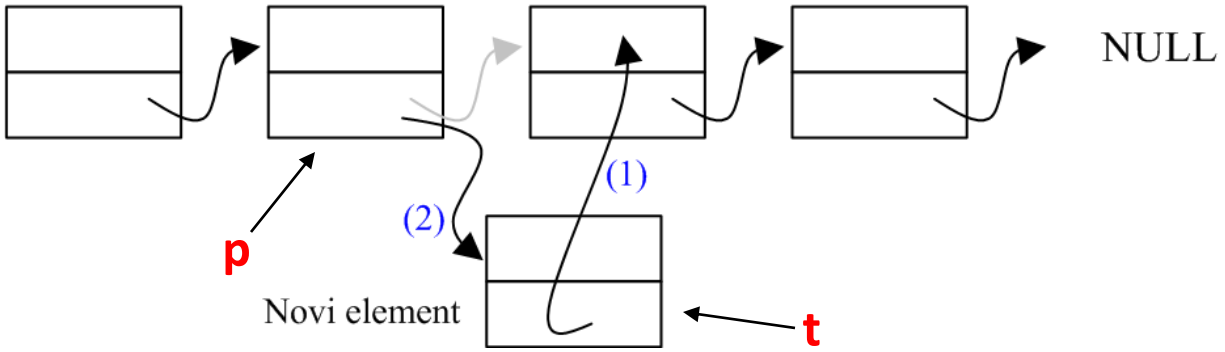
Programiranje I

Liste – Nastavak
Binarna drveta

Dodavanje elementa u sredinu liste

- Prvo grafički ilustrujmo šta se dešava u ovom slučaju:



- Treba odrediti poziciju (element liste na koji pokazuje pokazivač p) iza koje treba ubaciti novi element (na koji pokazuje pokazivač t).
- Zatim, pokazivač next iz elementa na koji pokazuje t treba usmjeriti na element koji se nalazi nakon elementa na koji pokazuje p (korak (1)).
- Konačno, pokazivač next iz elementa na koji pokazuje p treba preusmjeriti da pokaže na novododati element t (korak (2)).

Dodavanje elementa u sredinu liste

- Dio koda:
  t->next = p->next;
  p->next = t;

- **Vježba**: Pokušajte da napišete funkciju (bez gledanja u zbirku ☺) za koju važi sljedeće:

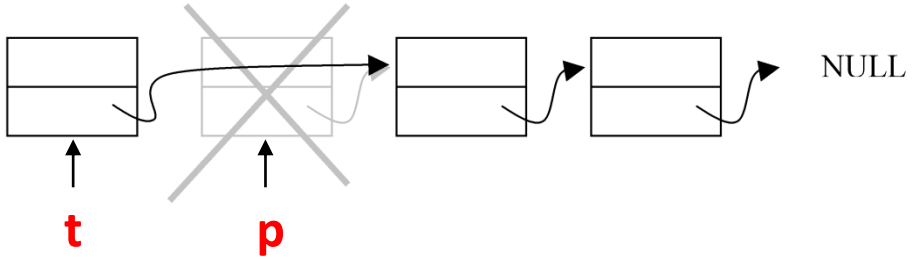
  Elementi (cijeli brojevi) upisani u čvorovima liste su sortirani u rastući redosljed. Napisati funkciju kojoj se prosljeđuju pokazivač na listu i cijeli broj, kojeg treba smjestiti u novi član liste postavljen tako da lista ostane sortirana. Funkcija vraća pokazivač na glavu liste.

Komentari problema umetanja

- Prva varijanta je da je predati element manji od elementa upisanog u glavi liste. U tom slučaju treba alocirati novi element, upisati u njega cijeli broj – argument funkcije, pa njegov pokazivač **next** usmjeriti na glavu liste. Tada novi element postaje glava liste i funkcija treba da vrati pokazivač na njega.
- Ako prva varijanta nije zadovoljena, krećemo u proceduru traženja prave pozicije za novi element liste. Sa srećom!

Brisanje elementa iz sredine liste

- Grafički se ovaj problem može prikazati na sljedeći način:



Procedura je relativno jednostavna. Treba pronaći element liste koji se briše. To se obavlja sa pozicije prethodnog elementa, na koji pokazuje pokazivač **t**. Zatim se zapamti pokazivač **t->next** (naredba **p=t->next**), a **t** se preusmjeri da pokaže na element nakon **t->next**, tj. **t->next = t->next->next**. Na kraju, vršimo dealociranje elementa na koji pokazuje **p** sa **free(p)**. Na ovaj način je "premošten" element koji se briše, a da bi se on obrisao moramo uvesti pomoćni pokazivač na njega (u našem slučaju **p**).

