Sadržaj

1. Strukture podataka

Strukture podataka su općenito gradivni blokovi od kojih su algoritmi sastavljeni. One su u suštini sastavljene od struktura za pohranu podataka, metoda za kreiranje, modifikovanje i pristup podacima. Formalno govoreći, strukture se sastoje iz tri dijela:

- operacija za manipulisanje određenim tipovima apstraktnih objekata;
- dijelova za pohranu u kojima se čuvaju apstraktni tipovi podataka;
- implementacije svake operacije u pogledu strukture za pohranu podataka.

Operacije koje se izvode nad apstraktnim tipovima objekata nazivaju se *apstraktni tipovi podataka*. Nasuprot tome, strukture za pohranu podataka, zajedno s njihovom implementacijom, nazivaju se *strukture podataka za implementaciju*. Na primjer, niz predstavlja apstraktni tip podataka koji omogućuje manipulaciju skupom vrijednosti istog tipa, uključujući operacije pristupa i izmjene elemenata pomoću njihovih indeksa. U programskom jeziku C++ nizovi cijelih brojeva se pohranjuju u kontinualnim memorijskim blokovima, dok se pokazivačka aritmetika koristi za određivanje adrese svakog elementa u memoriji računara. Uvođenjem apstraktnih tipova podataka jasno se definiše vrsta strukture, kao i operacije koje podržava, bez ulaska u detalje implementacije u memoriji. Ovakav pristup dopušta programerima da se fokusiraju na rad sa apstraktnim tipovima podataka umjesto na rješavanje problema na niskom nivou, kao što su greške u memoriji, što značajno olakšava proces otklanjanja grešaka. Preporučuje se odvajanje algoritma od struktura podataka koje ga implementiraju, što je poznato kao *apstrakcija podataka*. Ova apstrakcija omogućuje razdvajanje različitih nivoa razmišljanja, gdje se, na primjer, cjelobrojni tip podataka može posmatrati kroz operacije sabiranja, množenja i poređenja, bez potrebe za razumijevanjem kako su brojevi predstavljeni ili kako se aritmetika izvodi na nivou računara. Modularni pristup uvođenjem apstraktnih tipova podataka osigurava razdvajanje programa u module sa jasno definisanim interfejsima. Modularnost olakšava nezavisan rad različitih razvojnih timova i omogućuje zamjenu modula boljim, robusnijim verzijama bez uticaja na ostatak sistema. Štaviše, izbor odgovarajuće strukture podataka direktno utiče na efikasnost algoritma i lakoću njegovog razumijevanja i implementacije. Objektno-orijentisani jezici poput C++-a i Jave pružaju raznovrstan skup predefinisanih struktura podataka, uključujući cijele brojeve, floating-point brojeve, karaktere, pokazivače i reference, koje se mogu koristiti samostalno ili u kombinaciji za kreiranje složenijih struktura podataka. Ovaj udžbenik se fokusira na korištenju programskog jezika C++ za rješavanje geometrijskih problema, pri čemu se za dizajn grafičkih sučelja koristi Visual C++ integrisano razvojno okruženje (IDE) [34, 43, 83]. U nastavku će biti predstavljeni ključni apstraktni tipovi podataka, poput *lista, stekova, binarnih stabala pretraživanja, upletenih stabala pretraživanja, randomiziranih stabala pretraživanja* i *riječnika*, kao specijalnog slučaja [61, 62]. Nji-

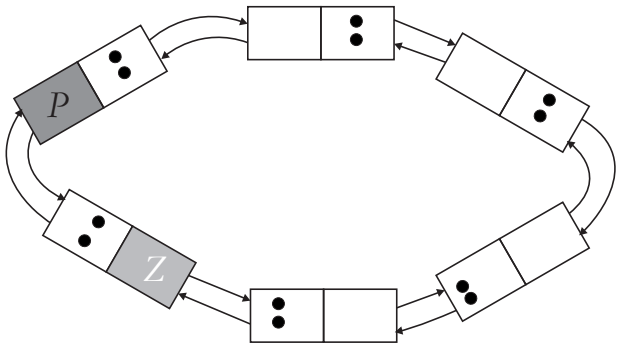
hova efikasnost čini ih pogodnim za rješavanje raznovrsnih geometrijskih problema, koji će biti detaljno istraženi i implementirani u ovom udžbeniku. Poznavanje ovih tipova obezbjeđuje opis problema na visokom nivou, dok razumijevanje njihove implementacije pruža fleksibilnost za istraživanje i primjenu alternativnih struktura tokom rješavanja novih problema.

1.1 Povezane liste

Jednostruko povezana lista (eng. *Singly Linked List*, **SLL**) predstavlja dinamičku strukturu podataka koja se sastoji od malih kontejnera nazvanih čvorovi, dizajniranih tako da se mogu fleksibilno povezivati prema potrebi. Čvorovi su sekvencijalno povezani, gdje svaki čvor sadrži podatke i referencu na sljedeći čvor. Prvi čvor u listi naziva se **glava** (eng. *Head*), dok je posljednji poznat kao **rep** (eng. *Tail*). Posljednji čvor obično nema referencu na sljedeći element, što se označava **nul-pokazivačem** (eng. *nullptr*). U literaturi se za nul-pokazivač često koristi oznaka **NULL**, što će se koristiti u nekim poglavljima ovog udžbenika. Kada je lista prazna, posljednji čvor može pokazivati na samog sebe. Za razliku od drugih tipova listi, kod jednostruko povezanih listi svaki čvor pokazuje samo na sljedeći čvor, bez referenci na prethodne elemente. Da bi se pronašao određeni čvor, pretraga uvijek počinje od glave liste i nastavlja se sekvencijalno, što podsjeća na „lov na blago“. Grafički prikaz jednostruko povezane liste dat je na slici 1.1 a).

Slika 1.1: a) Jednostruko povezana lista. b) Dvostruko povezana lista.

