zauzeća memorije od strane fajlova u folderu i njegovim podfolderima



Stablo – primer sintaksno stablo

Apstraktno sintaksno stablo (engl. *abstract syntax tree* – AST) ili, prostije, **sintaksno stablo**, je reprezentacija apstraktne sintaksne strukture izvornog koda napisanog u nekom programskom jeziku

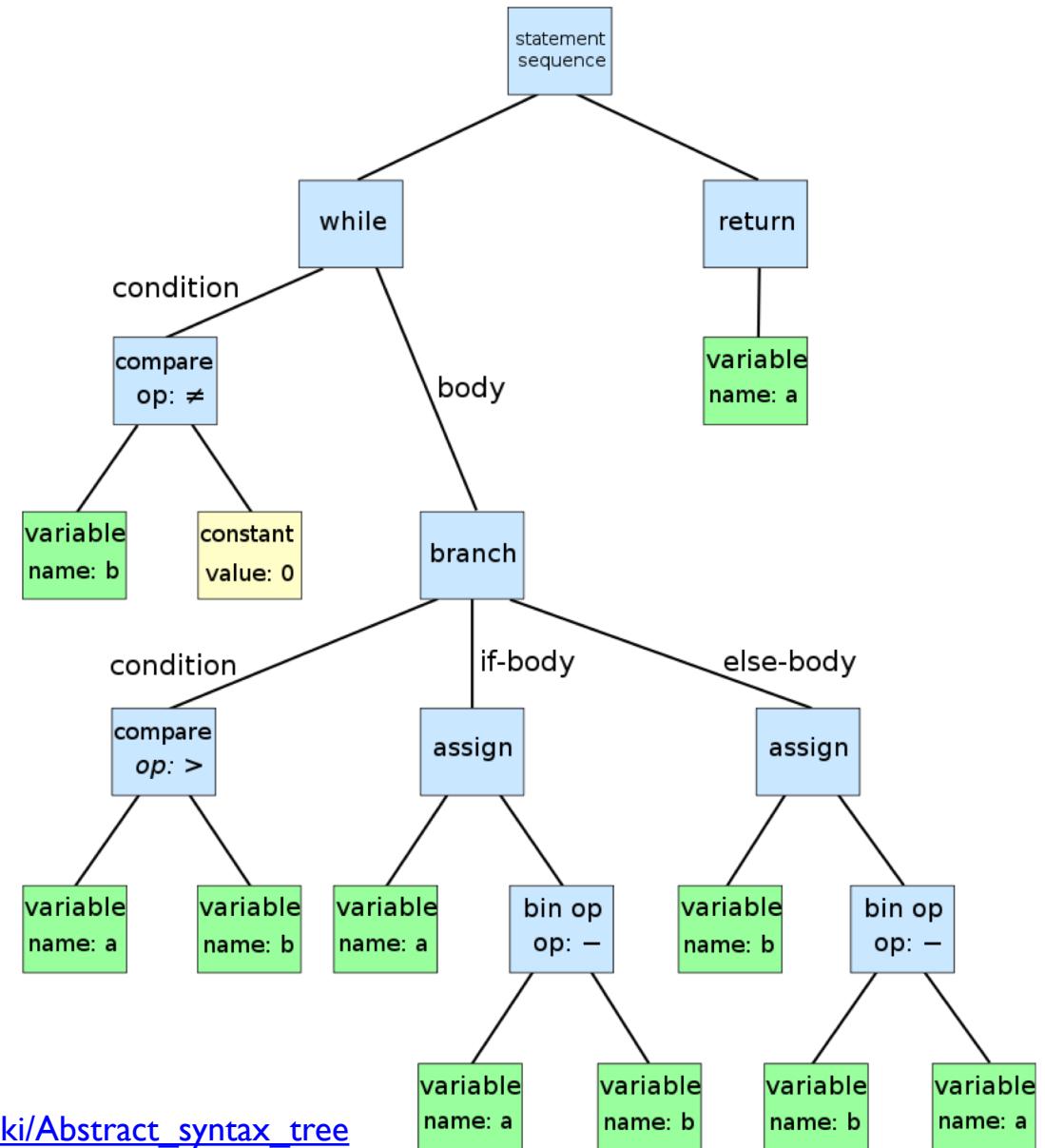
Svaki **čvor u sintaksnom stablu** odgovara nekom **elementu** koji se javlju u **konstrukciji izvornog koda**