Zadatak za vježbu

- Kreirati funkciju kojoj se proslijeđuje glava liste i cijeli broj upisan u elementu liste kojeg treba obrisati. Lista je sortirana u neopadajući redosljed. Funkcija treba da pored brisanja elementa vrati glavu liste. U slučaju da traženi element ne postoji u listi ne treba vršiti brisanje. Treba predvidjeti i situaciju da je glava liste traženi element, kao i situaciju da je glava liste ujedno i rep liste i da je to traženi element (kad je lista prazna i nakon operacije brisanja treba vratiti NULL).

Brojanje elemenata liste – rekurzivno

- Da bismo ilustrovali rekurzivni rad sa listama, napišimo funkciju koja broji elemente liste. Radi olakšice, pretpostavimo da lista ima barem jedan element. Ako je taj element rep liste, funkcija treba da vrati 1. Ako element nije rep liste, funkcija treba da vrati 1 + broj_elemenata_u_ostatku_liste, a ostatak liste je opet lista koja počinje od narednog elementa.

```
int brojElemente(struct lista *glava) {
   if(glava->next == NULL)
        return 1;
   else
        return 1 + brojElemente(glava->next); }
```

- Napomena: Rekurzivna realizacija funkcije nije efikasna zbog velikog broj poziva funkcije. Iterativna realizacija je efikasnija.

Nadovezivanje listi – iterativno

- Imamo dvije liste i na kraj prve želimo da nadovežemo drugu. To znači da rep prve liste treba da pokaže na glavu druge liste. Nakon što odredimo rep prve liste, njegov pokazivač **next** treba preusmjeriti na glavu druge liste:

```
struct lista *nadovezi(struct lista *gl1, struct lista *gl2) {
    struct lista *pom = gl1;
    while(pom->next != NULL)
        pom = pom->next;
    pom->next = gl2;
    return gl1;
}
```

Uvodimo pomoćni pokazivač za kretanje kroz prvu listu.

Idemo na kraj prve liste.

Rep prve liste ukazuje na glavu druge liste.

Vraćamo glavu novonastale liste.

Specijalni tipovi listi

- Prvi specijalni tip liste je **stek** (eng. stack). U pitanju je lista kod koje se dodavanje novih elemenata i brisanje postojećih izvodi samo sa pozicije repa.
- Stek je FILO (First-In-Last-Out) memorija, odnosno član liste koji prvi uđe u listu posljednji iz nje izlazi, i obrnuto – posljednji uđe, prvi izađe.
- Operacije kod steka su:
  - **push** (dodavanje elementa),
  - **pop** (uklanjanje posljednjeg dodatog elementa) i
  - **peek** (čitanje posljednjeg dodatog elementa, bez modifikacije steka).
- Implementiraju se i operacije provjere da li je stek pun ili prazan.
- Osim preko listi, stekovi mogu realizovati i pomoću nizova.

Specijalni tipovi listi

- Drugi specijalni tip liste je **red** (eng. queue). Kod reda se podaci dodaju na početak liste, a brišu sa kraja liste.
- Kao takav, red je FIFO (First-In-First-Out) memorija, odnosno elementi koji prvi ulaze u listu iz nje se kao prvi i brišu, i obrnuto – oni koji posljednji uđu, posljednji izađu.
- Tri osnovne operacije kod reda su:
  - **enqueue** (dodavanje elementa na početak reda),
  - **dequeue** (uklanjanje elemenata sa kraja reda) i
  - **peek** (čitanje elementa sa kraja reda, bez njegovog uklanjanja).
- Ovdje, također, možemo implementirati operacije provjere da li je red pun ili prazan.
- Rad sa stekom i redom je jednostavniji nego rad sa jednostruko povezanom listom koja dozvoljava da se upis i brisanje vrše na svim pozicijama.

Dvostruko-povezana lista

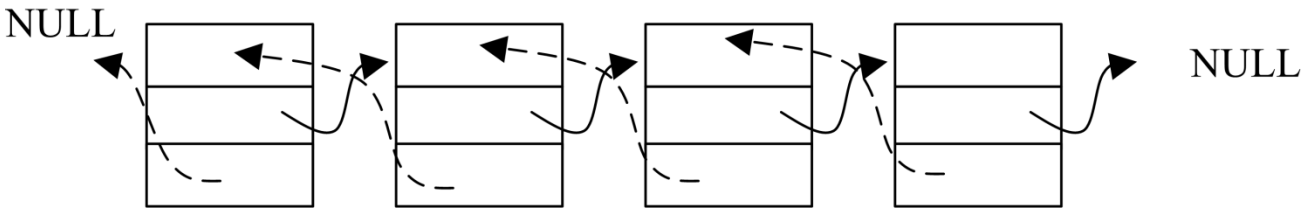
- Kvalitativno drugačiji tip liste je dvostruko-povezana lista. Kod ove liste čvorovi pamte, pored narednog, i prethodni čvor.
- Struktura pomoću koje se realizuje dvostruko-povezana lista se može deklarisati kao:

```
struct lista2pov {
    ... // podaci elementa liste
    struct lista2pov *next;
    struct lista2pov *prev;
};
```