Prednosti jednostruko povezane liste, kao dinamičke strukture podataka, u poređenju sa fiksnim kontejnerima poput nizova, ogledaju se u brzom i jednostavnom ažuriranju poretka čvorova. Za ažuriranje je potrebno konstantno vrijeme reda $\mathcal{O}(1)$. Na primjer, kada je potrebno dodati novi čvor B iza čvora A, alocira se novi prostor za čvor B, zatim se pokazivač čvora A preusmjerava na čvor B, a pokazivač čvora B na čvor na koji je čvor A prethodno ukazivao. Ove operacije se izvode u konstantnom vremenu. Nasuprot tome, kod nizova, ubacivanje novog elementa na određeno mjesto zahtijeva pomjeranje elemenata. Ako se u nizu od n elemenata želi ubaciti novi element A na i-to mjesto $(0 \leq i < n)$, potrebno je pomjeriti $n-i$ elemenata udesno. U slučaju dodavanja elementa na početak niza, svaki član mora biti pomjeren za jedno mjesto, što u najgorem slučaju zahtijeva $\mathcal{O}(n)$ vremena. Jednostruko povezana lista ima značajnu prednost u situacijama kada se broj elemenata unaprijed ne zna ili kada se broj elemenata dinamički mijenja tokom trajanja programa. Kod fiksnih kontejnera poput nizova, unaprijed se mora alocirati memorijski prostor, što može biti neefikasno. Na primjer, niz od 100.000 elemenata, iako možda neiskorišten, rezervisani memorijski prostor neće dopustiti drugim procesima pristup. Ova rigidnost predstavlja značajan nedostatak nizova, zbog čega ih treba izbjegavati kada je moguće, radi optimizacije performansi i resursa. Kod jednostruko povezane liste svaki čvor koji pokazuje na sljedeći čvor naziva se **sljedbenik** (eng. *successor*). Kod kružne jednostruko povezane liste, posljednji čvor pokazuje na prvi čvor. Pored jednostruko povezane liste, često se koristi i dvostruko povezana lista, čija je grafička ilustracija prikazana na slici 1.1 b). Kod dvostruko povezane liste, svaki čvor je povezan i sa prethodnim i sa sljedećim čvorom. Čvor koji pokazuje na prethodni čvor naziva se **prethodnik** (eng. *predecessor*). U kompjutacionoj geometriji, najčešće se koristi **dvostruko povezana kružna lista** (eng. *Doubly Linked Circular List*, **DLCL**) , čija je grafička ilustracija prikazana na slici 1.2.
U daljem tekstu koriste se kružne dvostruko povezane liste, koje će radi jednostavnosti biti nazivane jednostavno listama. Ove liste se najčešće koriste za reprezentaciju poligona, pri čemu čvorovi liste



Slika 1.2: Dvostruko povezana kružna lista.

predstavljaju vrhove poligona. Za potrebe implementacije, svaki čvor biće predstavljen kao instanca, odnosno objekat klase **Cvor**, što je prikazano u Listingu1.1.

Listing 1.1: Definicija klase **Cvor**

```
class Cvor{
 protected:
   Cvor*sljedeciC;//pokazivac na sljedbenik
   Cvor*prethodniC;//pokazivac na prethodnik
 public:
  Cvor();
  virtual~Cvor(){};
  Cvor*sljedeci();
  Cvor*prethodni();
  Cvor*ubaci(Cvor *);
  Cvor*ukloni();
  void povezi(Cvor *);
};
```

Listing 1.1 prikazuje definiciju klase **Cvor**. U nastavku se implementiraju konstruktor, destruktor i ostale metode ove klase. Prilikom implementacije konstruktora, potrebno je osigurati da instance klase inicijalno upućuju na same sebe. Ovo se postiže inicijalizacijom privatnih članova, odnosno atributa `sljedeciC` i `prethodniC`, na vrijednost **this**. Implementacija konstruktora data je ispod:

```
//Implementacija konstruktora klase Cvor.
Cvor::Cvor():prethodniC(this),sljedeciC(this){};
```

Jasno je da unutar konstruktora, pokazivač **this** će pokazivati na objekat koji se kreira. Na primjer, za ovakvo instanciranje:

```
Cvor *A(new Cvor);
```

pokazivač **this** se odnosi na objekat *A*, pa je sljedeći i prethodni čvor od čvora *A*, zapravo on sam. Destruktor je zadužen da dealocira kreirane objekte ili da reaguje prilikom poziva operatora **delete**. Konkretno, u gornjem slučaju, destruktor je zadužen da dealocira objekat *A*. Razlog zbog čega je stavljeno da destruktor za klasu **Cvor** bude virtualni, leži u činjenici da isti treba osloboditi objekte odnosno instance klasa koje će naslijediti klasu **Cvor**, kao npr. klasa **Lista** koja će biti definisana i implementirana u nastavku. Da bi se omogućio "skok" sa jednog čvora na drugi, u nastavku su date dvije implementacije funkcija članica *prethodni*() i *sljedeci*().

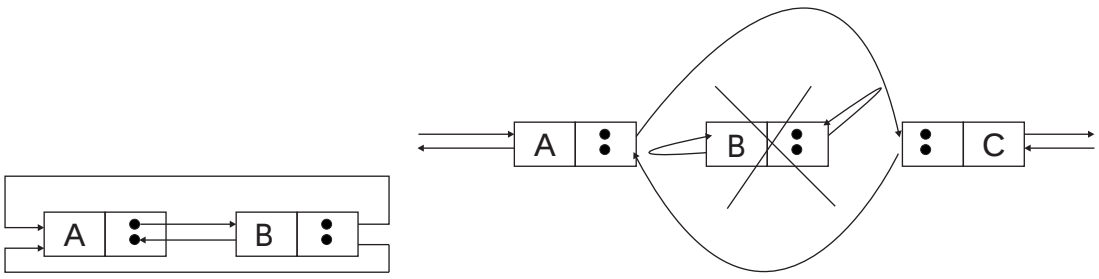
```
//Implementacija metoda prethodni i sljedeci.
Cvor*Cvor::prethodni(){
 return prethodniC;
}
Cvor*Cvor::sljedeci(){
 return sljedeciC;
}
```

Za ubacivanje novog čvora u listu (dvostruko povezana kružna lista) zadužena je metoda *ubaci*(·). Njeno funkcionisanje odnosno dodavanje novog čvora *B* u listu se najbolje može objasniti ukoliko se posmatra slika 1.3 a), na kojoj se vidi da sljedbenik čvoru *A* treba biti čvor *B*, dok u isto vrijeme čvor *A* je prethodnik čvoru *B*. Također, sljedbenik čvora *B* je čvor *A*, dok je prethodnik čvora *A* čvor *B*. Na osnovu sadržaja slike, impelementacija metode *ubaci*(·) ovako izgleda:

```
//Implementacija metode ubaci
Cvor*Cvor::ubaci(Cvor*b){
 Cvor*a(sljedeciC);b->sljedeciC=a;b->prethodniC=this;
 sljedeciC=b;a->prethodniC=b;return b;
}
```

Slika 1.3 b) biće korištena za implementaciju metode *ukloni*(), kako bi se olakšalo objašnjenje procesa uklanjanja određenog čvora iz liste. Na osnovu prikazanog sadržaja na slici 1.3 b), implementacija spomenute metode postaje jednostavna, što je prikazano u donjem isječku koda:

```
//Implementacija metode ukloni
Cvor*Cvor::ukloni(){
 prethodniC->sljedeciC=sljedeciC;
 sljedeciC->prethodniC=prethodniC;
 sljedeciC=prethodniC=this;return this;
}
```