Sintaksa je apstraktna u smislu da **ne predstavlja svaki detalj** koji se pojavljuje u “pravoj” sintaksi – npr. vitičaste zgrade { } koje ograničavaju blokove su implicitne, kao i karakteri ; kojima se označava kraj naredbe

Stablo – primer sintaksno stablo

Euklidov algoritam:

```
while (b != 0){
    if (a > b)
        a = a - b;
    else
        b = b - a;
}
return a;
```



Izvor: https://en.wikipedia.org/wiki/Abstract_syntax_tree

Binarno stablo

Binarno stablo

Binarno stablo (engl. *binary tree*) je uređeno stablo kod kojeg svaki čvor ima najviše dvoje dece

Deca čvora u binarnom stablu nazivaju se **levo i desno dete** (engl. *left and right child*)

Podstabla S_1 i S_2 čvora S nazivaju se **levo i desno podstablo**

Binarno stablo koje u svakom čvoru ima ili nijedno ili oba podstabla naziva se **striktno binarno stablo (pravo binarno stablo)**

Realizacija binarnog stabla može biti **sekvencijalna** i **spregnuta**

Binarno stablo - vrste

Potpuno binarno stablo visine h ima $2^{h+1}-1$ čvorova

Potpuno binarno stablo (visine h) je **striktno binarno stablo** čiji su svi listovi na nivou h

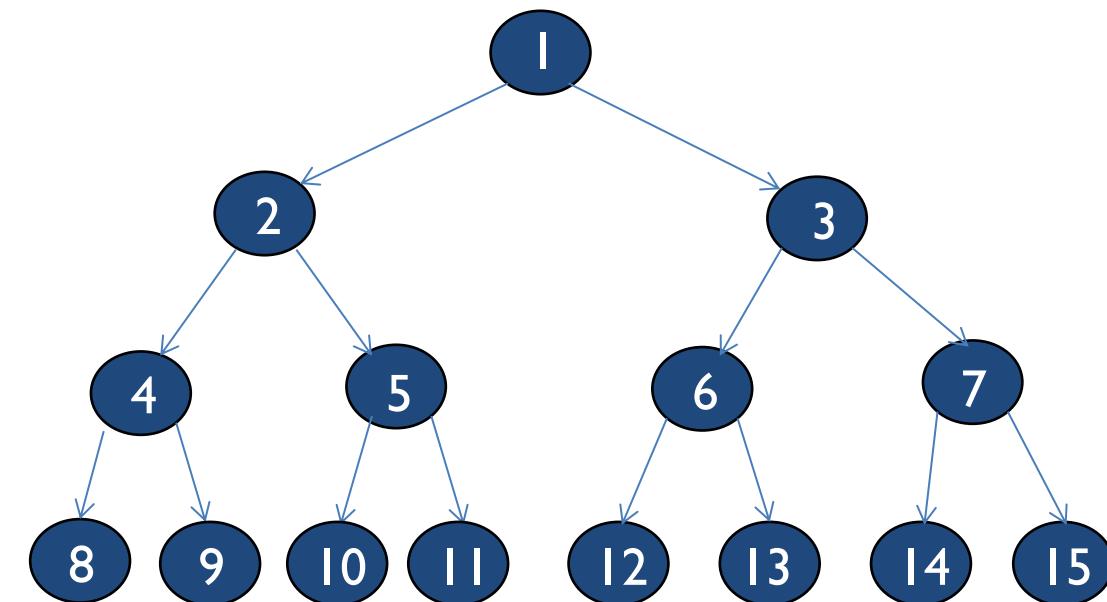
Gotovo potuno binarno stablo (visine h) je **striktno binarno stablo** čiji su svi listovi na nivou h ili $h-1$ i kod koga bilo koji čvor koji ima desnog potomka na nivou h ima i sve leve potomke na nivou h

Kompletno binarno stablo je binarno stablo kod koga su **svi nivoi, osim možda poslednjeg, kompletno popunjeni** i svi čvorovi su **što je moguće više “levo”**