Pokazivači na naredni i prethodni čvor liste

Vizuelizacija dvostruko-povezane liste

- Uobičajena grafička predstava dvostruko-povezane liste je:

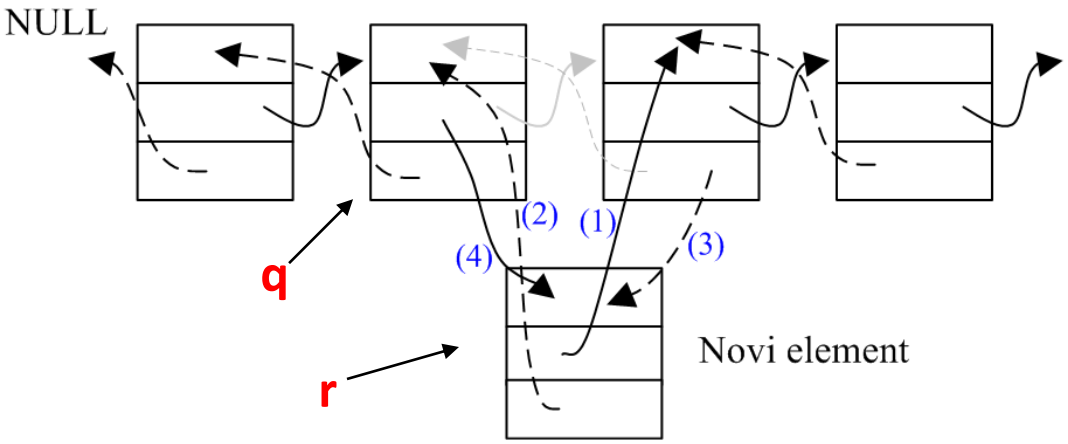


Punom linijom su označeni pokazivači na naredne elemente u listi, dok su isprekidanom označeni pokazivači na prethodne elemente. Ovaj tip liste ima dodatnu funkcionalnost, odnosno dozvoljava kretanje i unazad i unaprijed po listi.

Međutim, rad sa ovakvim tipom liste zahtjeva više pažnje, jer se mora voditi računa o dva pokazivača.

Dodavanje elementa u DP listu

- Vizuelizacija dodavanja elementa u dvostruko-povezanu (DP) listu:



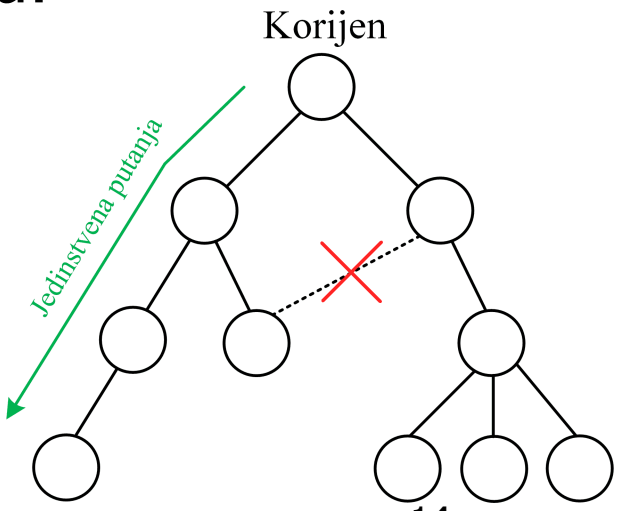
Neka je r pokazivač na element liste koji se dodaje, dok je q pokazivač na element liste iza kojeg se novi element dodaje.

```
r->next=q->next;     (korak (1))
r->prev=q;           (korak (2))
q->next->prev=r;     (korak (3))
q->next=r;           (korak (4))
```

Jednostavno pravilo kojega se treba držati je da prvo novi element uspostavi veze sa ostalim čvorovima u listi, pa da se tek onda prekidaju i preusmjeravaju stare veze.

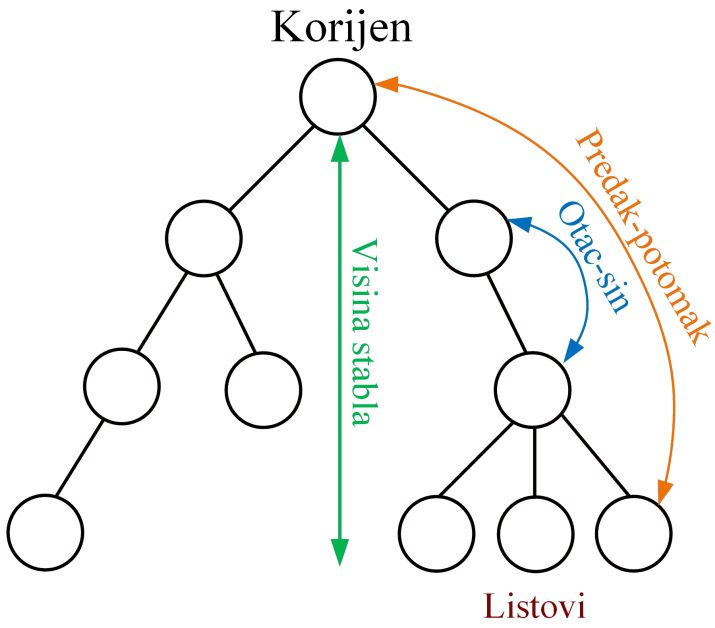
Stablo (drvo)