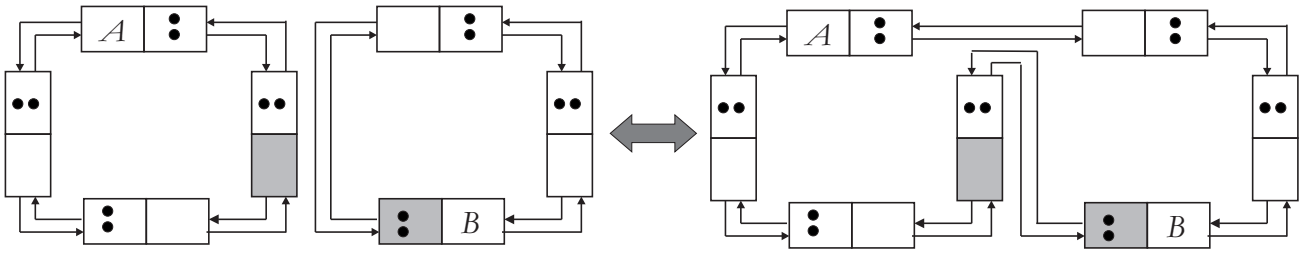


Slika 1.3: a) Ubacivanje čvora *B* u listu. b) Uklanjanje čvora *B* iz liste.

Preostalo je još da se implementira metoda *povezi*(·). Da bi se ona implementirala, biće korištena slika 1.4 a) na kojoj su zadate dvije liste disjunktnih čvorova, koje redom sadrže čvorove *A* i *B*. Spomenuti čvorovi koriste se za povezivanje listi, čime se kao rezultat dobija veća lista čvorova, što se vidi na slici 1.4 b). Na osnovu sadržaja prikazanog na slici 1.4, vidi se da je operacija povezivanja čvorova poopćenje operacija "ubacivanja" i "uklanjanja". Naime, ukoliko se čvorovi *A* i *B* nalaze u istoj listi (Slika 1.4 b)), tada brisanje čvora prouzrokuje generisanje dvije manje liste (Slika 1.4 a)), što je ustvari operacija uklanjanja čvorova.

Obrnuto, ako se čvorovi A i B nalaze u disjunktnim listama (Slika 1.4 a)), onda se nakon spajanja dobija jedna veća lista, što zaista predstavlja operaciju ubacivanja čvorova. Na osnovu rečenog, implementacija metode *povezi*(·) data je ispod:

```
//Implementacija metode povezi
void Cvor::povezi(Cvor*b){
 Cvor*a(this),*pom1(a->sljedeciC),*pom2(b->sljedeciC);
 a->sljedeciC=pom2;b->sljedeciC=pom1;
 pom1->prethodniC=b;pom2->prethodniC=a;
}
```



Slika 1.4: Povezivanje čvorova A i B iz disjunktnih listi.

Na osnovu implementacije metode *povezi*(·), primjećuje se da ukoliko čvor A prethodi čvoru B u povezanoj listi, operacija povezivanja utiče na uklanjanje čvora B. Ovaj zaključak vodi do saznanja da su metode za ubacivanje i uklanjanje čvorova samo specijalni slučajevi metode *povezi*(·). Također, ako bi se pokušala pozvati spomenuta metoda tako da se čvor A poveže sa samim sobom, rezultat ne bi imao nikakav koristan efekat.

1.2 Lista čvorova

U ovom odjeljku biće definisana i implementirana nova klasa pod nazivom **ListaCvorova**. Poznato je da lista predstavlja uređen skup konačno mnogo elemenata, pri čemu je njena dužina jednaka broju elemenata koje ona sadrži. Lista sa nula elemenata zove se **prazna lista** (eng. *Empty List*). Pretpostavlja se da svaki element liste zauzima jedno mjesto u memoriji, pri čemu se prvi element nalazi na prvom mjestu, drugi na drugom, i tako redom. Glava liste se u početku nalazi na nultom mjestu, tako da je uvijek ispred prvog elementa liste. Također, kada se dođe do kraja liste, glava se nalazi odmah iza posljednjeg elementa liste. Pristup elementima liste vrši se preko trenutnog pokazivača ili **prozora** (eng. *Window*). Pomoću trenutnog pokazivača, moguće je kretati se unutar liste, pomjerati se na prethodnu ili sljedeću poziciju, kao i ubacivati, brisati ili čitati sadržaj čvora na kojem se pokazivač nalazi. Također, putem njega može se doći do početka ili kraja liste. U nastavku slijedi definicija generičke klase **ListaCvorova**, koja javno nasljeđuje klasu **Cvor**.

Listing 1.2: Definicija generičkes klase **ListaCvorova**

```
template<class T>
class Lista;//najava kompajleru klase Lista
 //Definicija generickе klase ListaCvorova
template<class T>
class ListaCvorova:public Cvor{
 public:
   ListaCvorova(T);
   ListaCvorova();
```

```
    T podatak;//generic javni element
    friend class Lista<T>;
 };
```

Najavljena klasa **Lista** kasnije će biti deklarisana i djelimično realizovana kroz Listing 1.3. Kao što se vidi iz definicije klase **ListaCvorova**, atribut podatak označen je da bude javan odnosno **public** iz razloga izbjegavanja dodatnog pisanja *seter* i *geter* metoda za pristup njemu. Također, proglašeno je da klasa **Lista** bude prijatelj klasi **ListaCvorova**, kako bi se na taj način izbjeglo uvođenje dodatnih metoda za pristup ostalim atributima. Osim toga, uveden je generički tip podataka T, što dopušta klasi da efikasno upravlja različitim tipovima podataka prilikom rješavanja mnogobrojnih geometrijskih problema. U nastavku slijedi implementacija konstruktora kako sa parametrima, tako i bez njih.

```
//Implementacija konstruktora generic klase ListaCvorova
template<class T>ListaCvorova<T>::ListaCvorova(T e):podatak(e){}

template<class T>ListaCvorova<T>::ListaCvorova(){}
```

Jasno je da će konstruktor klase **ListaCvorova** prilikom inicijalizacije pozvati konstruktor klase **Cvor**. Za klasu **ListaCvorova** ne definiše se destruktor, jer će biti pozvan destruktor bazne klase **Cvor** prilikom uništavanja njenih objekata. Jedna upotreba klase **ListaCvorova** izgleda ovako:

```
ListaCvorova<int> * listaC(new ListaCvorova<int>);
```