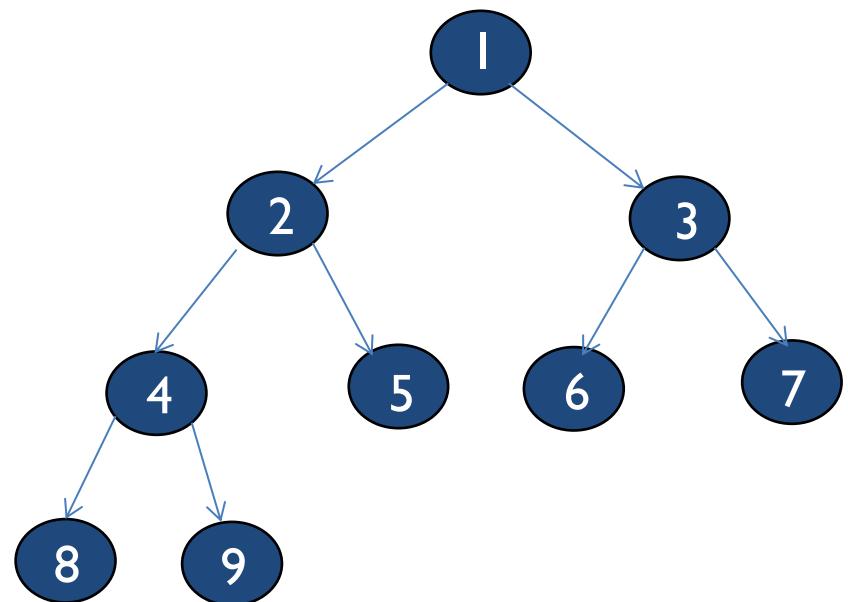
Balansirano binarno stablo je binarno stablo kod koga se **broj elemenata podstabla na istom hijerarhijskom nivou razlikuje najviše za 1**

Binarno stablo - vrste

Potpuno binarno stablo



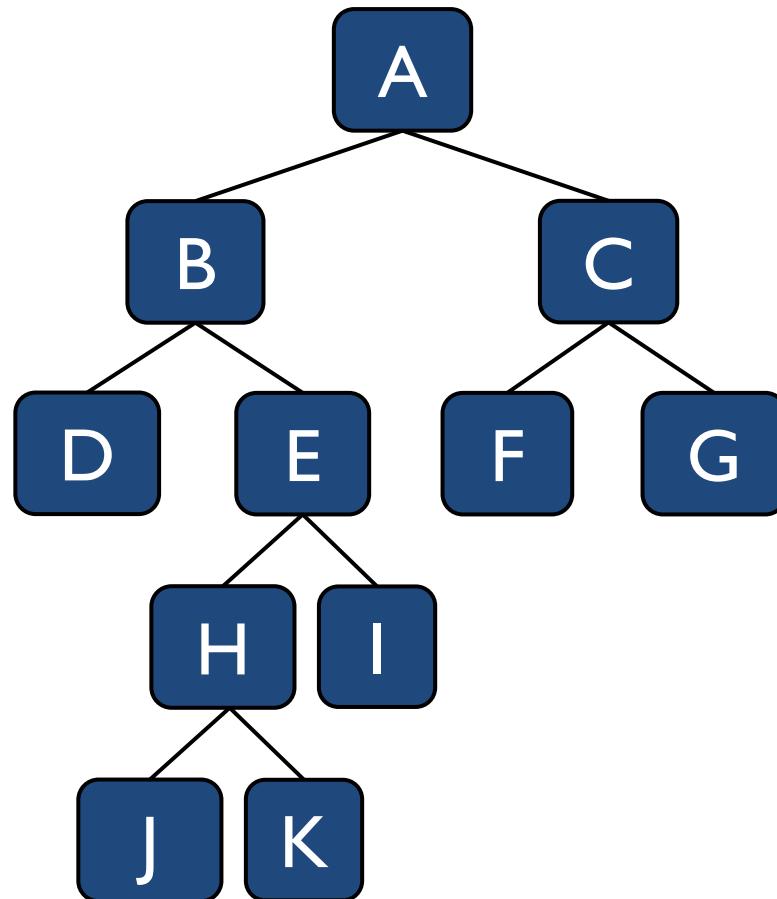
Kompletno binarno stablo



Binarno stablo – primer i primene

Primene binarnih stabala:

- Evaluacija aritmetičkih izraza
- Procesi odlučivanja
- Traženje –
binarna stabla traženja
- Sortiranje –
gomila (engl. *heap*)
heapsort algoritam