- Još jedan, moramo priznati, čudan naziv u programiranju.
- **Stablo** ili **drvo** je veoma napredan i veoma korišćen tip podataka.
- **Stablo** je usmjereni aciklični graf (graf u kome ne postoji ciklus) koji zadovoljava sljedeća svojstva:
  - Jedan čvor, koji se naziva korijen, nije kraj nijedne ivice;
  - Svakom čvoru, osim korijena, odgovora tačno jedna ivica čiji je kraj taj čvor;
  - Postoji jedinstvena putanja od korijena ka svakom čvoru drveta.



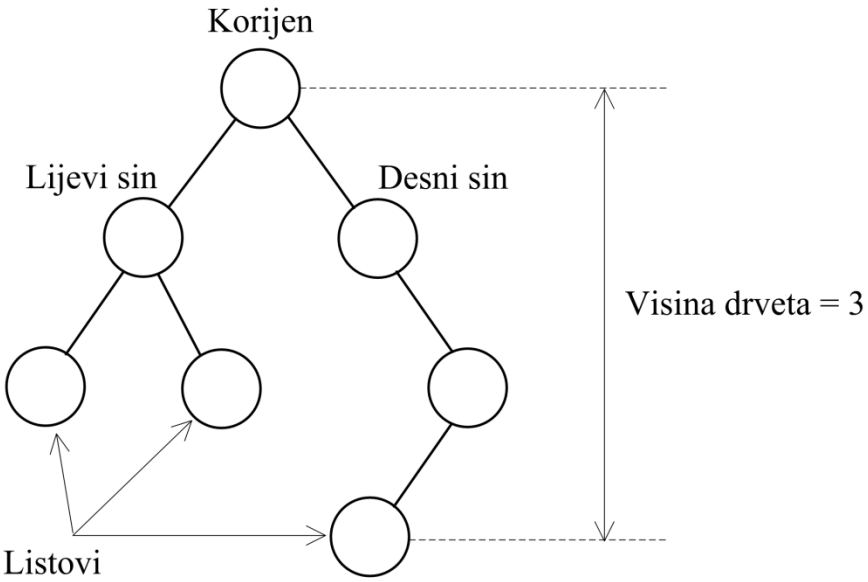
Stablo – Pojmovi

- Stablo je predstavljeno kao uređeni par čvorova i ivica: **T=(V,E)**. Ako (v,w)∈E, tada se kaže da je v **otac** čvoru w, odnosno da je w **sin** čvora v. Ako postoji putanja od v ka w preko proizvoljno mnogo ivica, kaže se da je v **predak** čvora w, odnosno da je w **potomak** čvora v. Čvorovi bez potomaka se nazivaju **listovima**. Čvor v i njegovi potomci čine podstablo čiji je korijen v.
- **Visina stabla** je dužina najdužeg puta od korijena ka listovima.



Binarno stablo

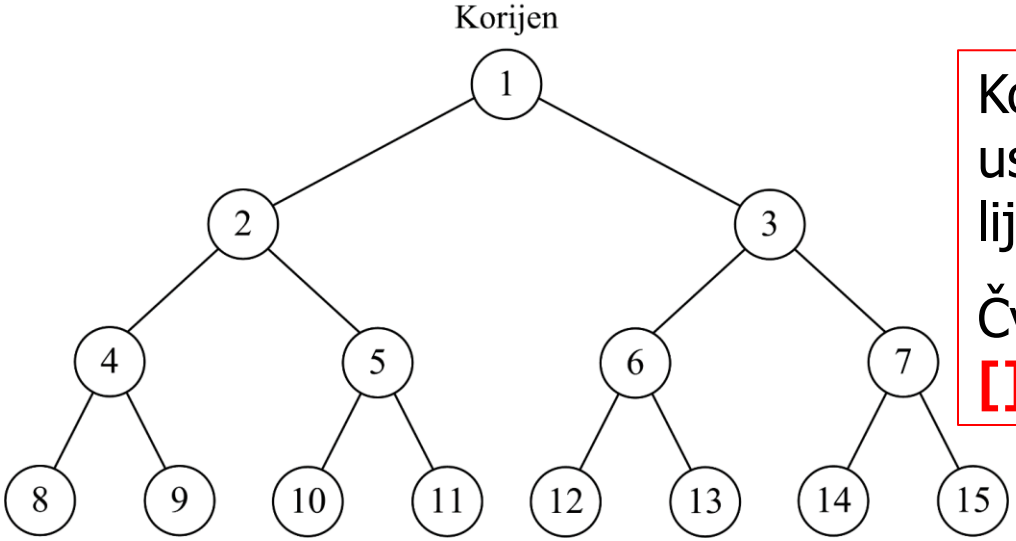
- Binarno stablo je specijalni i najčešće korišćeni tip stabla. U njemu svaki čvor ne može da ima više od dva sina od kojih se jedan naziva lijevi sin, a drugi desni sin (ovo implicira moguće postojanje lijevog i desnog podstabla).