Listing 1.3: Definicija generičke klase **Lista**

```
//Definicija generic klase Lista
template<class T>
class Lista{
   ListaCvorova<T>*glava;
   ListaCvorova<T>*prozor;//prozor nad trenutnim cvorom
   int broj;//duzina liste
 public:
   Lista();
   T prvi(){prozor=(ListaCvorova<T>*)glava->sljedeci();return prozor->podatak;}
   ~Lista(){while(broj>0){prvi();ukloni();} delete glava;}
   T zadnji(){prozor=(ListaCvorova<T>*)glava->prethodni();return prozor->podatak;}
   T unos(T el){prozor->ubaci(new ListaCvorova<T>(el));++broj; return el;}
   bool jelPrvi(){return((prozor==glava->sljedeci())&&(broj>0));}
   bool jelPosljednji(){return((prozor==glava->prethodni())&&(broj>0));}
   int duzina()const{return broj;}
   bool jelGlava()const{return prozor==glava;}
   T dodajNaPocetak(T el){glava->ubaci(new ListaCvorova<T>(el));++broj;return el;}
   void azuriraj(T el){if(prozor!=glava)prozor->podatak=el;}
   T dodajNaKraj(T el){
    glava->prethodni()->ubaci(new ListaCvorova<T>(el));++broj;return el;}
   T vrati(){return prozor->podatak;}
   T prethodni(){
    prozor=(ListaCvorova<T>*)prozor->prethodni();return prozor->podatak;}
   T sljedeci(){
    prozor=(ListaCvorova<T>*)prozor->sljedeci();return prozor->podatak;}
   T ukloni();
   Lista *spojiL(Lista*);
};
```

Prije pružanja implementacija metode klase **Lista**, uvodi se pravilo kojim se članovi liste dodaju u smjeru kretanja kazaljke na satu (**negativna orijentacija**), dok se čitaju u suprotnom smjeru (**pozitivna orijentacija**). U nastavku se daje implementacija konstruktora i ostalih metoda klase **Lista**, koje nisu implementirane na licu mjesta.

```
//Implementacija konstruktora genericke klase Lista
template<class T>
Lista<T>::Lista():broj(0),glava(new ListaCvorova<T>(NULL)){
 prozor=glava;//na pocetku prozor pokazuje na glavu
}
```

Nakon napravljenog konstruktora, primjer upotrebe klase Lista dat je ispod:

```
Lista<int>*lista(new Lista<int>());
```

Implementacija javne metoda *ukloni*() bazira se na brisanju trenutnog elementa liste, odnosno čvora na koji pokazuje atribut `prozor`. Njena izvedba data je niže:

```
//Implementacija metode ukloni()
template<class T>
T Lista<T>::ukloni(){
 if(prozor==glava)return NULL;
 auto el(prozor->podatak);
 prozor=(ListaCvorova<T>*)prozor->prethodni();
 delete(ListaCvorova<T>*)prozor->sljedeci()->ukloni();
 --broj; return el;
}
```

Nije teško uočiti na osnovu implementacije spomenute metode, da ona vraća element storiran u trenutnom čvoru, te da se pokazivač `prozor` preusmjerava da pokazuje na prethodni čvor onog čvora koji se briše. Metoda *spojiL*(·) služi za povezivanje elemenata dvaju listi u novu listu, koja se vraća kao rezultat. Njena implementacija izgleda ovako:

```
//Implementacija metode spojiL
template<class T>
Lista<T>*Lista<T>::spojiL(Lista<T>*l){
 ListaCvorova<T>*a((ListaCvorova<T>*)glava->prethodni());
 a->spoji(l->glava);broj+=l->broj;l->glava->ukloni();l->broj=0;
 l->prozor=glava;return this;
}
```

1.2.1 Primjeri upotrebe klase Lista

U ovom dijelu prikazuje se korištenje klase **Lista**. Pretpostavlja se da je definisana vrlo jednostavna klasa **Tacka**, kao što je prikazano u Listingu 1.4.

Listing 1.4: Definicija klase **Tacka**

```
class Tacka{
   double x,y;
 public:
   Tacka(){};
   Tacka(double, double){this->x=x;this->y=y;};
```

```
    double dajX()const{return x;}
    double dajY()const{return y;}
 };
```

Sljedeći primjer demonstrira dodavanje tačaka u listu, tako da se istovremeno tačke dodatno čuvaju preko niza pokazivača:

```
Lista<Tacka* >*l1(new Lista<Tacka* >());
Tacka ** p2(new Tacka*[5]);
for(int i(0);i<5;i++)p2[i]=l1->unos(new Tacka(i,i));
```

Čitanje elemenata iz liste obavlja se na ovaj način:

```
while(!l1->jelPosljednji()){
  Tacka *e{l1->sljedeci()};//pocevsi od C++11
  cout<<"("<<e->dajX()<<","<<e->dajY()<<")"<<endl;
}
```

Sasvim analogno se testiraju ostale metode. U nastavku se prikazuje kako se uz pomoć metode *spojiL*(·) spajaju liste "l1" i "l2" u novu listu "l3", pri čemu se kreiranje listi "l1" i "l2" obavlja na gore opisani način:

```
Lista<Tacka*>*l3(l2->spojiL(l1));
```

Nerijetko se u praksi zahtijeva pretvaranje elemenata niza u elemente liste. To se može obaviti pomoću generičke funkcije *nizUListu*():

```
template<class T>
Lista<T>*nizUListu(T a[], int);
```

Implementacija funkcije *nizUListu*() ovako izgleda:

```
template<class T>
Lista<T>*nizUListu(T a[],int n){
  Lista<T>*s(new Lista<T>);
  for (int i(0);i<n;i++) s->dodajNaKraj(a[i]);
  return s;
}
```

Napisana funkcija zbog svoje generičke prirode može se upotrijebiti kako za konvertovanje niza brojeva u listu, tako i za pretvaranje niza stringova, što je učinjeno ispod:

```
//Konvertovanje niza brojeva
double A[]{1,2,3,4,5};//Od C++11
Lista<double>*l(nizUListu(A,5));
while(!l->jelPosljednji())cout<<l->sljedeci()<<endl;
delete l;
```

```
//Konvertovanje niza ciji su elementi znakovne niske
char*B[]{"Huso","Haso", "Akif", "Mule"};
Lista<char*>*l{nizUListu(B,4)};//Od C++11
while(!l->jelPosljednji())cout<<l->sljedeci()<<endl;
delete l;
```

Još jedna veoma česta korelacija između nizova i listi odnosi se na traženje najmanjeg odnosno najvećeg elementa u listi. Da bi se ovo postiglo, dovoljno je implementirati funkciju *najmanjiElement*() za traženje najmanjeg elementa, dok se funkcija za traženje najvećeg elementa potpuno analogno ostvaruje:

```
template<class T>
T najmanjiElement(Lista<T>&l, int(*kriterijFun)(T,T)){
```

```
 if(l.duzina()==0)return NULL;
 T e(l.prvi());
 for(l.prvi(); !l.jelGlava(); l.sljedeci())
  if(kriterijFun(l.vrati(),e)<0)e=l.vrati();
  return e;
}
```

Ovako napisana funkcija može se iskoristiti da se pronađe najmanji element liste "l" iz prethodnog primjera, gdje se za niz "B" posmatraju konstantne niske. Za njihovo poređenje koristi se funkcija **strcmp** iz biblioteke **cstring**.

```
std::cout<<najmanjiElement(*l,strcmp);
```

Ukoliko se umjesto poziva kriterijFun(l.vrati(),e)<0, posmatra poziv kriterijFun(l.vrati(),e)>0, tada se kao rezultat dobija funkcija za traženje najvećeg elementa u listi.