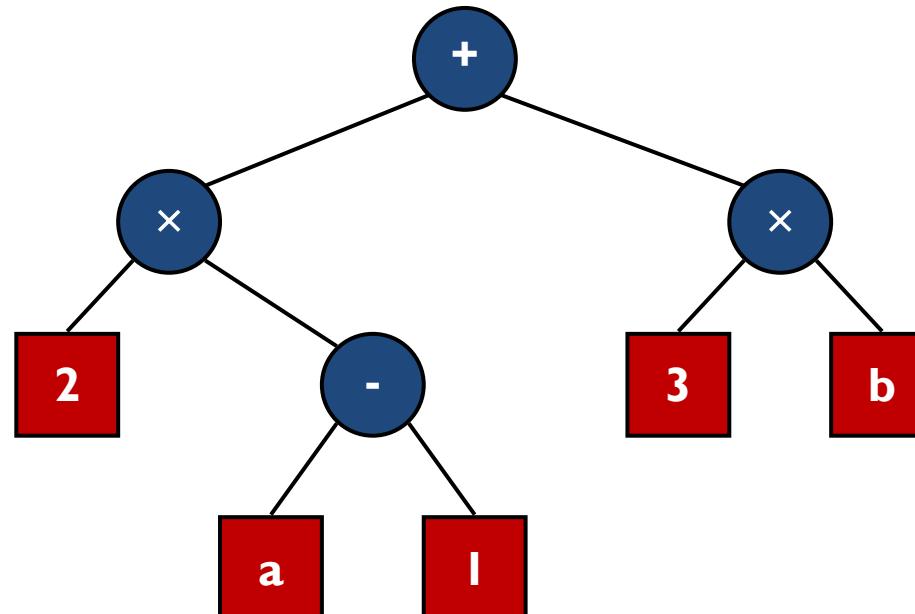
Binarno stablo - primena

Binarno **stablo aritmetičkog izraza**

interni čvorovi – operatori

eksterni čvorovi – operandi

Primer: binarno stablo za aritmetički izraz $(2 \times (a - l)) + (3 \times b)$



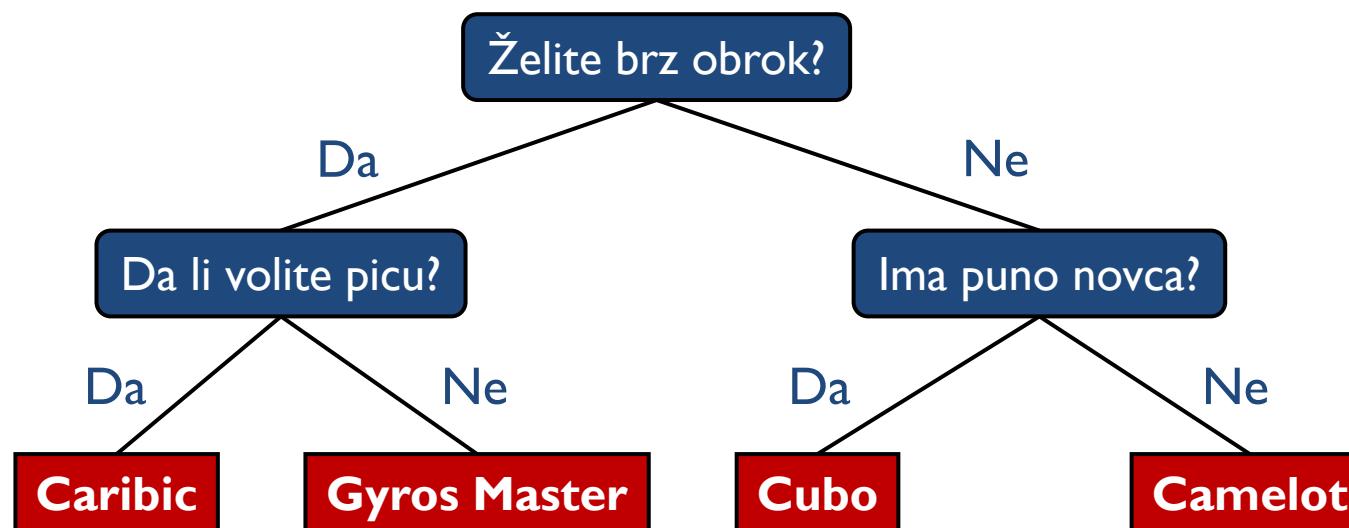
Binarno stablo - primena

Binarno **stablo odlučivanja** (engl. *binary decision tree*)

interni čvorovi – pitanja sa da/ne odgovorima

eksterni čvorovi – odluke

Primer: gde za večeru?