Vizuelizacija jednog binarnog stabla sa ilustracijom nekih od važnih pojmova (korijen, listovi, lijevi i desni sin, i visina stabla).

Potpuno binarno stablo

- Kod **potpunog binarnog stabla**, svi čvorovi osim listova imaju oba sina i svi listovi su na istom rastojanju od korijena.
- Potpuno binarno stablo visine 3 ima 15 čvorova. Koliko čvorova ima potpuno binarno stablo visine 4? Koliko stablo visine n?

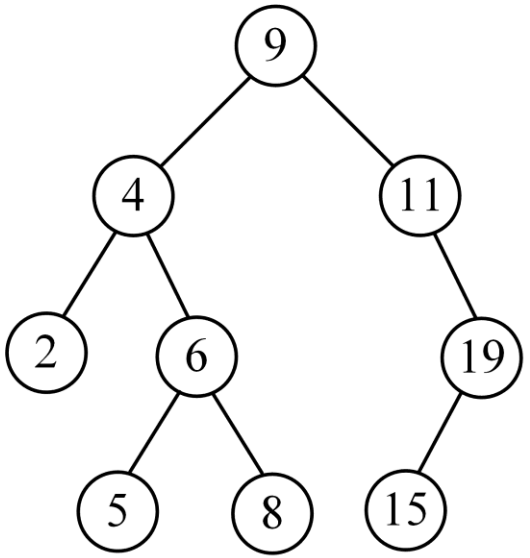


Kod potpunog binarnog stabla usvaja se konvencija da čvor **I** ima lijevog sina **2I** i desnog sina **2I+1**.

Čvoru **J** otac je čvor **[J/2]**, gdje je **[]** operator zaokruživanja nadolje.

Binarno stablo pretrage

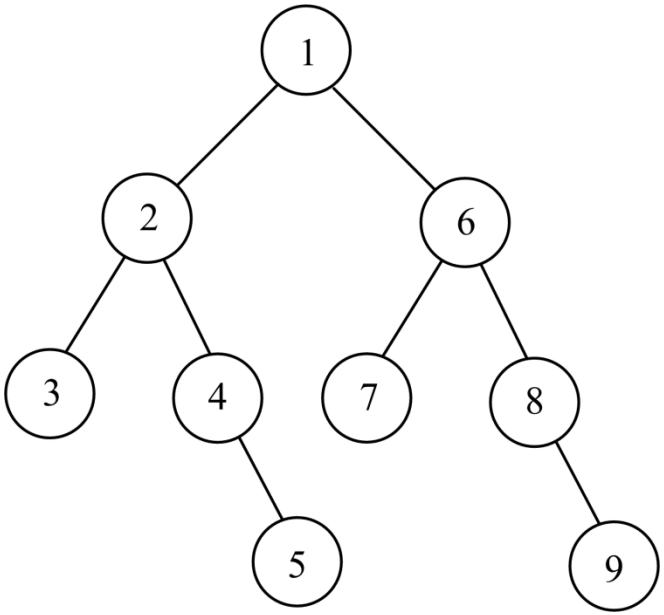
- Kod binarnog stabla pretrage (BSP), poznatog i kao sortirano binarno stablo, vrijednost svakog čvora je veća od vrijednosti svakog čvora u njegovom lijevom podstablu i manja od vrijednosti svakog čvora u njegovom desnom podstablu.
- Postoje realizacije BSP-a sa i bez ponavljanja elemenata.
- Novi čvorovi se dodaju kao listovi.
- Prilikom brisanja, može se desiti nekoliko različitih situacija. Istražite sami.
- BSP ostaje sortirano prilikom dodavanja i brisanja čvorova, što omogućava bržu pretragu nego većina drugih struktura.
- Složenost pretrage je O(V), gdje je V visina stabla.



Predstava stabla preko podnizova

- Čvorovi stabla se mogu predstaviti preko struktura. Kako se mogu predstaviti veze između čvorova stabla?
- Jedan od načina predstavljanja stabla je preko nizova LEFTSON i RIGHTSON. Ovi nizovi imaju elemenata koliko je čvorova u stablu.
- **LEFTSON[I]** predstavlja lijevi sin čvora **I**, dok **RIGHTSON[I]** predstavlja desni sin čvora **I**. U slučaju da čvor **I** nema nekog od sinova u odgovorajućem nizu se može upisati **0** (u C-u je pogodniji upis **–1**, pošto numerisanje čvorova može da se obavi kao kod nizova, od **0** pa na dalje).