1.3 Stek

Opće je poznato da su liste neograničene strukture podataka, jer se može bilo kojem njenom elementu pristupiti na proizvoljan način. Pored lista postoje strukture podataka čijim elementima se pristupa sa određenim ograničenjem. Takve strukture podataka se zovu **stek** (eng. *Stack*). Kod steka prvi član koji je dodan, posljednji se skida, dok zadnji dodan element, prvi se skida sa steka. Zbog ovoga, stekovi su poznati kao liste sa **LIFO strukturom** (eng. *Last In First Out*, **LIFO**). Dvije osnovne operacije kod steka su: **dodaj** (eng. *Push*) i **skini** (eng. *Pop*). Jasno je da operacija dodaj svaki put dodaje element na vrh steka, dok operacija skini jedino skida element sa vrha steka. Stoga, jedini element koji je dostupan na steku je onaj koji se nalazi na samom vrhu (prvi za skidanje) odnosno **najviši element** (eng. *Topmost Element*). Pored ove dvije operacije, postoje još operacije kao što su: jel prazan, veličina steka, najviši element, do najvišeg elementa, donji element, itd. Jasno je da se statička implementacija steka može realizovati preko niza. Međutim, problem kod ove implementacije jeste taj što bi dužina niza ograničavala dužinu steka. Stoga se umjesto niza kao fiksnog kontejnera koriste liste kao dinamičke strukture podataka. Koristeći liste za realizaciju steka, posljednji element liste predstavlja najviši element steka, dok prvi element liste predstavlja donji element (najniži element) steka odnosno prvi element koji je dodan na stek. Zbog jednostavnosti implementacije steka, u nastavku se kroz Listing 1.5 daje definicija klase **Stek**.

Listing 1.5: Definicija klase **Stek**

```
template<class T>
class Stek{
   Lista<T>*s;
 public:
   Stek(){s=new Lista<T>;}
   ~Stek(){delete s;}
   void dodaj(T e){s->dodajNaKraj(e);}
   T skini(){s->zadnji();return s->ukloni();}
   bool jelPrazan()const{ return(s->duzina()==0);}
   int velicina()const{return s->duzina();}
   T najvisi()const{return s->zadnji();}
   T doNajviseg(){s->zadnji();return s->prethodni();}
   T donji()const{return s->prvi();}
};
```

Jasno je da je klasa **Stek** apstraktni tip podataka kod koje je klasa **Lista** skrivena unutar njenih privatnih članica. Stoga, klasi **Lista** se ne može pristupiti izvana, niti se ista može modifikovati. Prema tome,

ona ima ulogu interfejsa. Funkcije članice koje koriste Stek, nemaju potrebe da vode računa kako je on implementiran odnosno da li je implementiran preko liste ili običnog niza. Jedan primjer upotrebe gornje klase dat je ispod:

```
char * A[4]{"aki", "maki", "laki", "raki"};//Od C++11
Stek<char *> s;
for(int i=0;i<4;i++) s.dodaj(A[i]);
for(int i=0;i<4;i++) A[i]=s.skini();
```

U gornjem primjeru, klasa **Stek** je iskorištena da se članovi niza A izokrenu naopačke. Stek se kao generička struktura podataka može koristiti za razne potrebe. Između ostalog, on će se koristiti kod Graham Scan algoritma za potrebe traženja konveksnog omotača.

1.4 Binarno stablo

Binarno stablo (eng. *Binary tree*) je usmjeren graf kod koga iz svakog čvora maksimalno izlaze dvije grane. Sem toga, binarno stablo se može posmatrati kao dinamički tip podataka koji dopušta da podaci u njemu uvijek budu sortirani i da efikasno podupire operacije ubacivanja, brisanja i traženja. Obilazak čvorova u binarnom stablu može se vršiti na tri načina:

- Inorder;
- Postorder;
- Preorder.

Kod Inorder-a, prvo se posjećuje **lijevo podstablo** (L), zatim **čvor** (Č), pa **desno podstablo** (D), tj. važi **L-Č-D** obilazak. Kod Postorder-a, prvo se posjećuje lijevo podstablo, zatim desno podstablo, pa čvor, tj. slijedi se **L-D-Č** obilazak. Kod Preorder-a, prvo se posjećuje čvor, zatim lijevo podstablo, pa tek onda desno podstablo, tj. vrijedi **Č-L-D** obilazak. U nastavku se opisuje način predstavljanja binarnog stabla u računaru. Binarno stablo predstavlja strukturiranu kolekciju čvorova koja može biti prazna, što znači da se radi o praznom binarnom stablu. Ako kolekcija nije prazna, tada se sastoji od tri disjunktna skupa čvorova: **korijenskog čvora** (eng. *Root Node*), koji čini jednočlani skup, lijevog podstabla, koje sadrži čvorove smještene na lijevoj strani stabla prema određenom kriteriju, te desnog podstabla, koje čine čvorovi s desne strane. Binarno stablo obično obuhvata interne i eksterne čvorove. Eksterni čvorovi su najčešće prazni i ne upućuju ni na šta. Veličina binarnog stabla definiše se brojem internih čvorova od kojih se stablo sastoji. Također, za neki čvor A kaže se da je roditelj čvora B ako je čvor B dijete čvora A. Štaviše, bilo koja dva čvora B i C su **braća i sestre** (eng. *Siblings*) samo ako dijele neki zajednički čvor A. Dodatno, ako za dva čvora A_1 i A_k vrijedi da čvor A_k pripada podstablu čiji je korijen čvor A_1, onda se kaže da je čvor A_k **potomak** (eng. *Descendant*) čvora A_1 odnosno da je čvor A_1 **predak** (eng. *Ancestor*) čvoru A_k. Za neki niz čvorova $A_1, A_2, \cdots, A_k$ kaže se da postoji **jedinstvena staza** (eng. *Unique Path*), koja omogućuje da se iz roditeljskog čvora A_1 stigne do potomaka A_l, pri čemu je čvor A_{l-1} roditelj čvoru A_l ($l = 2, 3, \cdots, k$). Razumije se da je **dužina staze** (eng. *Length of path*) jednaka broju ivica koje povezuju te čvorove i za gornji niz čvorova A_i ($i = 1, 2, \cdots, k$) jednaka je $k-1$. **Dubina čvora** (eng. *Depth of Node*) u oznaci $d(A)$ rekurzivno se definiše kao

$$d(A) = \begin{cases} 0, & \textit{ako je A korijenski čvor} \\ 1 + d(roditelj(A)), & \textit{inače} \end{cases}. \tag{1.1}$$