Binarno stablo - operacije

Binarno stablo kao apstraktni tip podataka je proširenje stabla, tj. nasledjuje sve operacije definisane za ATP stablo

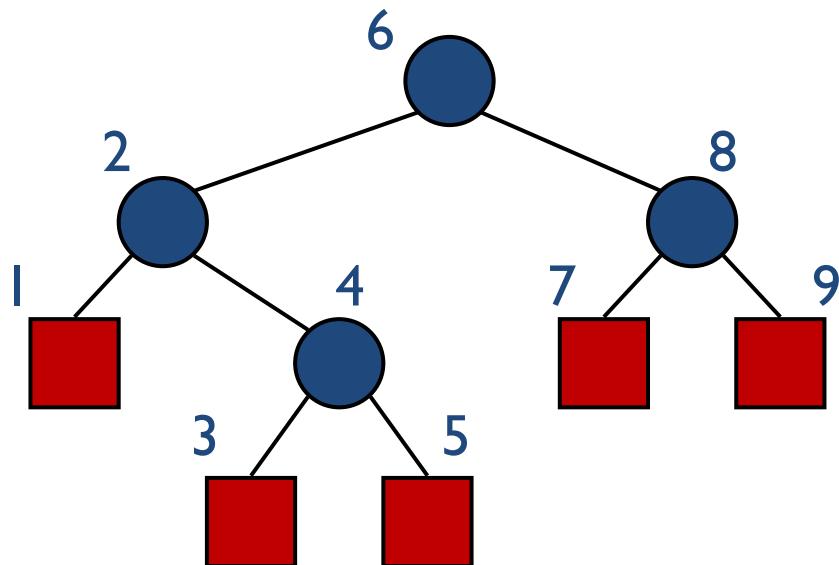
Dodatne operacije:

- **leftChild (levoDete)** – vraća poziciju levog deteta
- **rightChild (desnoDete)** – vraća poziciju desnog deteta
- **sibling (brat)** – vraća poziciju brata

Prilikom implementacije obično se definišu i dodatne operacije za ažuriranje binarnog stabla

Binarno stablo - obilazak

Kod binarnih stabala, pored preorder i postorder obilazaka, definišemo i **inorder** obilazak (s leva na desno) – čvor se posećuje nakon obilaska njegovog **levog podstabla**, a pre obilaska desnog **podstabla**

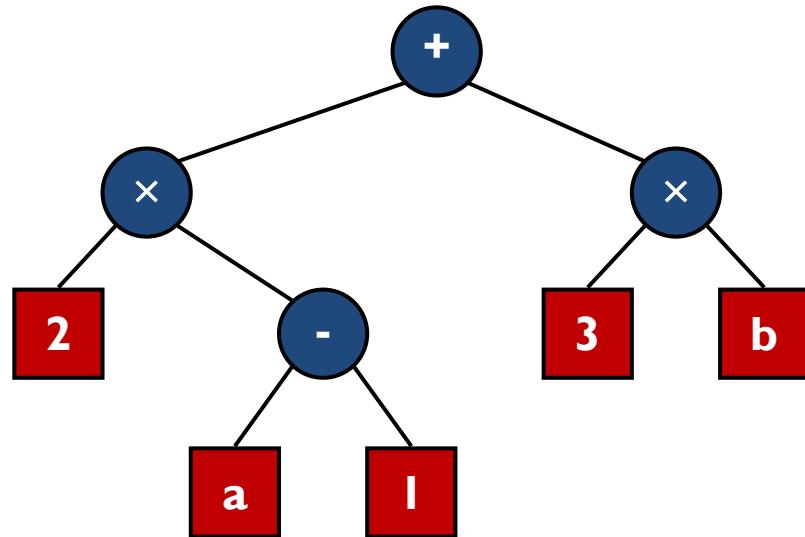


algoritam *inorder(v)*
ako je **jelInterni (v)**
 inorder (levoDete (v))
 poseti(v)
ako je **jelInterni (v)**
 inorder (desnoDete (v))

Prikaz aritmetičkih izraza

Specijalizacija inorder obilaska:

- odštampaj operand ili operator prilikom posete čvoru
- odštampaj '(' pre obilaska levog podstabla
- odštampaj ')' posle obilaska desnog podstabla



$$((2 \times (a - l)) + (3 \times b))$$

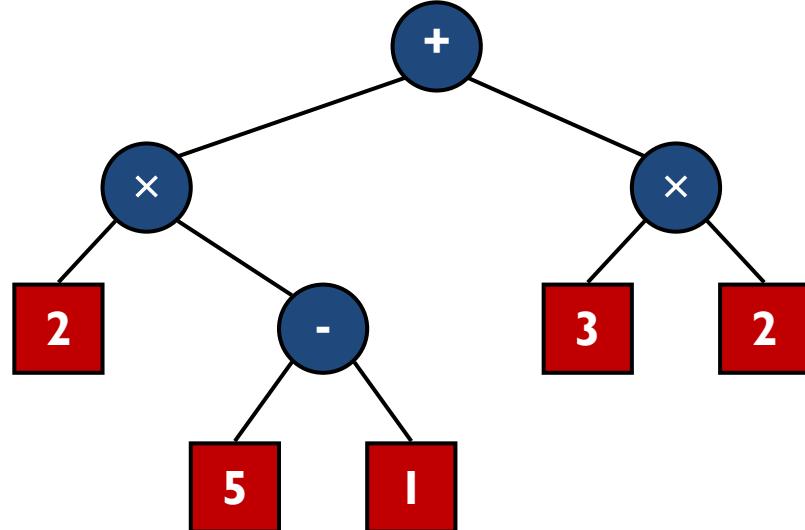
```

algoritam inorder (v)
ako jeInterni (v){
  štampaj('(')
  inorder (levoDete (v))
  štampaj(v.podatak ())
ako jeInterni (v){
  inorder (desnoDete (v))
  štampaj (')')
  
```

Evaluacija aritmetičkih izraza

Rekurzivna funkcija koja vraća vrednost podstabla

Prilikom posete internom čvoru, kombinuju se vrednosti podstabala



algoritam evalIzraz(v)

ako *jeEksterni (v)*

vrati *v.podatak ()*

u suprotnom

x \leftarrow *evalIzraz(levodete (v))*

y \leftarrow *evalIzraz(desnoDete (v))*

\diamond \leftarrow operator smešten u *v*

vrati *x* \diamond *y*

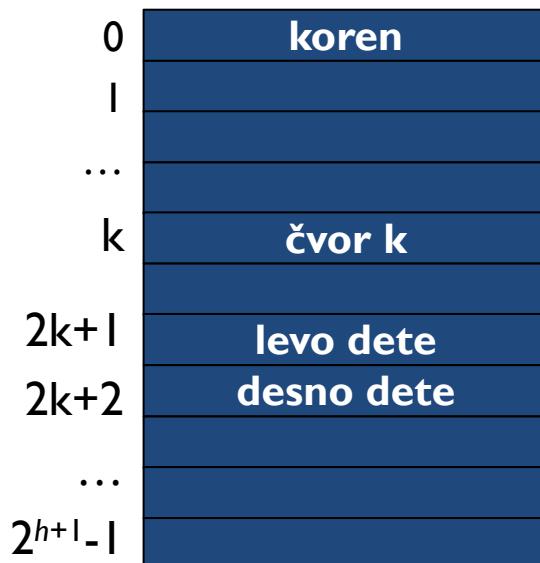
I4

Binarno stablo – sekvensijalna realizacija

Reprezentacija pogodna za **potpuno ili gotovo potpuno binarno stablo**

Koristi se polje **stablo** dužine $2^{h+1}-1$ (h - visina stabla)

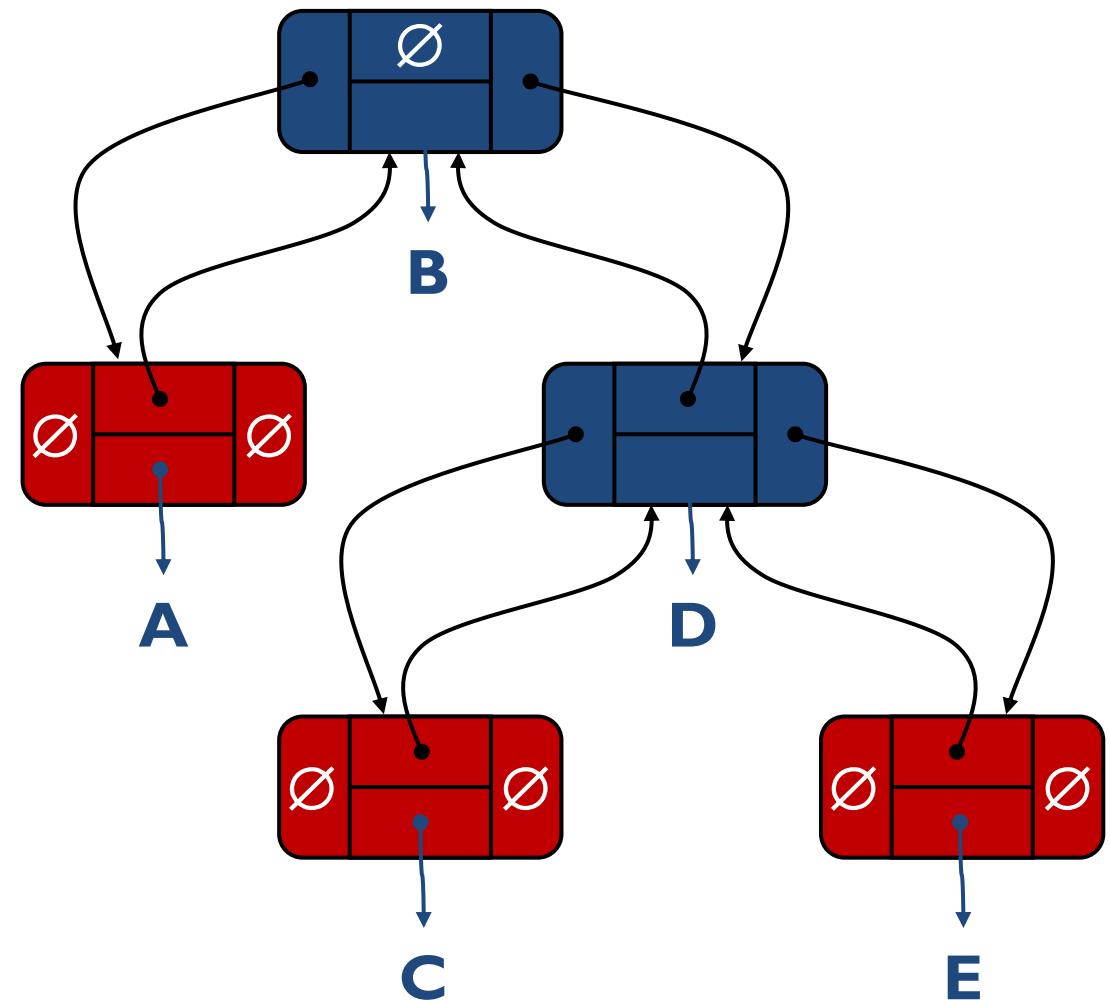
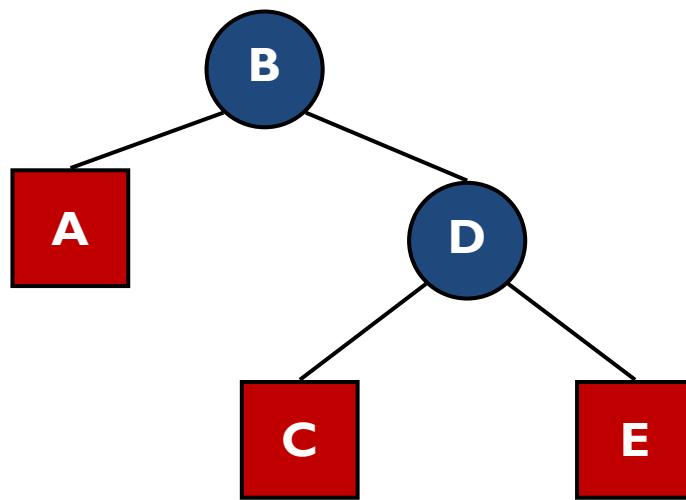
stablo[0] pamti **koren**, ako se u **stablo[k]** pamti vrednost **čvora k**, njegovo **levo dete** se pamti u **stablo[2k+1]**, a **desno dete** u **stablo[2k+2]**, **roditelj** se dobija kao **stablo[(k-1)/2]** (celobr. deljenje)



Binarno stablo – spregnuta realizacija

Čvor se sastoji od sledećih polja:

- **podatak**
- **pokazivač na roditelja**
- **pokazivač na levo dete**
- **pokazivač na desno dete**



Implementacija binarnog stabla

Zadatak I:

Napisati u jeziku C program kojim se **binarno stablo** realizuje **spregnuto**. Implementirati operacije za dodavanje i traženje elemenata u stablu, brisanje stabla, preorder, inorder i postorder obilazak stabla.

Vežba I:

Izmeniti prethodni program tako da korisnik može da bira željenu operaciju za rad sa binarnim stablom. Omogućiti ponavljanje izabranih operacija sve dok korisnik ne odluči da izađe iz programa.

Binarno stablo – Zadatak I

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

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

```
typedef struct cvor {  
    int podatak;  
    struct cvor *desnoDete, *levoDete;  
} tCvorBinarnogStabla;
```

```
void dodajCvor(tCvorBinarnogStabla**, int);
```

```
void preorderObilazak(tCvorBinarnogStabla*);
```

```
void inorderObilazak(tCvorBinarnogStabla*);
```

```
void postorderObilazak(tCvorBinarnogStabla*);
```

```
void obrisiStablo(tCvorBinarnogStabla*);
```

```
tCvorBinarnogStabla* trazi(tCvorBinarnogStabla**, int);
```

Binarno stablo – Zadatak I

```
int main()
{
    tCvorBinarnogStabla *koren, *pomocni;
    koren = NULL;

    // Dodavanje cvorova u stablo
    dodajCvor(&koren, 7);
    dodajCvor(&koren, 4);
    dodajCvor(&koren, 13);
    dodajCvor(&koren, 6);
    dodajCvor(&koren, 42);
    dodajCvor(&koren, 17);
    dodajCvor(&koren, 2);

    // Obilazak stabla
    printf("Preorder (sa vrha ka dnu) obilazak:\n");
    preorderObilazak(koren);

    printf("Inorder (sa leva na desno) obilazak:\n");
    inorderObilazak(koren);

    ...
}
```

Binarno stablo – Zadatak I

...

```
printf("Postorder (sa dna ka vrhu) obilazak:\n");
postorderObilazak(koren);

// Trazenje vrednosti u stablu
pomocni = trazi(&koren, 4);
if (pomocni)
{
    printf("Pronadjen cvor=%d\n", pomocni->podatak);
}
else
{
    printf("Vrednost nije pronadjena u stablu!\n");
}

// Brisanje stabla
obrisiStablo(koren);

return 0;
}
```

Binarno stablo – Zadatak I

```
void dodajCvor(tCvorBinarnogStabla **stablo, int vrednost)
{
    tCvorBinarnogStabla *noviCvor = NULL;
    if (!(*stablo))
    {
        noviCvor = (tCvorBinarnogStabla *)malloc(sizeof(tCvorBinarnogStabla));
        noviCvor->levoDete = noviCvor->desnoDete = NULL;
        noviCvor->podatak = vrednost;
        *stablo = noviCvor;
        return;
    }

    if (vrednost < (*stablo)->podatak)
    {
        dodajCvor(&(*stablo)->levoDete, vrednost);
    }
    else if (vrednost > (*stablo)->podatak)
    {
        dodajCvor(&(*stablo)->desnoDete, vrednost);
    }
}
```

Binarno stablo – Zadatak I

```
void preorderObilazak(tCvorBinarnogStabla *stablo)
{
    if (stablo)
    {
        printf("%d\n", stablo->podatak);
        preorderObilazak(stablo->levoDete);
        preorderObilazak(stablo->desnoDete);
    }
}

void inorderObilazak(tCvorBinarnogStabla *stablo){
    if (stablo){
        inorderObilazak(stablo->levoDete);
        printf("%d\n", stablo->podatak);
        inorderObilazak(stablo->desnoDete);
    }
}
```

Binarno stablo – Zadatak I

```
void postorderObilazak(tCvorBinarnogStabla *stablo)
{
    if (stablo)
    {
        postorderObilazak(stablo->levoDete);
        postorderObilazak(stablo->desnoDete);
        printf("%d\n", stablo->podatak);
    }
}

void obrisiStablo(tCvorBinarnogStabla *stablo)
{
    if (stablo)
    {
        obrisiStablo(stablo->levoDete);
        obrisiStablo(stablo->desnoDete);
        free(stablo);
    }
}
```

Binarno stablo – Zadatak I

```
tCvorBinarnogStabla* trazi(tCvorBinarnogStabla **stablo, int vrednost)
{
    if (!(*stablo))
    {
        return NULL;
    }
    if (vrednost < (*stablo)->podatak)
    {
        trazi(&((*stablo)->levoDete), vrednost);
    }
    else if (vrednost > (*stablo)->podatak)
    {
        trazi(&((*stablo)->desnoDete), vrednost);
    }
    else if (vrednost == (*stablo)->podatak)
    {
        return *stablo;
    }
}
```