Predstavljanje preko podnizova



	LEFTSON	RIGHTSON
1	2	6
2	3	4
3	0	0
4	0	5
5	0	0
6	7	8
7	0	0
8	0	9
9	0	0

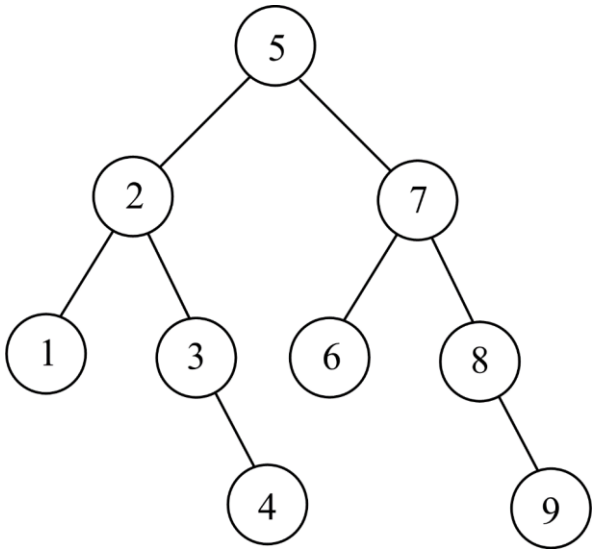
- **Tumačenje**: Čvor 1 ima dva sina od kojih je 2 lijevi, a 6 desni; čvor 2 ima oba sina i to čvor 3 lijevi i čvor 4 desni; čvor 3 je list (nema sinova) itd.

Obilazak stabla

- Algoritmi nad stablom često podrazumijevaju da se svaki čvor stabla mora obići jednom. Stoga se mora definisati jedinstvena metodologija za obilazak stabla (eng. tree traversal).
- Postoje tri metodologije obilaska:
  - inorder (srednji redosljed)
  - preorder
  - postorder

Obilazak po srednjem redosljedu

- Obilazak stabla po srednjem redosljedu (inorder) se obavlja:
  - prvo se obiđe lijevo podstablo,
  - zatim se obiđe korijen podstabla i
  - na kraju se obiđe desno podstablo.



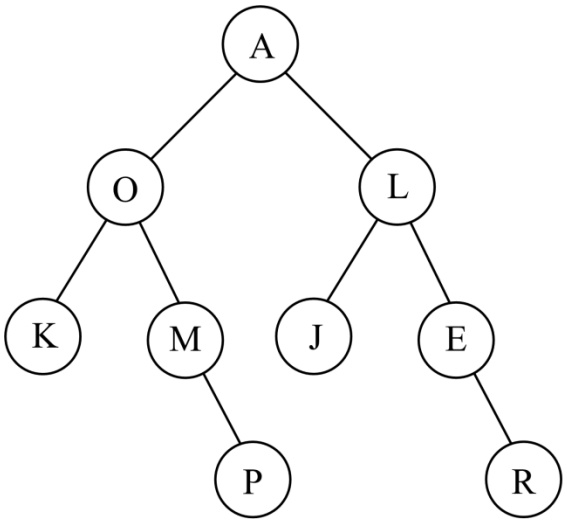
Prvo se obiđe lijevo podstablo korijena (5). Pošto lijevi sin korijena, tj. čvor (2), ima svoje lijevo podstablo, idemo dalje, tj. obilazimo njegovo lijevo podstablo. U njegovom lijevom podstablu se nalazi list (1), i pošto (1) nema svoje lijevo podstablo, obilazimo prvo njega. Dakle, prvi čvor koji smo obišli je (1). Sa njim završavamo obilazak lijevog podstabla čvora (2), pa obiđemo čvor (2), i prelazimo na njegovo desno podstablo. Pošto čvor (3) nema lijevo podstablo, obilazimo čvor (3), pa prelazimo na njegovo desno podstablo. Čvor (4) je list – obiđemo i njega. Sami nastavite dalje.