Na osnovu relacije 1.1 slijedi da je dubina čvora jednaka dužini jedinstvene staze od korijena pa do čvora do kojeg se traži dubina. **Visina čvora** (eng. *Height of Node*) u oznaci $h(A)$ rekurzivno se određuje kao

$$h(A) = \begin{cases} 0, & \textit{ako je A eksterni čvor} \\ 1 + max(h(lStablo(A)), h(dStablo(A))), & \textit{inače} \end{cases}. \tag{1.2}$$

Prema relaciji 1.2, visina čvora A predstavlja najdužu stazu koja vodi od čvora A do nekog eksternog čvora smještenog na kraju pripadajućeg podstabla. Visina binarnog stabla, pak, definiše se kao visina njegovog korijenskog čvora. U nastavku slijedi definicija generičke klase **CvorStabla**.

Listing 1.6: Definicija generičke klase **CvorStabla**

```
template<class T>
class CvorStabla{
 protected:
  CvorStabla *lijevoPodstablo;
  CvorStabla *desnoPodstablo;
  T podatak;
 public:
  CvorStabla(T);
  virtual ~CvorStabla();
  friend class Stablo<T>;
  friend class UpletenoStablo<T>;
};
```

Prema sadržaju Listinga 1.6, generičke klase **Stablo** i **UpletenoStablo** definisane su kao prijatelji klase **CvorStabla**. Ova relacija nudi njihovim metodama direktan pristup atributima klase **CvorStabla** bez potrebe za korištenjem "seter" i "geter" metoda. Pretpostavlja se da klasa **CvorStabla** predstavlja čvor u binarnom stablu. Shodno tome, njen konstruktor treba biti implementiran tako da inicijalno sadrži samo jedan interni čvor. Taj čvor upućuje na lijevi i desni eksterni čvor, koji su pri inicijalizaciji postavljeni na vrijednost **NULL**. Nadalje, prilikom kreiranja instance stabla, odnosno internog korijenskog čvora koji ga predstavlja, potrebno je tom čvoru dodijeliti odgovarajuću vrijednost. Zbog toga, pored inicijalizacije privatnih članica `lijevoPodstablo` i `desnoPodstablo`, atribut `podatak` mora se inicijalizovati vrijednošću koja se prosljeđuje kao formalni parametar konstruktora. Implementacija konstruktora može biti oblikovana na sljedeći način:

```
template<class T>
CvorStabla<T>::CvorStabla(T p): podatak(p),lijevoPodstablo(NULL),desnoPodstablo(
    NULL){}
```

Zadatak virtualnog destruktora klase **CvorStabla** je da rekurzivno uklanja potomke lijevog i desnog podstabla, ali samo ako oni postoje. Budući da će se klasa **CvorStabla** koristiti u daljoj implementaciji, upotreba ključne riječi **virtual** osigurava automatsku dealokaciju memorije zauzete prilikom alociranja objekata tipa **CvorStabla**. Implementacija destruktora prikazana je ispod:

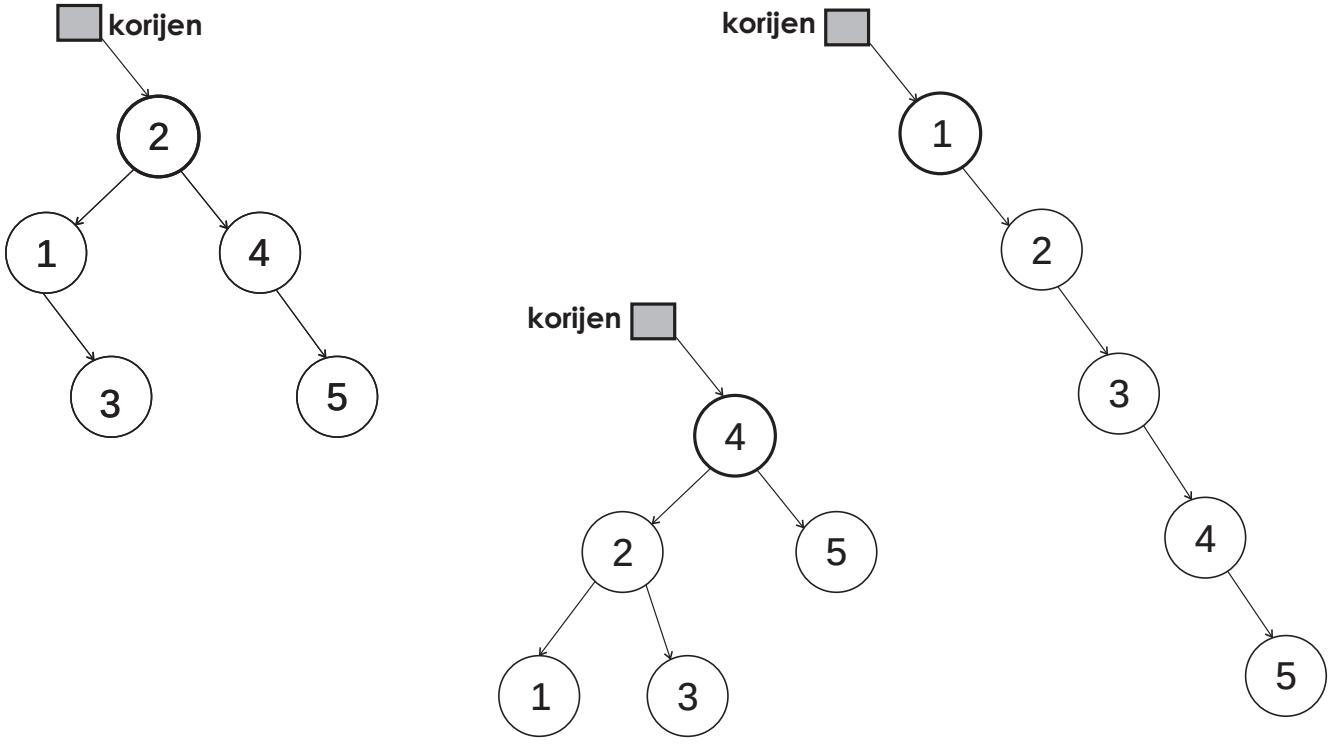
```
//Implementacija destruktora
template<class T>CvorStabla<T>::~CvorStabla(){
 if(lijevoPodstablo) delete lijevoPodstablo;
 if(desnoPodstablo) delete desnoPodstablo;
}
```

1.4.1 Binarna stabla pretrage

Jedan od ključnih razloga za uvođenje binarnog stabla pretrage je pružanje efikasnog traženja podataka s vremenskom složenošću koja je proporcionalna $\mathcal{O}(\log n)$. Binarno stablo se naziva **binarno stablo pretrage** (eng. *Binary Search Tree*, **BST**) ako su njegovi podaci organizovani prema sljedećim pravilima:

- svi podaci storirani u čvorovima lijevog podstabla čvora A su manji od podatka u samom čvoru A;
- svi podaci storirani u čvorovima desnog podstabla čvora A su veći od podatka koji je storiran u čvoru A.