Preorder i postorder obilasci

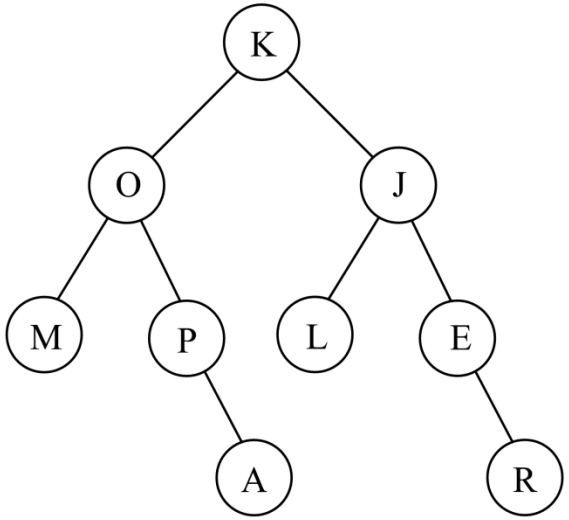
- Preorder obilazak:
  - prvo se obiđe korijen podstabla,
  - zatim se obiđe lijevo podstablo i
  - na kraju se obiđe desno podstablo.
- Postorder obilazak:
  - prvo se obiđe lijevo podstablo,
  - zatim se obiđe desno podstablo i
  - na kraju se obiđe sam korijen podstabla.

Obilasci – Primjer

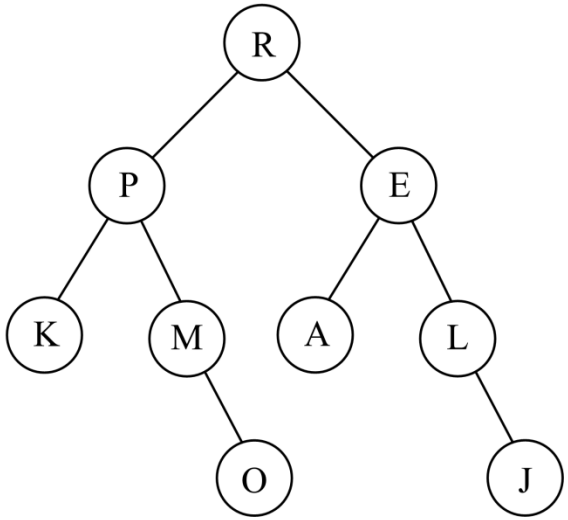
- Slova riječi KOMPAJLER su postavljena u stablu po odgovarajućim obilascima. Provjerite!



a) Inorder obilazak



b) Preorder obilazak



c) Postorder obilazak

Drvo preko samoreferentne strukture

- Drugi način za realizaciju veza između čvorova stabla je preko samoreferentne strukture. Ova struktura sada mora imati dva pokazivača koji pokazuju na lijevog i desnog sina (kod binarnog drveta). Primjer strukture **drvo**:

```
struct drvo {
        int i;                      // i ostali podaci članovi
        struct drvo *left;          // pokazivač na lijevog sina
        struct drvo *right;         // pokazivač na desnog sina
}
```

- Ukoliko drvo nema nekog od sinova odgovarajući pokazivač je NULL. Svi algoritmi koji rade sa drvetom polaze od korijena; stoga se pokazivač na korijen prosljeđuje funkcijama za rad sa drvetom.

Formiranje drveta preko struktura

- U programu koji formira drvo potrebno je imati nekoliko pokazivača na strukture tipa **drvo**. Na primjer:
  struct drvo *p, *r, *q, *root;
- Alocira se memorija za korijen (preskačemo provjeru):
  p = (struct drvo *) malloc(sizeof(struct drvo));
- Upišu se podaci u ovaj element:
  p->i = 4; p->left = NULL; p->right = NULL;
- Ovaj čvor možemo proglasiti korijenom:
  root = p;
- Svi algoritmi koji rade sa drvetom polaze od korijena. Stoga korijen treba trajno memorisati.

Formiranje drveta preko struktura

- Alocirajmo memoriju za dva sina korijena, ako ih korijen ima:
  q = (struct drvo *) malloc(sizeof(struct drvo));
  t = (struct drvo *) malloc(sizeof(struct drvo));
- Ponovo smo izostavili provjeru alokacije (vi nemojte!). Korijen (root, odnosno p) sada pokazuje na sinove:
  p->left = q; p->right = t;
- U sinove su upišu podaci i oni sada pokažu na NULL:
  q->i = 7; q->left = NULL; q->right = NULL;
  t->i = 1; t->left = NULL; t->right = NULL;
- Nastaviti sa formiranjem drveta, počevši od korijena poddrveta q i t.

Određivanje visine drveta

- Uradimo nekoliko primjera koji ilustruju rad sa drvetom.
- Prvi problem je određivanje visine drveta. Ovdje imamo četiri slučaja:
  - Korijen nema sinova. Visina je 0.
  - Korijen ima samo lijevog sina. Visina je jednaka 1 plus visina lijevog podstabla.
  - Korijen ima samo desnog sina. Visina je jednaka 1 plus visina desnog podstabla.
  - Korijen ima oba sina. Visina je jednaka 1 plus visina višeg od dva podstabla.
- Mnoštvo obrada drveta ima sličnu strukturu sa modifikacijama koje se primjenjuju u prethodna četiri slučaja.