Generalno, za isti skup podataka se mogu predložiti razni tipovi binarnog stabla pretrage. Na primjer, elementi 1, 2, 3, 4 i 5 se mogu na više načina čuvati unutar stabla u zavisnosti od toga koji od njih se prvi unosi u stablo, kao i od toga kojim se redoslijedom oni dodaju u stablo. Na slici 1.5 su predstavljene tri verzije binarnog stabla za čuvanje istog skupa podataka.



Slika 1.5: Tri načina kako se elementi predstavljaju unutar stabla.

Također, kako postoji linearno uređenje među stavkama odnosno podacima storiranim u čvorovima binarnog stabla, ti se podaci mogu porediti. Na primjer, ukoliko se u čvorovima nalaze brojevi, tada se mogu upotrijebiti operatori za poređenje, npr. operator manje (<), jednako (=) ili veće (>). Također, ukoliko se u čvorovima nalaze nizovi znakova odnosno stringovi, oni se tada mogu leksiografski upoređivati. Štaviše, ako se u čvorovima nalaze neki složeniji geometrijski objekti (npr. segmenti), oni se upoređuje na osnovu unaprijed date kriterijske funkcije. Generalno, pretraživanje **stavki** (eng. *Items*) u binarnom stablu zasniva se na **kriterijskoj funkciji za poređenje** (eng. *Comparison Function*). Ona vraća vrijednosti iz skupa $\{-1, 0, 1\}$ prema sljedećim pravilima:

a) ako je prva stavka (podatak ili vrijednost neke promjenljive) manja od druge stavke, tada kriterijska funkcija vraća vrijednost -1;
b) ako je prva stavka jednaka drugoj stavci, tada ona vraća vrijednost 0;
c) ako je prva stavka veća od druge stavke, tada ona vraća vrijednost 1.

Slijedi definicija generičke klase **Stablo**, čija je osnovna svrha omogućiti reprezentaciju različitih tipova binarnih stabala pretrage.

Listing 1.7: Definicija generičke klase **Stablo**

```
template<class T>class Stablo{
```

```
  CvorStabla<T>*korijen;
  int(*funkcijaPoretka)(T,T);//kriterijska funkcija
  CvorStabla<T>*vratiNajmanji(CvorStabla<T> *);
  void ukloni(T,CvorStabla<T>* &k);
  void inorderObilazak(CvorStabla<T>*,void(*fP)(T));
 public:
  Stablo(int(*fP)(T,T));
  ~Stablo(){if(korijen) delete korijen;};
  int jelPrazno()const{return(korijen==NULL);};
  T pronadji(T);
  T dajNajmanji();
  void ukloni(T e){ukloni(e, korijen);};
  T ukloniNajmanji();
  void ubaci(T);
  void inorderObilazak(void(*f)(T)){inorderObilazak(korijen, f);}
};
```

U nastavku se daju implementacije privatne i javne funkcije članice generičke klase **Stablo**. Kao prvo, daje se implementacija konstruktor klase:

```
//Implementacija konstruktora klase
template<class T>
Stablo<T>::Stablo(int(*f)(T,T)):funkcijaPoretka(f),korijen(NULL){}
```

Zatim se implementira metoda *pronadji*(·), koja provjerava prisustvo određenog podatka u stablu. Podatak koji treba pronaći prosljeđuje se metodi putem formalnog parametra generičkog tipa T. Proces pretrage odvija se kao **cik-cak traženje** (eng. *Zig-Zag Searching*), pri čemu se pretraga započinje od korijena stabla i nastavlja kroz lijevo ili desno podstablo, ovisno o kriterijskoj funkciji, sve dok se željeni podatak ne pronađe. Ako traženi podatak nije prisutan u stablu, metoda vraća vrijednost **NULL**, što označava da se podatak ne nalazi u stablu. U suprotnom, metoda vraća proslijeđeni podatak, potvrđujući njegovu prisutnost. Implementacija ove metode data je ispod:

```
//Implementacija metode pronadji()
template<class T> T Stablo<T>::pronadji(T e){
 CvorStabla<T>*cvor=korijen;
 while(cvor){
  int rezultat=(*funkcijaPoretka)(e, cvor->podatak);
  if(rezultat<0)cvor=cvor->lijevoPodstablo;
  else if(rezultat>0) cvor=cvor->desnoPodstablo;
  else return cvor->podatak; // pronadjen podatak
 }
 return NULL;//podatak nije pronadjen
}
```

Traženje najmanjeg elementa u stablu počevši od nekog unaprijed zadatog čvora realizuje se preko privatne funkcije članice *vratiNajmanji*(·). Zbog osobine stabla, pretraga se vrši samo kroz potomke lijevog podstabla. Ako lijevi potomak početnog čvora ima vrijednost **NULL**, to znači da je taj čvor najmanji. U suprotnom, pretraga se nastavlja niz lijevo podstablo, sve dok se ne dođe do čvora čiji lijevi potomak ima vrijednost **NULL**, čime se potvrđuje da je taj čvor najmanji. Slično tome, može se implementirati metoda *vratiNajveci*, koja se razlikuje od prethodne samo po tome što pretraga ide duž desnog umjesto lijevog podstabla. Na osnovu ovog objašnjenja, predlaže se implementacija privatne metode *vratiNajmanji*(·):

```
//Implementacija privatne metode vratiNajmnji
template<class T>
CvorStabla<T>*Stablo<T>::vratiNajmanji(CvorStabla<T>*e){
 if(e==NULL)return NULL;
 while(e->lijevoPodstablo)e=e->lijevoPodstablo;
 return e;
}
```

Ovako implementirana metoda za parcijalno traženje najmanjeg elementa može se upotrijebiti za nalaženje najmanjeg elementa u cijelom stablu. Dovoljno je da pretraga počne od korijenskog čvora. Dakle, implementacija javne metode *dajNajmanji*(·) ima ovaj oblik:

```
//Implementacija javne metode dajNajmnji
template<class T> T Stablo<T>::dajNajmanji(){
 CvorStabla<T>*nE{vratiNajmanji(korijen)};//Od C++11
 return (nE?nE->podatak:NULL);
}
```