Visina drveta zadatog preko nizova

```
#include <stdio.h>
int visina(int i)
int max(int, int);
int n, le[100], ri[100];

int main() {
   int j;
   printf("Unijeti broj cvorova");
   scanf("%d", &n);
   printf("Unijeti LEFTSON i RIGHTSON nizove ");
   for(j=1; j<=n; j++)
       scanf("%d%d", le+j, ri+j);
   printf("%d", visina(1));
}
```

```
int visina(int i) {
   int l, r;
   l = le[i];
   r = ri[i];
   if(l + r == 0)
       return 0;
   else if(l == 0)
       return 1 + visina(r);
   else if(r == 0)
       return 1 + visina(l);
   else
       return 1 + max(visina(l), visina(r));
}

int max (int p, int q) {
    if(p>q)  return p;
    else  return q;
}
```

Visina drveta – Komentar

- Iako ćemo vam prepustiti da sami protumačite veći dio ovog programa, dajemo vam nekoliko komentara.
- Funkcija int visina(int i) daje visinu podstabla čiji je korijen čvor i.
- Kako glavni program poziva funkciju za čvor **1** (korijen) to funkcija vraća rezultat koji je jednak visini kompletnog drveta.
- Pored prethodno opisanog pravila vezanog za određivanje visine drveta, naš program počiva i na činjenici da smo niz lijevih sinova i niz desnih sinova memorisali kao globalne promjenljive.
- Tumačite ostatak kôda sami, a pokušajte i da prepravite realizaciju da radi u slučaju kada ne želimo da koristimo globalne promjenljive.

Visina drveta – Preko struktura

- Pretpostavljamo da je drvo već formirano i to preko struktura. Napišimo funkciju koja određuje visinu drveta. Funkciji se prosljeđuje pokazivač na korijen:

```
int visina(struct drvo *cvor) {
    if(cvor->left == NULL && cvor->right == NULL)
        return 0;
    else if(cvor->left == NULL)
        return 1 + visina(cvor->right);
    else if(cvor->right == NULL)
        return 1 + visina(cvor->left);
    else
        return 1 + max(visina(cvor->left), visina(cvor->right));
}
```

Inorder štampanje čvorova drveta

- Da li se prethodni program mogao uraditi ako ispitujemo da li je aktuelni čvor NULL pokazivač?
- Funkcije koje rade sa drvetom preko strukture intenzivno koriste rekurziju.
- Primjer efikasne rekurzije može biti funkcija koja po inorder obilasku obilazi i štampa sadržaj svih čvorova drveta:

```
void print_drvo(struct drvo *cvor) {
        if(cvor->left!=NULL)    print_drvo(cvor->left);
        printf("%d\n", cvor->i);
        if(cvor->right!=NULL)  print_drvo(cvor->right);
}
```

Drvo – Za vježbu

- Za vježbu odraditi sljedeće probleme kod drveta:
  - Napisati funkciju koja formira potpuno binarno drvo.
  - Napisati funkciju koja dealocira drvo polazeći od korijena.
  - Napisati funkciju koja mjeri težinu drveta (broj čvorova drveta).
  - Napisati funkciju koja određuje da li je drvo potpuno binarno.
  - Napisati funkciju koja određuje koliko drvo ima listova.

Drvo – Za vježbu

- Napisati funkciju koja određuje najtežu putanju od korijena do lista. Najteža putanja je ona koja daje najveću vrijednost sume brojeva upisanih u čvorovima na svom putu.
- Napisati funkciju koja određuje najtežu prosječnu putanju od korijena ka listovima. To je ona putanja koja daje najveću vrijednost sume brojeva upisanih u čvorovima na tom putu podijeljenu sa brojem čvorova na tom putu.
- Drvo je sortirano inorder. Napisati funkciju koja vraća pokazivač na čvor u kome je upisan traženi broj ili NULL ako traženi broj ne postoji u drvetu.
- Napisati funkciju koja određuje širinu drveta. To je najveći broj čvorova koji se nalazi na određenoj distanci od korijena.
- Sve funkcije napisati za drvo predstavljeno preko nizova i samoreferentnih struktura.

KURS – ZAKLJUČAK

- Studenti nakon ovog kursa treba da su ovladali sljedećim pojmovima:
  - Elementarni tipovi podataka i operacije (80%)
  - Nizovi i pokazivači (60%)
  - Stringovi (80%)
  - Matrice (30%)
  - Funkcije (poziv po vrijednosti i referenci) (70%)
  - Funkcije - napredne opcije (30%)
  - Fajlovi (50%)
  - Strukture (70%)
  - Napredni linkovani tipovi podataka (40%)

procenti predstavljaju procijenjeni značaj za programersko opismenjavanje

KURS – ZAKLJUČAK

- Studenti koji su savladali ovaj kurs se mogu smatrati programerski pismenim, a to dalje znači da mogu da nastave sa usavršavanjem u oblasti programiranja.

- **Sa srećom!**