Na početku poglavlja opisani su tri načina za obilaženje čvorova stabla, a u nastavku će biti implementiran inorderObilazak, dok se preostali obilasci mogu realizovati na sličan način. Kao što je rečeno, kod inorder obilaska, prvo se posjećuje lijevo podstablo (L), zatim čvor (Č), a na kraju desno podstablo (D). U sljedećem isječku koda, prvo će biti implementirana privatna metoda *inorderObilazak*(·, ·), koja omogućuje rekurzivno posjećivanje jednog dijela stabla (lokalna pretraga).

```
template<class T>
void Stablo<T>::inorderObilazak(CvorStabla<T>*e, void(*f)(T)){
 if(e){
  inorderObilazak(e->lijevoPodstablo, f);//prvo lijevo podstablo
  (*f)(e->podatak);//pa potom cvor
  inorderObilazak(e->desnoPodstablo, f);//zatim desno podstablo
 }
}
```

Na temelju gornje implementacije, realizacija javne metode *inorderObilazak*(·) sastoji se od pozivanja njene privatne metode, pri čemu se kao formalni parametri prosljeđuju korijenski čvor i kriterijska funkcija. Jedan primjer upotrebe javne metode *inOrderObilazak*(·) za ispis elemenata stabla u sortiranom poretku prikazan je u sljedećem isječku koda:

```
s->inorderObilazak(izlistajSadrzaj);
```

pri čemu je promjenljiva *s* instanca generičke klase **Stablo**, koja u svojim čvorovima pohranjuje stringove leksiografski poredane, dok prototip funkcije *izlistajSadrzaj*(·) ovako izgleda:

```
void izlistajSadrzaj(char *A){std::cout<<A;}
```

Slijedi opis ubacivanja elemenata u binarno stablo pretrage. Naime, za izvršenje dodavanja, koristi se "cik-cak" tehnika kako bi se pronašla odgovarajuća pozicija za eksterni čvor u koji se pohranjuje nova vrijednost. Drugim riječima, kada se pronađe prava pozicija, potrebno je kreirati novu instancu koja će biti povezana putem pokazivača sa svojim roditeljskim čvorom. Implementacija metode *ubaci*(·) ovako izgleda:

```
template<class T> void Stablo<T>::ubaci(T pE){
 if(korijen== NULL){
```

```
   korijen= new CvorStabla<T>(pE); return;
  }
  else{
   int rez;
   CvorStabla<T> *e,*k(korijen);
   while(k){
    e=k;rez=(*funkcijaPoretka)(pE,e->podatak);
    if(rez<0)k=e->lijevoPodstablo;
    else if(rez>0)k=e->desnoPodstablo;
    else return;//element pE je u stablu
   }//kraj while
   if(rez<0)e->lijevoPodstablo=new CvorStabla<T>(pE);
   else e->desnoPodstablo=new CvorStabla<T>(pE);
  }
 }
```

Na slici 1.6 b) prikazana je grafička ilustracija ubacivanja broja 1.5 u postojeće stablo prikazano na slici 1.6 a). Cik-cak staza, koja počinje od korijenskog čvora 3, podebljano je označena na slici 1.6 b). U nastavku slijedi opis brisanja proizvoljnog elementa iz binarnog stabla. Treba napomenuti da je brisanje elemenata nešto složenije od unosa, jer ono zahtijeva uništavanje čvora koji sadrži traženi element. Ako se briše čvor (roditelj) koji je povezan samo s jednim nepraznim čvorom (dijete) putem pokazivača, brisanje je jednostavno. U tom slučaju, nakon brisanja tog čvora, potrebno je samo preusmjeriti pokazivač roditeljskog čvora prema nepraznom djetetu. Dakle, brisanje se svodi na preusmjeravanje pokazivača. Međutim, situacija postaje znatno složenija kada čvor koji se briše ima nepraznu djecu. Prije implementacije metode *ukloni*$(\cdot)$, neka je stanje binarnog stabla kao na slici 1.7 a), u čijim čvorovima su pohranjeni elementi skupa $\{5,4,6,2,1,3,0,7,9,8\}$. Kao što se vidi sa slike 1.7 a), ukoliko se obrišu čvorovi koji sadrže elemente 6, 7, 9, 8, 4, 1, 3, 0, onda u tom slučaju se obavlja samo preusmjeravanje pokazivača, pa je brisanje relativno lako. Međutim, ukoliko se briše čvor 2, budući da on upućuje na dva neprazna čvora 1 i 3, jedan od načina da se izvede njegovo brisanje jeste da se izvrši preusmjeravanje pokazivača sa lijevog podstabla čvora 4 na čvor 3, pa da se zatim pronađe pozicija gdje bi se mogao postaviti pokazivač koji bi upućivao na čvor 1. Jasno je da se treba preusmjeriti pokazivač sa lijevog podstabla čvora 3 na čvor 1. Na ovaj način se generiše binarno stablo u kojem više ne egzistira čvor 2 (Slika 1.7 b)). Rezime postupka brisanja određenog elementa a iz binarnog stabla može se sažeti na sljedeći način. Prvo se, korištenjem rekurzivne verzije cik-cak tehnike, pronalazi čvor koji sadrži element a. Nakon što se identifikuje odgovarajući čvor, razmatraju se tri moguća scenarija koja mogu nastupiti:

i) Čvor koji sadrži element a ima prazno lijevo podstablo, kao na primjer čvor koji sadrži broj 6 (Slika 1.7 a)). U tom slučaju vrši se preusmjeravanje pokazivača, tako da roditelj čvora (čvor 5 na slici) koji se briše sada upućuje na čvor na koje je upućivalo desno podstablo (dijete) čvora koji se briše. Npr. na slici 1.7 a), desno podstablo čvora 5 treba upućivati na čvor 7 nakon brisanja čvora 6;

ii) Čvor koji sadrži element a ima neprazno lijevo, a prazno desno podstablo. Na primjer, na slici 1.7 a), to je čvor koji sadrži broj 9. Analogno slučaju i), vrši se preusmjeravanje pokazivača, tako da roditeljski čvor sada upućuje na lijevo podstablo čvora koji se briše;

iii) Čvor koji sadrži element a ima neprazno lijevo i desno podstablo. Na slici 1.7 a), to je čvor koji sadrži broj 2. Tada se pronađe najmanji element k_1 u desnom podstablu čvora koji se briše. Konkretno, na slici to je čvor koji sadrži broj 3. Potom se kopiraju podaci iz čvora k_1 (na slici je to čvor 3) u čvor a (na slici je to čvor 2), te se nadalje rekurzivno vrši brisanje čvora k_1.

Na osnovu gore ispričanog, implementacija privatne metode *ukloni*$(\cdot)$ generičke klase **Stablo** može se ovako napisati: