Čas 5.1, 5.2, 5.3 - implementacija struktura podataka

U poglavlju o korišćenja struktura podataka upoznali smo različite strukture podataka. U ovom poglavlju ćemo ih razvrstati po tome kako su implementirane.

- U grupu sekvencijalnih struktura podataka (kontejnera) spadaju `array`, `vector`, `list`, `forward_list` i `deque`.
- U grupu adaptora kontejnera spadaju `stack`, `queue` i `priority_queue`.
- U grupu asocijativnih kontejnera spadaju `set`, `multiset`, `map` i `multimap`.
- U grupu neuređenih asocijativnih kontejnera spadaju `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

Svaki sekvencijalni kontejner ima svoju specifičnu implementaciju.

- `array` je implementiran kao običan statički niz i služi samo da omogući da se klasični statički nizovi koriste na isti način kao i drugi kontejneri.
- `vector` je implementiran preko dinamičkog niza koji se po potrebi realocira.
- `list` je implementiran preko dvostruko povezane liste, dok je `forward_list` implementiran preko jednostruko povezane liste.
- `deque` je specifična struktura podataka implementirana kao vektor u kome se nalaze pokazivači na nizove fiksne veličine.

Adaptori kontejnera samo predstavljaju sloj iznad nekog od postojećih sekvencijalnih kontejnera i pružaju apstraktni interfejs iznad implementacije sekvencijalnog kontejnera, implementirajući funkcije tog interfejsa korišćenjem sekvencijalnog kontejnera za skladištenje podataka. Adaptori imaju svoj podrazumevani sekvencijalni kontejner, koji se može promeniti prilikom deklarisanja promenljivih.

- `stack` implementira funkcije steka korišćenjem vektora za skladištenje podataka. Tip `stack<int>` podrazumeva zapravo `stack<int, vector<int>>`, a moguće je koristiti i, na primer, `stack<int, forward_list<int>>` u kom se za implementaciju steka koristi jednostruko povezana lista.
- `queue` implementira funkcije reda korišćenjem deka za skladištenje podataka. Tip `queue<int>` podrazumeva zapravo `queue<int, deque<int>>`.
- `priority_queue` implementira funkcije reda sa prioritetom korišćenjem vektora za skladištenje podataka. Tip `priority_queue<int>` zapravo predstavlja `priority_queue<int, vector<int>, less<int>>` gde je `less<int>` funkcija koja se koristi za poređenje elemenata i prozrokuje uređenost po opadajućem redosledu prioriteta (na vrhu je element sa najvećim prioritetom). U vektoru su smešteni elementi specijalnog drveta koji se naziva *hip* i koje će biti objašnjeno kasnije.

Asocijativni kontejneri su implementirani pomoću samobalansirajućih uređenih binarnih drveta (obično su to crveno-crna drveta, RBT).

- `set<T>` je implementiran pomoću uređenog binarnog drveta u kome se u čvorovima nalaze elementi skupa i u kom su u svim čvorovima različite vrednosti.
- `multiset<T>` može biti implementiran pomoću uređenog binarnog drveta u kome se u čvorovima nalaze elementi multiskupa i u kom je moguće da postoji više čvorova u kome su različite vrednosti ili pomoću uređenog binarnog drveta u čijim se čvorovima nalaze elementi multiskupa uz njihov broj pojavljivanja, bez ponavljanja elemenata skupa.
- `map<K, V>` je implementiran pomoću uređenog binarnog drveta na osnovu ključeva u kome se u čvorovima nalaze podaci o ključevima i njima pridruženim vrednostima i u kom su u svim čvorovima različite vrednosti ključeva.
- `multimap<K, V>` može biti implementiran pomoću pomoću uređenog binarnog drveta na osnovu ključeva u kome se u čvorovima nalaze podaci o ključevima i njima pridruženim vrednostima i u kom više čvorova može imati istu vrednost ključa.

Neuređeni asocijativni kontejneri su implementirani pomoću heš-tabela.

Dinamički nizovi (vector)

Bez imalo dileme možemo reći da je najkorišćenija struktura podataka niz. S obzirom na to da veoma često broj potrebnih elemenata niza znamo tek u fazi izvršavanja programa, umesto klasičnih, statički alociranih nizova često se koriste dinamički alocirani nizovi. Videli smo da se u jeziku C++ za to koristi struktura podataka `vector`. U ovom poglavlju ćemo dati neke naznake kako bi on mogao biti implementiran.

Osnovna ideja dinamičkog niza je to da se u startu alocira neki pretpostavljeni broj elemenata i da se, kada se ustanovi da taj broj elemenata nije više dovoljan, izvrši realokacija niza, tako što se alocira novi, veći niz, zatim se u taj novi niz iskopiraju elementi originalnog niza i na kraju se taj stari niz obriše, a pokazivač preusmeri ka novom nizu (u jeziku C++ ne postoji operator koji bi bio ekvivalentan C funkciji `realloc`).

Implementacija osnovne funkcionalnosti dinamičkog niza u jeziku C++ mogla bi da izgleda ovako (jednostavnosti radi koristimo globalne promenljive i funkcije - one se jednostavno izbegavaju kada se koristi objektno programiranje, ali to nije u fokusu ovog kursa). Ulogu pokazivača `NULL` iz jezika C ima pokazivač `nullptr`, ulogu funkcije `malloc` ima operator `new`, a ulogu funkcije `free` ima operator `delete` (postoje ozbiljne razlike između ovih C funkcija i C++ operatora, ali ni to nije u fokusu ovog kursa).

```
// adresa početka niza
```

```
int* a = nullptr;
// broj alociranih i broj popunjenih elemenata
int alocirano = 0, int n = 0;

// čitanje vrednosti elementa na datoj poziciji
// ne proverava se pripadnost opsegu niza
int procitaj(int i) {
  return a[i];
}

// upis vrednosti elementa na datu poziciju
// ne proverava se pripadnost opsegu niza
void postavi(int i, int x) {
  a[i] = x;
}

// pomoćna funkcija koja vrši realokaciju niza na dati
// broj elemenata (veći ili jednak trenutnom broju elemenata)
void realociraj(int m) {
  // alociramo novi niz
  int* novo_a = (int*)new int[m];
  alocirano = m;
  // ako postoji stari niz, kopiramo njegove elemente u novi
  // i brišemo stari niz
  if (a != nullptr) {
    copy_n(a, n, novo_a);
    delete[] a;
  }
  // pokazivač usmeravamo ka novom nizu
  a = novo_a;
}

// dodavanje datog elementa na kraj dinamičkog niza
// niz se automatski realocira ako je potrebno
void dodajNaKraj(int x) {
  if (alocirano <= n)
    realociraj(2 * alocirano + 1);
  a[n++] = x;
}

// brisanje celog niza
void obrisi() {
  delete[] a;
}
```

Ako se dinamička realokacija vrši na osnovu neke geometrijske strategije za

povećanje veličine (npr. svaki put povećamo broj elemenata za 30%), dodavanje na kraj će biti moguće u amortizovanom konstantnom vremenu. Indeksni pristup elementu (pristup na osnovu pozicije) zahteva konstantno vreme (isto kao kod običnog niza).

Liste (list, forward_list)

Strukture liste podrazumevaju da se podaci čuvaju u memoriji u čvorovima kojima se pored podataka čuvaju pokazivači. U zavisnosti od toga da li se čuvaju samo pokazivači na sledeći ili i na prethodni element razlikuju se:

- jednostruko povezane liste
- dvostruko povezane liste

Pod pretpostavkom da su poznati pokazivači na početak i na kraj liste, jednostruko povezane liste dopuštaju dodavanje na početak i na kraj, kao i brisanje sa početka u vremenu $O(1)$, dok brisanje elementa sa kraja zahteva vreme $O(n)$. Umetanje i brisanje elementa iza ili ispred čvora na koji ukazuje poznati pokazivač se može izvršiti u vremenu $O(1)$, međutim pronalaženje pozicije na koju treba ubaciti element obično zahteva prolaz kroz listu i zahteva vreme $O(n)$. Pristup elementu na datoj poziciji zahteva vreme $O(n)$.

Pod pretpostavkom da su poznati pokazivači na početak i na kraj liste, dvostruko povezane liste dopuštaju i dodavanje i brisanje i sa početka i sa krajaliste u vremenu $O(1)$. Ostale operacije se izvršavaju u istom vremenu kao i kod jednostruko povezanih lista. Mane dvostruko povezanih u odnosu na jednostruko povezane liste su to što zbog čuvanja pokazivača na prethodne elemente zahtevaju više memorije i pojedinačne operacije mogu biti malo sporije jer se zahteva ažuriranje više pokazivača. Prednosti su to što omogućuju efikasnije izvršavanje nekih operacija (pre svega brisanje sa kraja i iteracija unazad).

Upotreba lista je sve ređa i ređa na savremenim sistemima. Osnovni problem je to što su čvorovi često raštrkani po memoriji, pa su promašaji keš memorije mnogo češći nego u slučaju rada sa strukturama u kojima su podaci u memoriji smešteni povezano (nizovima i strukturama u kojima se koriste veći blokovi povezanih bajtova). Takođe, zbog čuvanja pokazivača liste zahtevaju mnogo više memorije nego nizovi. Prednosti lista u odnosu na nizove i dekove nastupaju pre svega u situacijama u kojima se u listama čuvaju veliki podaci. Tada se tokom realokacije dinamičkih nizova kopiraju velike količine podataka, što može biti neefikasno, pa je korisnije upotrebiti liste kod kojih se podaci ne realociraju.

S obzirom na to da se implementacija lista detaljno pokriva u kursu P2, ona u ovom materijalu neće biti prikazana.

Dekovi (deque)

Kao što smo videli u poglavlju o primenama struktura podataka, jedna veoma korisna struktura podataka je red sa dva kraja koja kombinuje funkcionalnost steka i reda.

Ako se za implementaciju koriste jednostruko povezane liste, tada se u vremenu $O(1)$ može vršiti ubacivanje na početak i brisanje sa početka, kao i ubacivanje na kraj (pod pretpostavkom da održavamo pokazivač na kraj). Brisanje sa kraja je operacija složenosti $O(n)$, čak i kada se čuva pokazivač na poslednji element (jer ne možemo da pronađemo pokazivač na pretposlednji element).

Ako se za implementaciju koriste dvostruko povezane liste, tada se ubacivanje i brisanje i sa početka i sa kraja može izvršiti u vremenu $O(1)$. Isto je i sa umetanjem elementa u sredinu (kada se zna njegova pozicija). Međutim, indeksni pristup (pristup članu na datoj poziciji i) u najgorem slučaju zahteva vreme $O(n)$. Jedini mogući način pretrage je linearna pretraga, čak i kada su elementi u redu sortirani.

U nastavku ćemo prikazati strukturu podataka *dek* (engl. deque) koja se često koristi za implementaciju redova sa prioritetom. Slično kao dvostruko povezane liste, dek pruža dodavanje i na početak i na kraj u vremenu $O(1)$ (doduše amortizovanom). Važna prednost u odnosu na dvostruko povezane liste je to što je indeksni pristup moguć u vremenu $O(1)$. Ovim je omogućena i binarna pretraga, što može nekada biti veoma važno. Dodavanje i brisanje elemenata sa sredine nije moguće izvršiti efikasno (te operacije zahtevaju vreme $O(n)$).

Dek možemo zamisliti kao niz segmenata iste, fiksne veličine. Svaki segment je struktura koja sadrži niz elemenata (bilo statički, bilo dinamički alociran) koji predstavlja neki deo reda. Na primer, red čiji su elementi 5, 7, 3, 1, 2, 5, 9, 7, 1, 4, 6, 2, može biti organizovan u 6 segmenata od po 4 elementa (u praksi je veličina pojedinačnih segmenata obično veća).

```
            +                                 +
? ? ? ?   ? ? 5 7   3 1 2 5   9 7 1 4   6 2 ? ?   ? ? ? ?
           levi                          desni
```

Dek čuva niz pokazivača na pojedinačne segmente. Dva segmenta su karakteristična: levi segment u kom se nalazi početak reda i desni segment u kom se nalazi kraj reda. Oba mogu biti samo delimično popunjeni. U svakom segmentu se čuva prva slobodna pozicija na koju se može dodati naredni element. Posebno se održavaju pozicije ta dva segmenta. U praznom deku levi i desni segment su susedni i prazni (tekući element levog segmenta je njegov desni kraj, a desnog segmenta je njegov levi kraj).

Dodavanje elementa na početak je veoma jednostavno ako levi segment nije popunjen do kraja. Element se samo dodaje na prvu slobodnu poziciju i ona se pomera nalevo. Kada je levi segment potpuno popunjen, prelazi se na popunjavanje prethodnog segmenta, ako on postoji. Ako ne postoji, onda se

vrši realokacija deka i proširuje se njegov broj segmenata (obično je novi broj segmenata nekoliko puta veći nego polazni, kako bi se naredne realokacije dešavale sve ređe i ređe i kako bi se obezbedilo amortizovano konstantno vreme dodavanja). Prilikom realokacije vrši se samo proširivanje niza pokazivača na segmente, a ne samih segmenata, što je veoma značajno ako se u deku čuvaju veći objekti (oni se prilikom alokaciji ne kopiraju, a ne kopiraju se ni sami segmenti). Prilikom realokacije, pokazivači na postojeće segmente se u proširenom nizu smeštaju na sredinu, a levo i desno od njih se smeštaju pokazivači na novo alocirane segmente koji su inicijalno prazni. Realokacijom se dobijaju segmenti levo od tekućeg potpuno popunjenog segmenta i dodavanje na početak se vrši u njih. Dodavanje na desni kraj teče potpuno analogno.

Brisanje sa levog kraja se vrši slično (uklanjanjem elemenata iz levog segmenta i prelaskom na naredni segment ako se nakon brisanja levi segment potpuno ispraznio). Brisanje sa desnog kraja je analogno.

Indeksni pristup elementu je moguće izvršiti u vremenu $O(1)$, tako što se prvo odredi kom segmentu pripada traženi element, a onda se pročita odgovarajući element iz tog segmenta. Jednostavan trik je da se traženi indeks uveća za prazan broj elemenata na levom kraju levog segmenta. Tada se pozicija segmenta u kom se element nalazi može jednostavno izračunati kao zbir pozicije levog segmenta i celobrojnog količnika indeksa i i veličine jednog segmenta, dok se pozicija unutar tog segmenta određuje kao ostatak u tom deljenju.

Prikažimo i jednu moguću implementaciju deka.

```
#include <iostream>

using namespace std;

// jedan segment
struct segment {
  // niz podataka
  int* podaci;
  // broj popunjenih elemenata niza
  int popunjeno;
  // pozicija u nizu na koje se moze ubaciti naredni element
  int tekuci;
};

// alokacija segmenta
segment* alocirajSegment(int velicina, int smer) {
  // alociramo novi segment
  segment* novi = new segment();
  // alociramo njegove podatke
  novi->podaci = new int[velicina];
  // nijedan podatak još nije upisan
  novi->popunjeno = 0;
```

```
  // tekucu poziciju odredjujemo u zavisnosti od smera popunjavanja
  novi->tekuci = smer == 1 ? 0 : velicina - 1;
  return novi;
}

// brisanje segmenta
void obrisiSegment(segment* s) {
  // brisemo podatke u segmentu
  delete[] s->podaci;
  // brisemo segment
  delete s;
}

struct dek {
  // niz pokazivača na segmente
  segment** segmenti;
  // broj segmenata u nizu
  int brojSegmenata;
  // velicina svakog segmenta
  int velicinaSegmenata;
  // pozicije na kojima se nalaze pokazivaci na krajnji levi
  // i kranjnji desni segment (oba mogu biti polupopunjena)
  int levo, desno;
};


// vrši se realociranje deka tako da ima dati broj segmenata
void realocirajDek(dek& d, int brojSegmenata) {
  // pravimo novi niz pokazivača na segmente
  segment** noviSegmenti = new segment*[brojSegmenata];
  // postojeće pokazivače na segmente ćemo smestiti negde oko
  // sredine novog niza
  int uvecanje = (brojSegmenata - d.brojSegmenata) / 2;
  // ako je postojao stari niz pokazivača na segmente
  if (d.segmenti != nullptr)
    // kopiramo te pokazivače u novi niz pokazivača, krenuvši od
    // pozicije uvecanje
    for (int i = 0; i < d.brojSegmenata; i++)
      noviSegmenti[uvecanje + i] = d.segmenti[i];
  // alociramo nove segmente na početku i na kraju novog niza
  for (int i = 0; i < uvecanje; i++)
    noviSegmenti[i] = alocirajSegment(d.velicinaSegmenata, -1);
  for (int i = uvecanje + d.brojSegmenata; i < brojSegmenata; i++)
    noviSegmenti[i] = alocirajSegment(d.velicinaSegmenata, 1);

  // brišemo stari niz pokazivača na segmente
```

```
  delete[] d.segmenti;
  // preusmeravamo pokazivač na niz pokazivača
  d.segmenti = noviSegmenti;
  // ažuriramo broj segmenata
  d.brojSegmenata = brojSegmenata;
  // ažuriramo granice popunjenog dela niza
  d.levo += uvecanje;
  d.desno += uvecanje;
}


// vraća referencu na i-ti element u deku
int& iti(const dek& d, int i) {
  // trenutni broj elemenata u krajnjem levom segmentu
  int uLevom = d.segmenti[d.levo]->popunjeno;
  // pomeramo indeks tako da je pozicija 0 na početku krajnjeg levog
  // segmenta
  i += d.velicinaSegmenata - uLevom;
  // određujemo segment u kom se nalazi traženi element
  segment* s = d.segmenti[d.levo + i / d.velicinaSegmenata];
  // čitamo element iz tog segmenta
  return s->podaci[i % d.velicinaSegmenata];
}

// dodajemo element na početak deka
int dodajNaPocetak(dek& d, int x) {
  // ako je levi segment potpuno popunjen
  if (d.segmenti[d.levo]->popunjeno == d.velicinaSegmenata) {
    // ako levo od njega nema više alociranih segmenata
    if (d.levo == 0)
      // realociramo dek i time alociramo nove segmente
      realocirajDek(d, d.brojSegmenata * 2);
    // prelazimo u prethodni segment
    d.levo--;
  }
  // segment u koji se može upisati element
  segment *s = d.segmenti[d.levo];
  // upisujemo element na tekuću slobodnu poziciju i pomeramo se
  // nalevo
  s->podaci[s->tekuci--] = x;
  // uvećavamo broj popunjenih elemenata
  s->popunjeno++;
}

// dodajemo element na kraj deka
int dodajNaKraj(dek& d, int x) {
```

```
  // ako je desni segment potpuno popunjen
  if (d.segmenti[d.desno]->popunjeno == d.velicinaSegmenata) {
    // ako desno od njega nema više alociranih segmenata
    if (d.desno == d.brojSegmenata - 1)
      // realociramo dek i time alociramo nove segmente
      realocirajDek(d, d.brojSegmenata * 2);
    // prelazimo na naredni segment
    d.desno++;
  }
  // segment u koji se može upisati element
  segment *s = d.segmenti[d.desno];
  // upisujemo element na tekuću slobodnu poziciju i pomeramo se
  // nalevo
  s->podaci[s->tekuci++] = x;
  // uvećavamo broj popunjenih elemenata
  s->popunjeno++;
}

// brisanje deka
void obrisiDek(dek& d) {
  // brisemo sve segmente
  for (int i = 0; i < d.brojSegmenata; i++)
    obrisiSegment(d.segmenti[i]);
  // brisemo niz pokazivaca na segmente
  delete[] d.segmenti;
}

int main() {
  // gradimo prazan dek
  dek d;
  d.velicinaSegmenata = 3;
  d.segmenti = nullptr;
  d.brojSegmenata = 0;
  d.levo = -1;
  d.desno = 0;

  // realociramo ga za 10 elemenata
  realocirajDek(d, 10);

  // dodajemo elemente na pocetak i na kraj
  for (int i = 0; i < 50; i++)
    dodajNaPocetak(d, i);
  for (int i = 50; i < 100; i++)
    dodajNaKraj(d, i);

  // ispisujemo elemente
```

```
  for (int i = 0; i < 100; i++)
    cout << iti(d, i) << endl;

  // brisemo dek
  obrisiDek(d);

  return 0;
}
```

Binarna drveta

Drveta su strukture podataka koje se koriste za predstavljanje hijerarhijskih odnosa između delova (na primer, izraza, sistema direktorijuma i datoteka, organizacije elemenata unutar HTML stranice, sintakse programa unutar prevodioca i slično). U nastavku ćemo se baviti upotrebom drveta za implementaciju struktura podataka čiju smo upotrebu razmotrili u posebnom poglavlju. Razmotrićemo dve posebne organizacije binarnih drveta: *uređena binarna drveta* (tj. *binarna drveta pretrage*) koja se koriste za implementaciju skupova, mapa (rečnika), multiskupova i multimapa, kao i *hipove* koji se koriste za implementaciju redova sa prioritetom.

Binarno drvo je rekurzivno definisani tip podataka: ili je prazno ili sadrži neki podatak i levo i desno poddrvo. U programskom jeziku Haskell, to se izražava veoma elegantno: drvo se konstruiše ili pomoću konstruktora (funkcije) `Null` bez parametra ili pomoću konstruktora (funkcije) `Node` čiji su parametri podatak tipa `Int` i dva drveta.

```
{- Tip podataka za predstavljanje uređenog binarnog drveta -}
data Tree =    Null
             | Node Int Tree Tree
```

Tako je `Node 3 (Node 5 Null Null) Null` jedno ispravno formirano drvo tj. podatak tipa `Tree`. Haskell ćemo u ovom materijalu koristi umesto pseudokoda za opis nekih operacija sa drvetima (od čitaoca se ne očekuje poznavanje ovog jezika).

Naravno, podatak ne mora biti tipa `Int` i moguće je definisati i tip podataka kome se prosleđuje tip elemenata unutar drveta (kao što se u C++-u za svaki skup prosleđuje tip podataka tog skupa, kao na primer `set<int>`).

```
{- Tip podataka za predstavljanje uređenog binarnog drveta
   sa podacima tipa a-}
data Tree a =   Null
              | Node a (Tree a) (Tree a)
```

Drvo koje sadrži cele brojeve je onda tipa `Tree Int`.

Uobičajeni način za predstavljanje drveta u jeziku C++ je preko čvorova uvezanih pomoću pokazivača.

```
struct cvor {
  int x;
  cvor *levo, *desno;
};
```

U slučaju da se drveta koriste za predstavljanje skupova ili multiskupova u čvorovima se čuva samo jedan podatak (element skupa). Ako se koriste za predstavljanje mapa ili multimapa, u čvorovima se čuvaju dva podatka: ključ i vrednost pridružena ključu.

```
struct cvor {
  int k, v;
  cvor *levo, *desno;
}
```

I u jeziku C++ moguće je parametrizovati tip čvora tipom podataka, ali se time nećemo detaljno baviti.

```
template<class K, class V>
struct cvor {
  K k; V v;
  cvor *levo, *desno;
}
```

Pošto je drvo rekurzivno-definisana struktura podataka, najlakše je funkcije koje operišu sa drvetima realizovati rekurzivno. U nekim situacijama je moguće relativno lako eliminisati rekurziju, dok je u nekim drugim situacijama implementiranje nerekurzivnih operacija komplikovano (i zahteva korišćenje steka). Većina funkcija koje ćemo mi implementirati će biti rekurzivna.

Binarna drveta pretrage (uređena binarna drveta)

Drvo je binarno drvo pretrage ako je prazno ili ako je njegovo levo i desno poddrvo uređeno i ako je čvor u korenu veći od svih čvorova u levom poddrvetu i manji od svih čvorova u desnom poddrvetu. U multiskupovima i multimapama je dozvoljeno postojanje duplikata u drvetu, ali u običnim skupovima i mapama nije.

Naglasimo da nije dovoljno proveriti da je vrednost u svakom čvoru veća od vrednosti u korenu levog poddrveta i vrednosti u korenu desnog poddrveta!

```
bool jeUredjeno(cvor* drvo) {
  if (drvo == nullptr)
     return true;
  if (drvo->levo != null && drvo->x <= drvo->levo->x)
     return false;
```

```
  if (drvo->desno != null && drvo->x >= drvo->desno->x)
     return false;
  return jeUredjeno(drvo->levo) && jeUredjeno(drvo->desno);
}
```

Prethodna vraća da je naredno drvo uređeno, što nije tačno, jer je vrednost 4, na pogrešnoj strani čvora 3.

```
      3
  2     5
1   4
```

Ispravna funkcija zahteva izračunavanje najmanje vrednosti desnog poddrveta i najveće vrednosti levog poddrveta.

```
bool jeUredjeno(cvor* drvo) {
  if (drvo == nullptr)
     return true;
  if (drvo->levo != null && drvo->x <= maxVrednost(drvo->levo))
     return false;
  if (drvo->desno != null && drvo->x >= minVrednost(drvo->desno))
     return false;
  return jeUredjeno(drvo->levo) && jeUredjeno(drvo->desno);
}
```

Ako se računanje najmanje i najveće vrednosti uvek izvršava iz početka, posebnim rekurzivnim funkcijama, provera će biti neefikasna. Mnogo bolje rešenje je da se ojača induktivna hipoteza i da funkcija pored provere uređenosti računa minimalnu i maksimalnu vrednost u drvetu.

```
bool jeUredjeno(cvor* drvo, int& minV, int& maxV) {
  if (drvo == nullptr) {
     min = numeric_limits<int>::max();
     max = numeric_limits<int>::min();
     return true;
  }

  int minL, maxL;
  bool uredjenoL = jeUredjeno(drvo->levo, minL, maxL);
  if (!uredjenoL) return false;

  int minD, maxD;
  bool uredjenoD = jeUredjeno(drvo->desno, minD, maxD);
  if (!uredjenoD) return false;

  if (drvo->x <= maxL) return false;
  if (drvo->x >= minD) return false;

  minV = minL;
```

```
  maxV = maxD;

  return true;
}
```

Umetanje u binarno drvo pretrage

Umetanje u binarno drvo pretrage se uvek vrši na mesto lista. Vrednost koja se umeće se poredi sa vrednošću u korenu, ako je manja od nje umeće se levo, a ako je veća od nje umeće se desno. Ako je jednaka vrednosti u korenu, razlikuju se slučaj u kom se duplikati dopuštaju i slučaj u kom se ne dopuštaju.

Na primer, ako se duplikati ne dopuštaju, umetanje se može izvršiti narednom funkcijom.

```
-- ubacivanje elementa x u prazno drvo
ubaci x Null = Node x Null Null
-- ubacivanje elementa x u neprazno drvo
ubaci x (Node k l d)
  | x < k   = Node k (ubaci x l) d   -- ubacujemo x levo
  | x > k   = Node k l (ubaci x d)   -- ubacujemo x desno
  | x == k  = Node k l d             -- drvo se ne menja
```

U jeziku C++, implementacija teče po sličnom principu.

```
cvor* napraviCvor(int x) {
  cvor* novi = new cvor();
  novi->levo = novi->desno = nullptr;
  novi->x = x;
  return novi;
}

cvor* ubaci(cvor* drvo, int x) {
  if (drvo == nullptr)
    return napraviCvor(x);
  if (x < drvo->x)
    drvo->levo = ubaci(drvo->levo, x);
  else if (x > drvo->x)
    drvo->desno = ubaci(drvo->desno, x);
  return drvo;
}
```

Brisanje celog drveta

Nakon završetka rada sa drvetom potrebno ga je ukloniti iz memorije.

```
void obrisi(cvor* drvo) {
   if (drvo != nullptr) {
      obrisi(drvo->levo);
      obrisi(drvo->desno);
      delete drvo;
   }
}
```

Brisanje iz uređenog binarnog drveta

Brisanje iz uređenog binarnog drveta je komplikovanije.

Implementacija u jeziku Haskell je veoma jezgrovita i lako razumljiva.

```
{-
Iz datog drveta se ukljanja čvor sa najmanjom vrednošću.
Funkcija vraća uređen par koji čine obrisana vrednost i izmenjeno drvo.
-}
-- deklaracija funkcije
obrisiMin :: Tree a -> (a, Tree a)
-- ako nema levog poddrveta, briše se element k iz korena, a
-- rezultat je desno poddrvo
obrisiMin (Node k Null d) = (k, d)
-- u suprotnom se briše najmanji element levog poddrveta
-- i dobijeno izmenjeno levo poddrvo se postavlja levo od korena
obrisiMin (Node k l d) = let (x, l') = obrisiMin l
                          in (x, Node k l' d)

{- Iz datog drveta ukljanja dati čvor sa datom vrednošću -}
obrisi :: Ord a => a -> Tree a -> Tree a
-- iz praznog drveta nema šta da se obriše (traženi element ne postoji)
obrisi x Null = Null
-- drvo je neprazno
obrisi x (Node k l d)
 -- ako je element koji se briše manji od korena, briše se iz
 -- levog poddrveta
 | x < k  = Node k (obrisi x l) d
 -- ako je element koji se briše veći od korena, briše se iz
 -- desnog poddrveta
 | x > k  = Node k l (obrisi x d)
 -- ako je element koji se briše jednak korenu, briše se iz korena
 | x == k = case d of
               -- ako desno poddrvo ne postoji, rezultat je levo poddrvo
               Null -> l
               -- u suprotnom se briše najmanji element iz
               -- desnog poddrveta i on se postavlja u koren
```

```haskell
                _      -> let (x', d') = obrisiMin d
                              in Node x' l d'
```

Implementacija prethodne funkcionalnosti u jeziku C++ teče po potpuno istom principu, ali je malo komplikovanija.

```cpp
// Iz datog drveta se ukljanja čvor sa najmanjom vrednošću
cvor* obrisiMin(cvor* drvo, int& x) {
   // iz praznog drveta nema šta da se briše
   if (drvo == nullptr) return nullptr;
   // ako je levo poddrvo prazno
   if (drvo->levo == nullptr) {
      // brišemo koren i vraćamo desno poddrvo
      cvor* desno = drvo->desno;
      x = drvo->x;
      delete drvo;
      return desno;
    }
    // u suprotnom brišemo najmanji element levog poddrveta
    drvo->levo = obrisiMin(drvo->levo, x);
    return drvo;
}

// Iz datog drveta ukljanja dati čvor sa datom vrednošću
cvor* obrisi(cvor* drvo, int x) {
  // iz praznog drveta nema šta da se briše
  if (drvo == nullptr)
    return nullptr;
  // čvor se nalazi levo, pa se tamo i briše
  if (x < drvo->x)
    drvo->levo = obrisi(drvo->levo, x);
  // čvor se nalazi desno, pa se tamo i briše
  else if (x > drvo->x)
    drvo->desno = obrisi(drvo->desno, x);
  // čvor se nalazi u korenu
  else {
    if (drvo->desno == nullptr) {
      // ako je poddrvo prazno brišemo koren i vraćamo levo poddrvo
      cvor* levo = drvo->levo;
      delete drvo;
      return levo;
    } else {
      // desno poddrvo nije prazno, pa brišemo najmanji čvor iz
      // njega i vrednost iz tog čvora upisujemo u koren
      int min;
      drvo->desno = obrisiMin(drvo->desno, min);
      drvo->x = min;
```

```
    }
  }
  // fizički čvor u kome je koren se nije promenio
  return drvo;
}
```

Balansirana binarna drveta

Mana klasičnih uređenih binarnih drveta je to što ako nisu balansirana operacije pretrage, umetanja i brisanja mogu zahtevati linearno vreme u odnosu na broj čvorova u drvetu. Balansirana binarna drveta garantuju da se to ne može dogoditi i da je vremenska složenost najgoreg slučaja ovih operacija logaritamska u odnosu na broj čvorova u drvetu. U nastavku ćemo razmotriti dve najčešće korišćene vrste balansiranih binarnih drveta.

- Adison-Veljski Landisova drveta
- Crveno-crna drveta

AVL

RBT

RBT mora da zadovolji sledeće invarijante.

1. Svaki čvor je ili crven ili crn
2. Koren je crn.
3. Svi listovi su crni i ne sadrže vrednosti (označavamo ih sa NIL).
4. Svi crveni čvorovi imaju tačno dva crna deteta.
5. Sve putanje od nekog čvora do njegovih listova sadrže isti broj crnih čvorova.

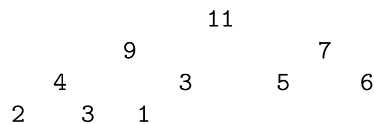
Navedena svojstva nam garantuju da će *svaka putanja od korena do njemu najdaljeg lista biti najviše duplo duža nego putanja do njegovog najbližeg lista*. Zaista, najkraća putanja do lista će se sastojati samo od crnih čvorova, dok će se se u najdužoj putanji naizmenično smenjivati crveni i crni listovi (jer na osnovu 4. posle crvenog čvora mora doći crni, a na osnovu 5. sve putanje imaju isti broj crnih čvorova). Ovo svojstvo nam garantuje određeni vid balansiranosti drveta i logaritamsku složenost operacija (pod uslovom da svako umetanje i brisanje u drvo održava nabrojanih 5 invarijanti).

Hip

Maks-hip (engl. Max-heap) je binarno drvo koje zadovoljava uslov da je svaki nivo osim eventualno poslednjeg potpuno popunjen, kao i da je vrednost svakog čvora veća ili jednaka od vrednosti u njegovoj deci. Min-hip se definiše analogno,

jedino što se zahteva da je vrednost svakog čvora manja ili jednaka od vrednosti u njegovoj deci. Maks-hip omogućava veoma efikasno određivanje maksimalnog elementa u sebi (on se uvek nalazi u korenu), a videćemo i veoma efikasno njegovo uklanjanje. Pošto je i operacija umetanja novog elementa u hip efikasna, ova struktura podataka je veoma dobar kandidat za implementaciju reda sa prioritetom.

Razmotrimo narednu implementaciju maks-hipa. Drvo smeštamo u niz. Koren na poziciju 0, naredna dva elementa na pozicije 1 i 2, zatim naredne elemente na pozicije 3, 4, 5 i 6 itd. Na primer, hip



predstavljamo nizom

```
11 9 7 4 3 5 6 2 3 1
```

Zahtevamo da se hip popunjava redom i su svi nivoi osim eventualno poslednjeg kompletno popunjeni, a da su u poslednjem nivou popunjeni samo početni elementi.

Lako je uočiti da ako se koren nalazi na poziciji i, onda se njegovo levo dete nalazi na poziciji $2i+1$, a desno dete na poziciji $2i+2$, dok mu se roditelj nalazi na poziciji $\left\lfloor \frac{i-1}{2} \right\rfloor$.

```
int roditelj(int i) {
  return (i-1) / 2;
}

int levoDete(int i) {
  return 2*i + 1;
}

int desnoDete(int i) {
  return 2*i + 2;
}
```

Najveci element se uvek nalazi u korenu, tj. na poziciji 0 (što se lako može dokazati indukcijom imajući u vidu da je vrednost u svakom čvoru veća od vrednosti njegovoj u deci).

```
int najveci(const vector<int>& hip) {
  return hip[0];
}
```

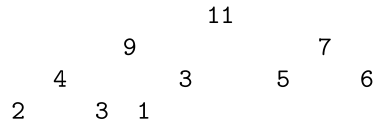
Vreme potrebno za ovo je očigledno $O(1)$.

Razmotrimo kako bismo mogli realizovati operaciju uklanjanja najvećeg elementa iz maks-hipa. Pošto se on nalazi u korenu, a drvo mora biti potpuno popunjeno (osim eventualno poslednjeg nivoa) na mesto izbačenog elementa je najjednostavnije upisati poslednji element iz hipa (najdešnji element poslednjeg nivoa). Ovim je zadovoljen uslov za raspored elemenata u drvetu, ali je svojstvo hipa moguće narušeno, jer taj element ne mora biti veći od svih ostalih (i obično nije). Na sreću, popravku možemo izvršiti relativno jednostavno. Potrebno je da uporedimo vrednost u korenu sa vrednošću njegove dece (ako postoje). Ako je vrednost u korenu veća ili jednaka od tih vrednosti, onda koren zadovoljava uslov maks-hipa i procedura može da se završi (jer za sve ostale čvorove znamo da zadovoljavaju taj uslov, jer je izbacivanje krenulo od ispravnog hipa). U suprotnom, menjamo vrednost u korenu sa većom od vrednosti njegove dece (tj. sa vrednošću njegovog deteta, ako ima samo jedno dete). Nakon toga koren zadovoljava uslov maks-hipa, i preostaje jedino da proverimo (i eventualno popravimo) ono pod drvo u čijem je korenu završila vrednost korena. Ovo je problem istog oblika, samo manje dimenzije u odnosu na polazni i lako se, dakle, rešava induktivno-rekurzivnom konstrukcijom.

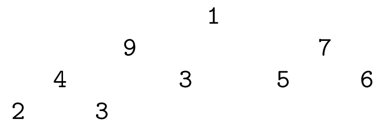
```
// element na poziciji k se pomera naniže, razmenom sa svojom decom
// sve dok se ne zadovolji uslov hipa
void pomeriNanize(vector<int>& hip, int k) {
  // pozicija najvećeg čvora (razmatrajući roditelja i njegovu decu)
  int najveci = k;
  int levo = levoDete(k), desno = desnoDete(k);
  if (levo < hip.size() && hip[levo] > hip[najveci])
    najveci = levo;
  if (desno < hip.size() && hip[desno] > hip[najveci])
    najveci = desno;
  // ako roditelj nije najveći
  if (najveci != k) {
    // menjamo roditelja i veće dete
    swap(hip[najveci], hip[k]);
    // rekurzivno obrađujemo veće dete
    pomeriNanize(hip, najveci);
  }
}

// izbacivanje najvećeg elementa iz hipa
int izbaciNajveci(vector<int>& hip) {
  // poslednji element izbacujemo iz hipa i
  // upisujemo ga na početnu poziciju
  hip[0] = hip.back();
  hip.pop_back();
  // pomeramo početni element naniže dok se ne
  // zadovolji uslov hipa
  pomeriNanize(hip, 0);
```

```
}
```

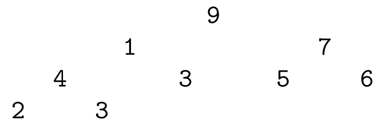
Prikažimo rad ovog algoritma na jednom primeru.



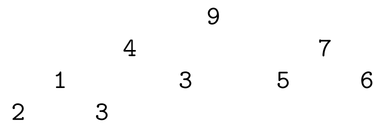
Izbacujemo element sa vrha i na njegovo mestu premeštamo poslednji element.



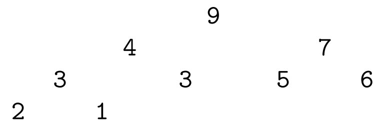
Menjamo vrednost u korenu, sa većom od vrednosti njegova dva deteta.



Isti postupak primenjujemo na levo poddrvo. Razmenjujemo vrednost u korenu sa većom od vrednosti njegova dva deteta.



Postupak se još jednom primenjuje na poddrvo sa korenom `1`.



Nakon ovoga, dobijeno drvo predstavlja maks-hip.

Broj koraka pomeranja naniže u najgorem slučaju odgovara visini drveta. Pošto u potpunom drvetu visine h može da stane $2^{h+1} - 1$ elemenata, visina logaritamski zavisi od broja elemenata. Dakle, vreme potrebno za uklanjanje najvećeg elementa iz hipa u kojem se nalazi n elemenata je $O(\log n)$.

Umetanje novog elementa funkcioniše po sličnom, ali obrnutom principu. Element je najjednostavnije ubaciti na kraj niza. Ipak, moguće je da mu tu nije mesto, jer je možda veći od svog roditelja. U tom slučaju moguće je izvršiti njihovu razmenu. Nakon zamene, poddrvo T sa korenom u novom čvoru zadovoljava uslov hipa i celo drvo bez poddrveta T takođe zadovoljava uslov hipa. Svakim pomeranjem novog elementa naviše, ostatak drveta bez poddrveta T se smanjuje i

problem se svodi na problem manje dimenzije i rešava se induktivno-rekurzivnom konstrukcijom. Jedna moguća implementacija je data u nastavku.

```
// element na poziciji k se pomera naviše, razmenom sa svojim
// roditeljem, sve dok se ne zadovolji uslov hipa
void pomeriNavise(vector<int>& hip, int k) {
  // pozicija roditelja čvora k
  int r = roditelj(k);
  // ako čvor k nije koren i ako je veći od roditelja
  if (k > 0 && hip[k] > hip[r]) {
     // razmenjujemo ga sa njegovim roditeljem
     swap(hip[k], hip[r]);
     // pomeramo roditelja navise
     pomeriNavise(hip, r);
  }
}

// ubacuje se element x u hip
void ubaci(int x) {
  // element dodajemo na kraj
  hip.push_back(x);
  // pomeramo ga naviše dok se ne zadovolji uslov hipa
  pomeriNavise(hip, hip.size() - 1);
}
```

Pošto je i u slučanju operacije pomeranja naniže i u slučaju operacije pomeranja naviše rekurzija repna, ona se može relativno jednostavno ukloniti.

```
void pomeriNavise(vector<int>& hip, int k) {
  int r = roditelj(k);
  while (k > 0 && hip[k] > hip[r]) {
     swap(hip[k], hip[r]);
     k = r;
     r = roditelj(k);
  }
}

void pomeriNanize(vector<int>& hip, int k) {
  while (true) {
    // pozicija najvećeg čvora
    // (razmatrajući roditelja i njegovu decu)
    int najveci = k;
    int levo = levoDete(k), desno = desnoDete(k);
    if (levo < hip.size() && hip[levo] > hip[najveci])
      najveci = levo;
    if (desno < hip.size() && hip[desno] > hip[najveci])
      najveci = desno;
```

```
    // ako je roditelj najveći, uslov hipa je zadovoljen
    if (najveci == k)
      break;
    // menjamo roditelja i veće dete
    swap(hip[najveci], hip[k]);
    // obrađujemo veće dete
    k = najveci;
  }
}
```

Još jedan način da implementiramo pomeranje naniže je da odredimo poziciju veceg od dva deteta i onda to dete uporedimo sa roditeljem.

```
void pomeriNanize(vector<int>& hip, int k) {
  int roditelj = k;
  // pretpostavljamo da je levo dete vece
  int veceDete = levoDete(k);
  // dok god čvor ima dece
  while (veceDete < hip.size()) {
    // poredimo da li je desno dete vece od levog
    int desno = veceDete + 1;
    if (desno < hip.size() && hip[desno] > hip[veceDete])
       veceDete = desno;
    // ako je roditelj veći ili jednak od oba deteta,
    // uslov hipa je zadovoljen
    if (hip[roditelj] >= hip[veceDete])
      break;
    // menjamo roditelja i veće dete
    swap(hip[veceDete], hip[roditelj]);
    // nastavljamo obradu od većeg deteta
    roditelj = veceDete;
    veceDete = levoDete(roditelj);
  }
}
```

Sortiranje uz pomoć hipa (heap sort)

U mnogim situacijama zahteva se formiranje hipa od elemenata nekog unapred zadatog niza. Na primer, algoritam sortiranja pomoću hipa (engl. Heap sort) predstavlja varijantu sortiranja selekcijom u kome se svi elementi niza postavljaju u maks-hip, a zatim se iz maks-hipa vadi jedan po jedan element i prebacuje ga na kraj niza. Interesanto, isti niz (ili vektor) se može koristiti i za smeštanje polaznog niza i za smeštanje hipa i za smeštanje sortiranog niza (elementi hipa se mogu smestiti u početni deo niza, dok se sortirani deo može smestiti u krajnji deo niza). Time se štedi memorijski prostor, međutim, potrebno je malo prilagoditi funkcije za rad sa hipom. Naime, funkcija za pomeranje naniže pored niza ili

vektora u kome su smešteni elementi, moraju kao parametar da primi i broj elemenata niza ili vektora koji predstavlja hip (jer stvarna dimenzija može biti i veća, ako se prostor iza hipa koristi za smeštanje elemenata sortiranog niza). Na osnovu tog broja je lako ustanoviti koji elementi niza pripadaju, a koji ne pripadaju hipu. Poređenje sa `hip.size()` se mora zameniti poređenjem sa tim brojem.

Jedan način da se od niza formira hip (u istom memorijskom prostoru) je da se krene od praznog hipa i da se jedan po jedan element ubacuje u hip.

```
// formira hip od elemenata niza a dužine n
// hip se smešta u istom memorijskom prostoru kao i niz
void formirajHip(int a[], int n) {
  // ubacujemo jedan po jedan element u hip i pomeramo ga naviše
  for (int i = 1; i < n; i++)
     pomeriNavise(a, i);
}
```

Invarijanta spoljne petlje (tj. induktivna hipoteza) je to da elementi na pozicijama $[0, i)$ čine maks-hip. Pošto znamo da operacija pomeranja elementa naviše ispravno umeće poslednji element u postojeći maks-hip, lako je dokazati da invarijanta ostaje održana. Na kraju petlje je $i = n$, što znači da su svi elementi niza (elementi na pozicijama $[0, n)$) složeni u maks-hip. Dodatno, razmene ne menjaju multiskup elemenata niza, pa je multiskup elemenata u hipu jednak multiskupu elemenata polaznog niza.

Ovaj način se naziva formiranje hipa naniže ili Vilijamsov metod. Složenost najgoreg slučaja formiranja hipa jednaka je $\Theta(n \log n)$. Naime, složenost operacije umetanja tj. pomeranja elementa naviše u hipu koji ima k elemenata jednaka je $O(\log k)$, pa je ukupna složenost formiranja hipa asimptotski jednaka zbiru svih logaritama od 1 do n, što je, kako smo ranije videli $\Theta(n \log n)$.

Drugi, efikasniji način formiranja hipa naziva se formiranje hipa naviše ili Flojdov metod. Ideja je da se elementi originalnog niza obilaze od pozadi i da se svaki element umetne u hip čiji koren predstavlja, tako što se spusti naniže kroz hip. Induktivna hipoteza u ovom pristupu je to da su svi elementi iz intervala (i, n) koreni ispravnih maks-hipova. Na primer, ako je dat niz

```
7 2   8 5 4 6 5 1 3
```

kada je $i = 1$, svi elementi osim prva dva čine korenove ispravnih hipova. Zaista, u nizu je smešteno sledeće drvo.

```
           7
       2       8
    5     4  6    5
  1   3
```

Elementi od 4 do 3 nemaju naslednike i trivijalno predstavljaju ispravne jednočlane hipove. I u opštem slučaju, svi elementi u drugoj polovini niza pred-

stavljaju listove i za njih je ova invarijanta trivijalno ispunjena. Element 5 je veći od oba svoja sina, pa je koren ispravnog maks-hipa. Slično, element 8 je veći od oba svoja sina, pa je i on koren ispravnog maks-hipa.

Postavlja se pitanje kako proširiti invarijantu. Element na poziciji i ne mora da bude veći od svoje dece, ali znamo da su oba njegova deteta koreni ispravnih hipova. Stoga je samo potrebno spustiti element sa vrha na njegovo mesto, što je operacija koju smo već razmatrali prilikom operacije brisanja elementa iz hipa.

```
// formira hip od elemenata niza a dužine n
// hip se smešta u istom memorijskom prostoru kao i niz
void formirajHip(int a[], int n) {
   // sve elemente osim listova pomeramo naniže
   for (int i = n/2; i >= 0; i--)
      pomeriNanize(a, n, i);
}
```

Ocenimo složenost ovog algoritma. Prvo, veoma je zgodno to što je jedna polovina elemenata niza već na svom mestu. Supštanje elemenata niz hip ima složenost $O(\log n)$, pa je ovde ponovo u pitanju neki zbir logaritamskih složenosti i na prvi se pogled može pomisliti da će i ovde složenost biti $\Theta(n \log n)$ - ona će svakako biti $O(n \log n)$, ali ćemo pokazati da će biti niža tj. $\Theta(n)$. Naime, nije svaki naredni logaritam u zbiru za jedan veći od prethodnog, kako je to bio slučaj prilikom pomeranja elemenata naviše. Ako bismo razmatrali hip u kome su svi nivoi potpuno popunjeni, postojala bi jedna pozicija sa koje bi se element spuštao celom visinom hipa, dve pozicije sa kojih bi se element spuštao visinom koja je za jedan manja, četiri elementa sa kojih je visina za 2 manja i tako dalje. Ukupan broj koraka za hip visine i bi bio $i + 2(i-1) + 4(i-2) + \ldots 2^{i-1} \cdot 1$ tj.

$$\sum_{k=0}^{i-1} 2^k (i-k).$$

Do ovog rezultata možemo doći i razmatranjem sledeće rekurentne jednačine. Broj koraka pomeranja naniže odgovara zbiru visina svih čvorova u drvetu. Neka je $H(i)$ zbir svih visina potpunog binarnog drveta visine i. Tada je $H(i) = 2H(i-1) + i$, jer se potpuno binarno drvo sastoji od dva potpuna binarna drveta visine $i-1$ i korena visine i. Važi i $H(0) = 0$. Razmotajmo ovu rekurentnu jednačinu.

$$
\begin{aligned}
H(i) &= 2H(i-1) + i \\
&= 2(2H(i-2) + (i-1)) + i = 4H(i-2) + 2(i-1) + i \\
&= 4(2H(i-3) + (i-2)) + 2(i-1) + i = 8H(i-3) + 4(i-2) + 2(i-1) + i \\
&= \ldots \\
&= 2^i H(0) + \sum_{k=0}^{i-1} 2^k (i-k) = \sum_{k=0}^{i-1} 2^k (i-k) \\
&= i \sum_{k=0}^{i-1} 2^k - \sum_{k=0}^{i-1} k 2^k = i(2^i - 1) - ((i-1)-1)2^{(i-1)+1} - 2 = 2^{i+1} - i - 2
\end{aligned}
$$

Poslednje važi jer znamo da je

$$
\sum_{k=0}^{n} 2^k = 2^{n+1} - 1, \quad \sum_{k=0}^{n} k 2^k = (n-1)2^{n+1} + 2.
$$

Prvo sledi na osnovu formule za zbir geometrijskog reda, a drugo njenim diferenciranjem i množenjem sa x.

Postoji i lakši način da se dođe do istog rezultata. Naime, jednačinu $H(i) = 2H(i-1)+i$ možemo zapisati kao $H(i)+i = 2(H(i-1)+(i-1))+2$ i onda uvesti smenu $G(i) = H(i) + i$. Zato je $G(i) = 2G(i-1) + 2$ i $G(0) = 0$. Odmotavanjem sada dobijamo

$$
\begin{aligned}
G(i) &= 2G(i-1) + 2 \\
&= 2(2G(i-2) + 2) + 2 = 4G(i-2) + 2 \cdot 2 + 2 \\
&= 4(2G(i-3) + 2) + 2 \cdot 2 + 2 = 8G(i-3) + 4 \cdot 2 + 2 \cdot 2 + 2 \\
&= \ldots \\
&= 2^i G(0) + 2 \sum_{k=0}^{i-1} 2^k = 2^{i+1} - 2
\end{aligned}
$$

Zato je $H(i) = G(i) - i = 2^{i+1} - i - 2$.

Pošto je $G(i) + 2 = 2(G(i-1) + 2)$, smenom $F(i) = G(i) + 2$ bi se dobilo da je $F(i) = 2F(i-1)$ i $F(0) = 2$, pa bi skoro direktno sledilo da je $F(i) = 2^{i+1}$ i $G(i) = F(i) - 2 = 2^{i+1} - 2$.

Od svih ovih kompleksnih izvođenja bitan nam je samo krajnji rezultat, a to je da je $H(i) = 2^{i+1} - i - 2$. Pošto je za potpuno drvo broj elemenata niza $n = 2^i - 1$, važi da je $H(i) = 2(n+1) - i - 2 = 2n - i$ i važi da je složenost

konstrukcije odozdo $\Theta(n)$. Dakle, formiranje hipa naviše ima asiptotski bolju složenost nego formiranje hipa naniže!

Ispitajmo koliko je ta razlika u praksi značajna. Za niz od 10^9 nasumično odabarnih brojeva formiranje naviše traje oko $1,902$ sekundi, a formiranje naniže oko $1,943$ sekundi. Razlika je praktično beznačajna, naročito u svetlu toga što kasnije vađenje elemenata iz hipa koje takođe ima asimptotsku složenost $\Theta(n \log n)$ traje dodatnih oko 50 sekundi i apsolutno dominira sortiranjem. Stvar je malo drugačija kada se krene od već sortiranog niza, jer tada formiranje naviše traje oko $0,782$ sekunde, a naniže oko $2,534$ sekunde, međutim, opet kasnija faza dominira i traje oko 11 dodatnih sekundi, pa ušteda ni u ovom slučaju nije naročito značajna. Objašnjenje možemo naći u tome što se kod operacije pomeranja naniže vrši više poređenja nego kod operacije pomeranja elemenata naviše, tako da iako se operacija pomeranja naviše malo češće vrši, ona je sama po sebi brža. Takođe, razlika u asimptotskim klasama n i $n \log n$ je prilično mala i nije je uvek lako uhvatiti u praksi, jer konstante koje zanemarujemo mogu značajno uticati na vreme izvršavanja.

Prikažimo na kraju i implementaciju faze sortiranja vađenjem elemenata hipa koja sledi nakon formiranja hipa. Invarijanta ove faze biće da se u nizu na pozicijama $[0, i]$ nalaze elementi ispravno formiranog hipa, a da se u delu (i, n) nalaze sortirani elementi koji su svi veći ili jednaki od elemenata koji se trenutno nalaze u hipu. Na početku je $i = n - 1$, pa je invarijanta trivijalno zadovoljena (elementi u intervalu $[0, n-1]$ čine ispravan hip, dok je interval $(n-1, n)$ prazan). U svakom koraku se najveći element hipa (element na poziciji 0) izbacuje iz hipa i dodaje na početak sortiranog dela niza (na poziciju i). Element sa pozicije i koji je ovom razmenom završio na vrhu hipa se pomera naniže, sve dok se zadovolji uslov hipa. Vrednost i se smanjuje za 1, čime se ispravno održava granica između hipa i sortiranog dela niza (hip je za jedan element kraći, a sortirani deo niza je za jedan element duži). Kada se petlja završi tada je $i = 0$, pa iz invarijante sledi da je niz ispravno sortiran (svi elementi na pozicijama $(0, n)$ su sortirani i veći ili jednaki elementu na poziciji 0). Pošto se vrše samo razmene elemenata multiskup elemenata niza se ne menja.

```
// sortira se niz a dužine n
void hipSort(int a[], int n) {
  // formiramo hip na osnovu elemenata niza
  formirajHip(a, n);
  for (int i = n-1; i > 0; i--) {
    // najveći element vadimo iz hipa i ubacujemo ga na početak
    // sortiranog dela niza
    swap(a[0], a[i]);
    // pomeramo koren hipa naniže, popravljajući hip
    // nakon izbacivanja največeg, broj elemenata hipa jednak je i
    pomeriNanize(a, i, 0);
  }
}
```

Heširanje

Čas 6.1, 6.2, 6.3 - još neke korisne struktre podataka

Prefiksno drvo

Uređena binarna drveta omogućavaju efikasnu implementaciju struktura sa asocijativnim pristupom kod kojih se pristup vrši po ključu koji nije celobrojna vrednost, već string ili nešto drugo. Još jedna struktura u vidu drveta koja omogućava efikasan asocijativni pristup je *prefiksno drvo* takođe poznato pod engleskim nazivom *trie* (od engleske reči *reTRIEeval*). Osnovna ideja ove strukture je da svaka putanja od korena do lista kodira jedan ključ, a da se odgovarajući podaci vezani za taj ključ čuvaju u listu. Ipak, moguće je i da se putanja koja kodira neki ključ završi i u unutrašnjem čvoru, što će biti ilustrovano primerom. U slučaju niski, koren sadrži praznu reč, a prelaskom preko svake grane se na do tada formiranu reč nadovezuje još jedan karakter. Pritom, zajednički prefiksi različitih ključeva su predstavljeni istim putanjama od korena do tačke razlikovanja. Jedan primer ovakvog drveta, kod kojeg prikazane oznake pripadaju granama, a ne čvorovima, je sledeći:

```
      a      n     d
    n   t    o   a   u
   a       ć ž  n  h ž
```

Ključevi koje drvo čuva su *ana*, *at*, *noć*, *nož*, *da*, *dan*, *duh* i *duž*. Primetimo da se ključ *da* ne završava listom i da stoga svaki čvor mora čuvati informaciju o tome da li se njime kompletira neki ključ (i u tom slučaju sadržati podatak) ili ne. Ilustracije radi, mogu se prikazati oznake na čvorovima, koje predstavljaju prefikse akumulirane do tih čvorova. Treba imati u vidu da ovaj prikaz ne ilustuje implementaciju, već samo prefikse duž grane. Simbol @ predstavlja praznu reč.

```
                @
    a           n          d
 an   at         no       da       du
ana            noć  nož  dan  duh  duž
```

U slučaju da neki čvor ima samo jednog potomka i ne predstavlja kraj nekog ključa, grana do njega i grana od njega se mogu spojiti u jednu, njihovi karakteri nadovezati, a čvor eliminisati. Ovkao se dobija kompaktnija reprezentacija prefiksnog drveta.

Pored opšteg asocijativnog pristupa podacima, očigledna primena ove strukture je i implementacija konačnih rečnika, na primer u svrhe automatskog kompletiranja ili provere ispravnosti reči koje korisnik kuca na računaru ili mobilnom telefonu.

Napomenimo još i da ova struktura nije rezervisana za čuvanje stringova. Na

primer, u slučaju celih brojeva ili brojeva u pokretnom zarezu, ključ mogu biti niske bitova koje predstavljaju takve brojeve.

U slučaju konačne azbuke veličine m, složenost operacija u najgorem slučaju je $O(mn)$, gde je n dužina reči koja se traži, umeće ili briše. Pretraga i umetanje se pravolinijski implementiraju, dok je u slučaju brisanja nekada potrebno brisati više od jednog čvora.

Ukoliko su ključevi relativno kratki, prednost prefiksnog drveta je što složenost zavisi od dužine zapisa ključa, a ne od broja elemenata u drvetu. Mana je potreba za čuvanjem pokazivača uz svaki karakter u drvetu.

U nastavku su date implementacije osnovnih operacija sa prefiksnim drvetom na primeru formiranja i pretrage rečnika.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// struktura čvora prefiksnog drveta u svakom čvoru čuvamo niz
// grana obeleženih karakterima ka potomcima i informaciju da li
// je u ovom čvoru kraj neke reči

struct node;

typedef vector<pair<char, node*>> edgeList;

struct node {
  bool keyEnding = false;
  edgeList edges;
};


// tražimo sufiks reči w koji počinje od pozicije i u drvetu na
// čiji koren ukazuje pokazivač trie
bool find(node *trie, const string& w, int i) {
  // ako je sufiks prazan, on je u korenu akko je u korenu obeleženo
  // da je tu kraj reči
  if (i == w.size())
    return trie->keyEnding;

  // prolazimo kroz sve grane iz korena tražeći slovo w[i]
  const edgeList& edges = trie->edges;
  for (int j = 0; j < edges.size(); j++)
    // ako nađemo granu na kojoj piše w[i]
    if (edges[j].first == w[i])
      // rekurzivno tražimo ostatak sufiksa od pozicije i+1
```

```
      return find(edges[j].second, w, i+1);

  // nismo našli granu sa w[i], pa reč ne postoji
  return false;
}

// tražimo u drvetu na čiji koren ukazuje pokazivač trie reč w
bool find(node *trie, const string& w) {
  return find(trie, w, 0);
}

// umeće sufiks reči w od pozicije i u drvo na čiji koren ukazuje
// pokazivač trie
void insert(node *trie, const string& w, int i) {
  // ako je sufiks prazan samo u korenu beležimo da je tu kraj reči
  if (i == w.size()) {
    trie->keyEnding = true;
    return;
  }

  // prolazimo kroz grane tražeći onu obeleženu sa w[i]
  int j;
  edgeList& edges = trie->edges;
  for (j = 0; j < edges.size(); j++)
    if (edges[j].first == w[i])
      break;
  // ako takva grana ne postoji, dodajemo je kreirajući novi čvor
  if (j == edges.size())
    edges.push_back(pair<char, node*>(w[i], new node()));

  // sada znamo da grana sa w[i] sigurno postoji i preko te grane
  // nastavljamo dodavanje sufiksa koji počinje na i+1;
  insert(edges[j].second, w, i+1);
}

// umeće reč w u drvo na čiji koren ukazuje pokazivač w
void insert(node *trie, string& w) {
  return insert(trie, w, 0);
}

// program kojim testiramo gornje funkcije
int main() {
  node* trie = new node();
  vector<string> words
    {"ana", "at", "noc", "noz", "da", "dan", "duh", "duz"};
  vector<string> words_neg
```

```
    {"", "a", "d", "ananas", "marko", "ptica"};
  for(auto w : words)
    insert(trie, w);
  for(auto w : words)
    cout << w << ": " << (find(trie, w) ? "da" : "ne") << endl;
  for(auto w : words_neg)
    cout << w << ": " << (find(trie, w) ? "da" : "ne") << endl;
  return 0;
}
```

Disjunktni podskupovi (union-find)

Ponekada je potrebno održavati u programu nekoliko disjunktnih podskupova određenog skupa, pri čemu je potrebno moći za dati element efikasno pronaći kom skupu pripada (tu operaciju zovemo `find`) i efikasno spojiti dva zadata podskupa u novi, veći podskup (tu operaciju zovemo `union`). Pomoću operacije `find` lako možemo za dva elementa proveriti da li pripadaju istom podskupu tako što za svaki od njih pronađemo oznaku podskupa i proverimo da li su one jednake.

Jedna moguća implementacija je da se održava preslikavanje svakog elementa u oznaku podskupa kojem pripada. Ako pretpostavimo da su svi elementi numerisani brojevima od 0 do $n-1$, onda ovo preslikavanje možemo realizovati pomoću običnog niza gde se na poziciji svakog elementa nalazi oznaka podskupa kojem on pripada. Operacija `find` je tada trivijalna (samo se iz niza pročita oznaka podskupa) i složenost joj je $O(1)$. Operacija `union` je mnogo sporija jer zahteva da se oznake svih elemenata jednog podskupa promene u oznake drugog, što zahteva da se prođe kroz ceo niz i složenosti je $O(n)$.

```
int id[MAX_N];
int n;

void inicijalizuj() {
  for (int i = 0; i < n; i++)
    id[i] = i;
}

int predstavnik(int x) {
   return id[x];
}

int ekvalentni(int x, int y) {
  return find(x) == find(y);
}

int unija(int x, int y) {
```

```
  int idx = id[x], idy = id[y];
  for (int i = 0; i < n; i++)
    if (id[i] == idx)
      id[i] = idy;
}
```

Ključna ideja je da elemente ne preslikavamo u oznake podskupova, već u da podskupove čuvamo u obliku drveta tako da svaki element slikamo u njegovog roditelja u drvetu. Korene drveta ćemo slikati same u sebe i smatrati ih oznakama podskupova. Dakle, da bismo na osnovu proizvoljnog elementa saznali oznaku podskupa kom pripada, potrebno je da prođemo kroz niz roditelja sve dok ne stignemo do korena. Naglasimo da su u ovim drvetima pokazivači usmereni od dece ka roditeljima, za razliku od klasičnih drveta gde pokazivači ukazuju od roditelja ka deci.

Prvi algoritam odgovara situaciji u kojoj osoba koja promeni adresu obaveštava sve druge osobe o svojoj novoj adresi. Drugi odgovara situaciji u kojoj samo na staroj adresi ostavlja informaciju o svojoj novoj adresi. Ovo, naravno, malo usporava dostavu pošte, jer se mora preći kroz niz preusmeravanja, ali ako taj niz nije predugačak, može biti značajno efikasnije od prvog pristupa.

Uniju možemo vršiti tako što koren jednog podskupa usmerimo ka korenu drugog.

```
int roditelj[MAX_N];
int n;

void incijalizuj() {
  for (int i = 0; i < n; i++)
    roditelj[i] = i;
}

int predstavnik(int x) {
  while (roditelj[x] != x)
    x = roditelj[x];
  return x;
}

int unija(int x, int y) {
  int fx = predstavnik(x), fy = predstavnik(y);
  roditelj[fx] = fy;
}
```

Složenost prethodnog pristupa zavisi od toga koliko su drveta kojima se predstavljaju podskupovi balansirana. U najgorem slučaju se ona mogu izdegenerisati u listu i tada je složenost svake od operacija $O(n)$. Iako ovo deluje lošije od prethodnog pristupa, gde je bar pronalaženje podskupa koštalo $O(1)$, kada su drveta izbalansirana, tada je složenost svake od operacija $O(\log n)$ i centralni zadatak da bi se na ovoj ideji izgradila efikasna struktura podataka je da se

nekako obezbedi da drveta ostanu izbalansirana. Ključna ideja je da se prilikom izmena (a one se vrše samo u sklopu operacije unije) ako je moguće obezbedi da se visina drveta kojim je predstavljena unija ne poveća u odnosu na visine pojedinačnih drveta koja predstavljaju skupove koji se uniraju (visinu možemo definisati kao broj grana na putanji od tog čvora do njemu najudaljenijeg lista). Prilikom pravljenja unije imamo slobodu izbora korena kog ćemo usmeriti prema drugom korenu. Ako se uvek izabere da koren plićeg drveta usmeravamo ka dubljem, tada će se visina unije povećati samo ako su oba drveta koja uniramo iste visine. Visinu drveta možemo održavati u posebnom nizu koji ćemo iz razloga koji će biti kasnije objašnjeni biti nazvani `rang`.

```
int roditelj[MAX_N]; int n;
int rang[MAX_N];

void inicijalizuj() {
  for (int i = 0; i < n; i++) {
    roditelj[i] = i;
    rang[i] = 0;
  }
}

int predstavnik(int x) {
  while (roditelj[x] != x)
    x = roditelj[x];
  return x;
}

int unija(int x, int y) {
  int fx = predstavnik(x), fy = predstavnik(y);
  if (rang[fx] < rang[fy])
     roditelj[fx] = fy;
  else if (rang[fy] < rang[fx])
     roditelj[fy] = fx;
  else {
     roditelj[fx] = fy;
     rang[fx]++;
  }
}
```

Prikažimo rad algoritma na jednom primeru. Podskupove ćemo predstavljati drvetima.

```
1  2  3  4  5  6  7  8

unija 1 2

1  3  4  5  6  7  8
```

```
2

unija 6 7

1  3  4  5  6  8
2           7

unija 4 7

1  3  5     6    8
2         4   7

unija 5 8

1  3    6     8
2     4   7   5

unija 1 3

  1       6     8
2   3   4   7   5

unija 5 4

  1       6
2   3   4  7  8
              5

unija 3 7

       6
  1  4  7  8
2   3     5
```

Dokažimo indukcijom da se u drvetu čiji je koren na visini h nalazi bar 2^h čvorova. Baza je slučaj početni u kome je svaki čvor svoj predstavnik. Visina svih čvorova je tada nula i sva drveta imaju $2^0 = 1$ čvor. Svaka unija održava ovu invarijantu. Po induktivnoj hipotezi pretpostavljamo da oba drveta koje predstavljaju podskupove koji se uniraju imaju visine h_1 i h_2 i bar 2^{h_1} i 2^{h_2} čvorova. Ukoliko se uniranjem visina ne poveća, invarijanta je očuvana jer se broj čvorova uvećao. Jedini slučaj kada se povećava visina unije je kada je $h_1 = h_2$ i tada unirano drvo ima visinu $h = h_1 + 1 = h_2 + 1$ i bar $2^{h_1} + 2^{h_2} = 2^h$ čvorova. Time je tvrđenje dokazano. Dakle, složenost svake operacije pronalaženja predstavnika u skupu od n čvorova je $O(\log n)$, a pošto uniranje nakon pronalaženja predstavnika vrši još samo $O(1)$ operacija, i složenost nalaženja unije je $O(\log n)$.

Recimo i da je umesto visine moguće održavati i broj čvorova u svakom od

podskupova. Ako uvek usmeravamo predstavnika manjeg ka predstavniku većeg podskupa, ponovo ćemo dobiti logaritamsku složenost najgoreg slučaja za obe operacije.

Iako je ova složenost sasvim prihvatljiva (složenost nalaženja n unija je $O(n \log n)$), može se dodatno poboljšati veoma jednostavnom tehnikom poznatom kao *kompresija putanje*. Naime, prilikom pronalaženja predstavnika možemo sve čvorove kroz koje prolazimo usmeriti ka korenu. Jedan način da se to uradi je da se nakon pronalaženja korena, ponovo prođe kroz niz roditelja i svi pokazivači usmere ka korenu.

```
int predstavnik(int x) {
  int koren = x;
  while (koren != roditelj[koren])
    koren = roditelj[koren];
  while (x != koren) {
    int tmp = roditelj[x];
    roditelj[x] = koren;
    x = tmp;
  }
  return koren;
}
```

Za sve čvorove koji se obilaze od polaznog čvora do korena, dužine putanja do korena se nakon ovoga smanjuju na 1, međutim, kao što primećujemo, niz rangova se ne menja. Ako rangove tumačimo kao visine, jasno je da prilikom kompresije putanje niz visina postaje neažuran. Međutim, interesantno je to da nema potrebe da se on ažurira. Nime, brojevi koji se sada čuvaju u tom nizu se više ne smatraju visinama čvorova, već prosto njihovim rangovima tj. pomoćnim podacima koji nam pomažu da preusmerimo čvorove prilikom uniranja. Pokazuje se da se ovim ne narušava složenost najgoreg slučaja i da funkcija nastavlja korektno da radi. Ako se umesto visine koristi broj čvorova u drvetu, onda se prilikom kompresije staze ta statistika ne menja, pa je obrazloženje korektnosti malo jednostavnije.

U prethodnoj implementaciji se dva puta prolazi kroz putanju od čvora do korena. Ipak, slične performanse se mogu dobiti i samo u jednom prolazu ako se svaki čvor usmeri ka roditelju svog roditelja. Za sve čvorove koji se obilaze od polaznog čvora do korena, dužine putanja do korena se nakon ovoga smanjuju dvostruko, što je dovoljno za odlične performanse.

```
int predstavnik(int x) {
  while (x != roditelj[x]) {
    x = roditelj[x]
    roditelj[x] = roditelj[roditelj[x]]
  }
  return x;
}
```

Primetimo da je ovim dodata samo jedna linija koda u prvobitnu implementaciju. Ovo jednostavnom promenom agregatna složenost operacija postaje samo $O(\alpha(n))$, gde je $\alpha(n)$ inverzna Akermanova funkcija koja strašno sporo raste. Za bilo koji broj n koji je manji od broja atoma u celom univerzumu važi da je $\alpha(n) < 5$, tako da je vreme praktično konstantno.

Problem: Logička matrica u početku sadrži sve nule. Nakon toga se nasumično dodaje jedna po jedna jedinica. Kretanje po matrici je moguće samo po jedinicama i to samo na dole, na gore, na desno i na levo. Napisati program koji učitava dimenziju matrice, a zatim poziciju jedne po jedne jedinice i određuje nakon koliko njih je prvi put moguće sići od vrha do dna.

Osnovna ideja je da se formiraju svi podskupovi elemenata između koji postoji put. Kada se uspostavi veza između dva elementa takva dva podskupa, podskupovi se spajaju. Provera da li postoji put između dva elementa svodi se onda na proveru da li oni pripadaju istom podskupu. Podskupove možemo čuvati na način koji smo opisali. Putanja od vrha do dna postoji ako postoji putanja od bilo kog elementa u prvom redu matrice do bilo kog elementa u dnu matrice. To bi dovelo do toga da u svakom koraku moramo da proveravamo sve parove elemenata iz gornjeg i donjeg reda. Međutim, možemo i bolje. Dodaćemo veštački početni čvor (nazovimo ga izvor) i spojićemo ga sa svim čvorovima u prvoj vrsti matrice i završni čvor (nazovimo ga ušće) i spojićemo ga sa svim čvorovima u poslednjoj vrsti matrice. Tada se u svakom koraku samo može proveriti da li su izvor i ušće spojeni.

```
// redni broj elementa (x, y) u matrici
int kod(int x, int y, int n) {
  return x*n + y;
}

int put(int n, const vector<pair<int, int>>& jedinice) {
  // alociramo matricu n*n
  vector<vector<int>> a(n);
  for (int i = 0; i < n; i++)
    a[i].resize(n);

  // dva dodatna veštačka čvora
  const int izvor = n*n;
  const int usce = n*n+1;

  // inicijalizujemo union-find strukturu za sve elemente matrice
  // (njih n*n), izvor i ušće
  inicijalizacija(n*n + 2);

  // spajamo izvor sa svim elementima u prvoj vrsti matrice
  for (int i = 0; i < n; i++)
    unija(izvor, kod(0, i, n));
```

```
  // spajamo sve elemente u poslednjoj vrsti matrice sa ušćem
  for (int i = 0; i < n; i++)
    unija(kod(n-1, i), usce);

  // broj obrađenih jedinica
  int k = 0;
  while (k < jedince.size()) {
    // čitamo narednu jedinicu
    int x = jedince[k].first, y = jedince[k].second;
    k++;
    // ako je u matrici već jedinica, nema šta da se radi
    if (a[x][y] == 1) continue;
    // upisujemo jedinicu u matricu
    a[x][y] = 1;
    // povezujemo podskupove u sva četiri smera
    if (x > 0 && a[x-1][y])
      unija(kod(x, y, n), kod(x-1, y, n));
    if (x + 1 < n && a[x+1][y])
      unija(kod(x, y, n), kod(x+1, y, n));
    if (y > 0 && a[x][y-1])
      unija(kod(x, y, n), kod(x, y-1, n));
    if (y + 1 > 0 && a[x][y+1])
      unija(kod(x, y, n), kod(x, y+1, n));
    // proveravamo da li su izvor i ušće spojeni
    if (predstavnik(izvor) == predstavnik(usce))
      return k;
  }

  // izvor i ušće nije moguće spojiti na osnovu datih jedinica
  return 0;
}
```

Upiti raspona

Određene strukture podataka su posebno pogodne za probleme u kojima se traži da se nad nizom elemenata izvršavaju upiti koji zahtevaju izračunavanje statistika nekih raspona tj. segmenata niza (engl. range queries).

Problem: Implementirati strukturu podataka koja obezbeđuje efikasno izračunavanje zbirova segmenata datog niza određenih intervalima pozicija $[a, b]$.

Rešenje grubom silom koje bi smestilo sve elemente u klasičan niz i pri svakom upitu iznova računalo zbir elemenata na pozicijama iz datog intervala imalo bi složenost $O(mn)$, gde je m broj upita, a n dužina niza, što je u slučaju dugačkih nizova i velikog broja upita nedopustivo neefikasno. Jednostavno

rešenje je zasnovano na ideji koju smo već ranije razmatrali. Umesto čuvanja elemenata niza, možemo čuvati niz zbirova prefiksa niza. Zbir svakog segmenta $[a, b]$ možemo razložiti na razliku prefiksa do elementa b i prefiksa do elementa $a - 1$. Svi prefiksi se mogu izračunati u vremenu $O(n)$ i smestiti u dodatni (a ako je ušteda memorije bitna, onda čak i u originalni niz). Nakon ovakvog pretprocesiranja, zbir svakog segmenta se može izračunati u vremenu $O(1)$, pa je ukupna složenost $O(n + m)$.

Donekle srodan problem može biti i sledeći.

Problem: Dat je niz dužine n koji sadrži samo nule. Nakon toga izvršavaju se upiti oblika $([a, b], x)$, koji podrazumevaju da se svi elementi na pozicijama iz intervala $[a, b]$ uvećaju za vrednost x. Potrebno je odgovoriti kako izgleda niz nakon izvršavanja svih tih upita.

Rešenje grubom silom bi u svakom koraku u petlji uvećavalo sve elemente na pozicijama $[a, b]$. Složenost tog naivnog pristupa bila bi $O(mn)$, gde je m broj upita, a n dužina niza.

Mnogo bolje rešenje se može dobiti ako se umesto elemenata niza pamte razlike između svaka dva susedna elementa niza. Ključni uvid je da se tokom uvećavanja svih elemenata niza menjaju samo razlike između elemenata na pozicijama a i $a-1$ (ta razlika se uvećava za x) kao i između elemenata na pozicijama $b+1$ i b (ta razlika se umanjuje za x). Ako znamo sve elemente niza, tada niz razlika susednih elemenata možemo veoma jednostavno izračunati u vremenu $O(n)$. Sa druge strane, ako znamo niz razlika, tada originalni niz možemo takođe veoma jednostavno rekonstruisati u vremenu $O(n)$. Jednostavnosti radi, možemo pretpostaviti da početni niz proširujemo sa po jednom nulom sa leve i desne strane.

Može se primetiti da se rekonstrukcija niza vrši zapravo izračunavanjem prefiksnih zbirova niza razlika susednih elemenata, što ukazuje na duboku vezu između ove dve tehnike. Zapravo, razlike susednih elemenata predstavljaju određeni diskretni analogon izvoda funkcije, dok prefiksni zbirovi onda predstavljaju analogiju određenog integrala. Izračunavanje zbira segmenta kao razlike dva zbira prefiksa odgovara Njutn-Lajbnicovoj formuli.

Dakle, niz zbirova prefiksa, omogućava efikasno postavljanje upita nad segmentima niza, ali ne omogućava efikasno ažuriranje elemenata niza, jer je potrebno ažurirati sve zbirove prefiksa nakon ažuriranog elementa, što je naročito neefikasno kada se ažuriraju elementi blizu početka niza (složenost najgoreg slučaja $O(n)$). Niz razlika susednih elemenata dopušta stalna ažuriranja niza, međutim, izvršavanje upita očitavanja stanja niza podrazumeva rekonstrukciju niza, što je složenosti $O(n)$.

Problemi koje ćemo razmatrati u ovom poglavlju su specifični po tome što omogućavaju da se upiti ažuriranja niza i očitavanja njegovih statistika javljaju isprepletano. Za razliku od prethodnih, statičkih upita nad rasponima (engl. static range queries), ovde ćemo razmatrati tzv. dinamičke upite nad rasponima

(engl. dynamic range queries), tako da je potrebno razviti naprednije strukture podataka koje omogućavaju izvršavanje oba tipa upita efikasno. Na primer, razmotrimo sledeći problem.

Problem: Implementirati strukturu podataka koja obezbeđuje efikasno izračunavanje zbirova segmenata datog niza određenih intervalima pozicija $[a, b]$, pri čemu se pojedinačni elementi niza često mogu menjati.

U nastavku ćemo videti dve različite, ali donekle slične strukture podataka koje daju efikasno rešenje prethodnog problema i njemu sličnih.

Segmentna drveta

Jedna struktura podataka koja omogućava prilično jednostavno i efikasno rešavanje ovog problema su *segmentna drveta*. Opet se tokom faze pretprocesiranja izračunavaju zbirovi određenih segmenata polaznog niza, a onda se zbir elemenata proizvoljnog segmenta polaznog niza izražava u funkciji tih unapred izračunatih zbirova. Recimo i da segmentna drveta nisu specifična samo za sabiranje, već se mogu koristiti i za druge statistike segmenata koje se izračunavaju asocijativnim operacijama (na primer za određivanje najmanjeg ili najvećeg elementa, nzd-a svih elemenata i slično).

Pretpostavimo da je dužina niza stepen broja 2 (ako nije, niz se može dopuniti do najbližeg stepena broja 2, najčešće nulama). Članovi niza predstavljaju listove drveta. Grupišemo dva po dva susedna čvora i na svakom narednom nivou drveta čuvamo roditeljske čvorove koji čuvaju zbirove svoja dva deteta. Ako je dat niz $3, 4, 1, 2, 6, 5, 1, 4$, segmentno drvo za zbirove izgleda ovako.

```
              26
      10              16
  7       3       11      5
3   4   1   2   6   5   1   4
```

Pošto je drvo potpuno, najjednostavnija implementacija je da se čuva implicitno u nizu (slično kao u slučaju hipa). Pretpostavićemo da elemente drveta smeštamo od pozicije 1, jer je tada aritmetika sa indeksima malo jednostavnija (elementi polaznog niza mogu biti indeksirani klasično, krenuvši od nule).

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
-  26 10 16 7  3  11 5  3  4  1  2  6  5  1  4
```

Uočimo nekoliko karakteristika ovog načina smeštanja. Koren je smešten na poziciji 1. Elementi polaznog niza nalaze se na pozicijama $[n, 2n - 1]$. Element koji se u polaznom nizu nalazi na poziciji p, se u segmentnom drvetu nalazi na poziciji $p + n$. Levo dete čvora k nalazi se na poziciji $2k$, a desno na poziciji $2k + 1$. Dakle, na parnim pozicijama se nalaze leva deca svojih roditelja, a na neparnim desna. Roditelj čvora k nalazi se na poziciji $\lfloor \frac{k}{2} \rfloor$.

Formiranje segmentnog drveta na osnovu datog niza je veoma jednostavno. Prvo se elementi polaznog niza prekopiraju u drvo, krenuvši od pozicije n. Zatim se svi unutrašnji čvorovi drveta (od pozicije $n-1$, pa unazad do pozicije 1) popunjavaju kao zbirovi svoje dece (na poziciju k upisujemo zbir elemenata na pozicijama $2k$ i $2k+1$).

```
// na osnovu datog niza a dužine n
// u kom su elementi smešteni od pozicije 0
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije 1
void formirajSegmentnoDrvo(int a[], int n, int drvo[]) {
  // kopiramo originalni niz u listove
  copy_n(a, n, drvo + n);
  // ažuriramo roditelje već upisanih elemenata
  for (int k = n-1; k >= 1; k--)
    drvo[k] = drvo[2*k] + drvo[2*k+1];
}
```

Složenost ove operacije je očigledno linearna u odnosu na dužinu niza n.

Prethodni pristup formira drvo odozdo naviše (prvo se popune listovi, pa onda koren). Još jedan način je da se drvo formira rekurzivno, odozgo naniže. Iako je ova implementacija komplikovanija i malo neefikasnija, pristup odozgo naniže je u nekim kasnijim operacijama neizbežan, pa ga ilustrujemo na ovom jednostavnom primeru. Svaki čvor drveta predstavlja zbir određenog segmenta pozicija. Segment je jednoznačno određen pozicijom k u nizu, ali da bismo olakšali implementaciju granice tog segmenta možemo kroz rekurziju prosleđivati kao parametar funkcije, zajedno sa vrednošću k (neka je to segment $[x, y]$). Drvo krećemo da gradimo od korena gde je $k = 1$ i $[x, y] = [0, n-1]$. Ako roditeljski čvor pokriva segment $[x, y]$, tada levo dete pokriva segment $[x, \lfloor \frac{x+y}{2} \rfloor]$, a desno dete pokriva segment $[\lfloor \frac{x+y}{2} \rfloor + 1, y]$. Drvo popunjavamo rekurzivno, tako što prvo popunimo levo poddrvo, zatim desno i na kraju vrednost u korenu izračunavamo kao zbir vrednosti u levom i desnom detetu. Izlaz iz rekurzije predstavljaju listovi, koje prepoznajemo po tome što pokrivaju segmente dužine 1, i u njih samo kopiramo elemente sa odgovarajućih pozicija polaznog niza.

```
// od elemenata niza a sa pozicija [x, y]
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije k
void formirajSegmentnoDrvo(int a[], int drvo[], int k, int x, int y) {
   if (x == y)
      // u listove prepisujemo elemente polaznog niza
      drvo[k] = a[x];
   else {
      // rekurzivno formiramo levo i desno poddrvo
      int s = (x + y) / 2;
      formirajSegmentnoDrvo(a, drvo, 2*k, x, s);
```

```
    formirajSegmentnoDrvo(a, drvo, 2*k+1, s+1, y);
    // izračunavamo vrednost u korenu
    drvo[k] = drvo[2*k] + drvo[2*k+1];
  }
}

// na osnovu datog niza a dužine n
// u kom su elementi smešteni od pozicije 0
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije 1
void formirajSegmentnoDrvo(int a[], int n, int drvo[]) {
  // krećemo formiranje od korena koji se nalazi u nizu drvo
  // na poziciji 1 i pokriva elemente na pozicijama [0, n-1]
  formirajSegmentnoDrvo(a, drvo, 1, 0, n-1);
}
```

Razmotrimo sada kako bismo našli zbir elemenata na pozicijama iz segmenta $[2, 6]$, tj. zbir elemenata $1, 2, 6, 5, 1$. U segmentnom drvetu taj segment je smešten na pozicijama $[2 + 8, 6 + 8] = [10, 14]$. Zbir prva elementa $(1, 2)$ se nalazi u čvoru iznad njih, zbir naredna dva elemenata $(6, 5)$ takođe, dok se u roditeljskom čvoru elementa 1 nalazi njegov zbir sa elementom 4, koji ne pripada segmentu koji sabiramo. Zato zbir elemenata pozicija $[10, 14]$ možemo razložiti na zbir elemenata na pozicijama $[5, 6]$ i elementa na poziciji 14.

Razmotrimo i kako bismo računali zbir elemenata na pozicijama iz segmenta $[3, 7]$, tj. zbir elemenata $2, 6, 5, 1, 4$. U segmentnom drvetu taj segment je smešten na pozicijama $[3 + 8, 7 + 8] = [11, 15]$. U roditeljskom čvoru elementa 2 nalazi se njegov zbir sa elementom 1 koji ne pripada segmentu koji sabiramo. Zbirovi elemenata 6 i 5 i elemenata 1 i 4 se nalaze u čvorovima iza njih.

Generalno, za sve unutrašnje elemente segmenta smo sigurni da se njihov zbir nalazi u čvorovima iznad njih. Jedini izuzetak mogu da budu elementi na krajevima segmenta. Ako je element na levom kraju segmenta levo dete (što je ekvivalentno tome da se nalazi na parnoj poziciji) tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom desno od njega koji takođe pripada segmentu koji treba sabrati (osim eventualno u slučaju jednočlanog segmenta). U suprotnom (ako se nalazi na neparnoj poziciji), u njegovom roditeljskom čvoru je njegov zbir sa elementom levo od njega, koji ne pripada segmentu koji sabiramo. U toj situaciji, taj element ćemo posebno dodati na zbir i isključiti iz segmenta koji sabiramo pomoću roditeljskih čvorova. Ako je element na desnom kraju segementa levo dete (ako se nalazi na parnoj poziciji), tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom desno od njega, koji ne pripada segmentu koji sabiramo. I u toj situaciji, taj element ćemo posebno dodati na zbir i isključiti iz segmenta koji sabiramo pomoću roditeljskih čvorova. Na kraju, ako se krajnji desni element nalazi u desnom čvoru (ako je na neparnoj poziciji), tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom levo od njega, koji pripada segmentu koji sabiramo (osim eventualno u slučaju

jednočlanog segmenta).

```
// izračunava se zbir elemenata polaznog niza dužine n koji se
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drveta
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
int saberi(int drvo[], int n, int a, int b) {
   a += n; b += n;
   mint zbir = 0;
   while (a <= b) {
      if (a % 2 == 1) zbir += drvo[a++];
      if (b % 2 == 0) zbir += drvo[b--];
      a /= 2;
      b /= 2;
   }
   return zbir;
}
```

Pošto se u svakom koraku dužina segmenta $[a, b]$ polovi, a ona je u početku sigurno manja ili jednaka n, složenost ove operacije je $O(\log n)$.

Prethodna implementacija vrši izračunavanje odozdo naviše. I za ovu operaciju možemo napraviti i rekurzivnu implementaciju koja vrši izračunavanje odozgo naniže. Za svaki čvor u segmentnom drvetu funkcija vraća koliki je doprinos segmenta koji odgovara tom čvoru i njegovim naslednicima traženom zbiru elemenata na pozicijama iz segmenta $[a, b]$. Na počeku krećemo od korena i računamo doprinos celog drveta zbiru elemenata iz segmenta $[a, b]$. Postoje tri različita moguća odnosa između segmenta $[x, y]$ i segmenta $[a, b]$. Ako su disjunktni, doprinos tekućeg čvora zbiru segmenta $[a, b]$ je nula. Ako je $[x, y]$ u potpunosti sadržan u $[a, b]$, tada je doprinos potpun, tj. ceo zbir segmenta $[x, y]$ (a to je broj upisan u nizu na poziciji k) doprinosi zbiru elemenata na pozicijama iz segmenta $[a, b]$. Na kraju, ako se segmenti seku, tada je doprinos tekućeg čvora jednak zbiru doprinosa njegovog levog i desnog deteta. Odatle sledi naredna implementacija.

```
// izračunava se zbir onih elemenata polaznog niza koji se
// nalaze na pozicijama iz segmenta [a, b] koji se nalaze u
// segmentnom drvetu koje čuva elemente polaznog niza koji se
// nalaze na pozicijama iz segmenta [x, y] i smešteno je u nizu
// drvo od pozicije k
int saberi(int drvo[], int k, int x, int y, int a, int b) {
  // segmenti [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // segment [x, y] je potpuno sadržan unutar segmenta [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // segmenti [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return saberi(drvo, 2*k, x, s, a, b) +
```

```
        saberi(drvo, 2*k+1, s+1, y, a, b);
}

// izračunava se zbir elemenata polaznog niza dužine n koji se
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drveta
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
int saberi(int drvo[], int n, int a, int b) {
  // krećemo od drveta smeštenog od pozicije 1 koje
  // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
  return saberi(drvo, 1, 0, n-1, a, b);
}
```

Iako nije sasvim očigledno i ova implementacija će imati složenost $O(\log n)$.

Prilikom ažuriranja nekog elementa potrebno je ažurirati sve čvorove na putanji od tog lista do korena. S obzirom da znamo poziciju roditelja svakog čvora i ova operacija se može veoma jednostavno implementirati.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije 1
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void promeni(int drvo[], int n, int i, int v) {
   // prvo ažuriramo odgovarajući list
   int k = i + n;
   drvo[k] = v;
   // ažuriramo sve roditelje izmenjenih čvorova
   for (k /= 2; k >= 1; k /= 2)
      drvo[k] = drvo[2*k] + drvo[2*k+1];
}
```

Pošto se k polovi u svakom koraku petlje, a kreće od vrednosti najviše $2n - 1$, i složenost ove operacije je $O(\log n)$.

I ovu operaciju možemo implementirati odozgo naniže.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije k
// koje sadrži elemente polaznog niza a dužine n sa pozicija iz
// segmenta [x, y], nakon što se na poziciju i niza upiše vrednost v
void promeni(int drvo[], int k, int x, int y, int i, int v) {
   if (x == y)
     // ažuriramo vrednost u listu
     drvo[k] = v;
   else {
     // proveravamo da li se pozicija i nalazi levo ili desno
     // i u zavisnosti od toga ažuriramo odgovarajuće poddrvo
     int s = (x + y) / 2;
     if (x <= i && i <= s)
       promeni(drvo, 2*k, x, s, i, v);
```

```
        else
          promeni(drvo, 2*k+1, s+1, y, i, v);
        // pošto se promenila vrednost u nekom od dva poddrveta
        // moramo ažurirati vrednost u korenu
        drvo[k] = drvo[2*k] + drvo[2*k+1];
    }
}

// ažurira segmentno drvo smešteno u niz drvo od pozicije 1
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void promeni(int drvo[], int n, int i, int v) {
    // krećemo od drveta smeštenog od pozicije 1 koje
    // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
    promeni(drvo, 1, 0, n-1, i, v);
}
```

Složenost je opet $O(\log n)$, jer se dužina intervala $[x, y]$ u svakom koraku bar dva puta smanjuje.

Umesto funkcije `promeni` često se razmatra funkcija `uvecaj` koja element na poziciji `i` polaznog niza uvećava za datu vrednost `v` i u skladu sa tim ažurira segmentno drvo. Svaka od ove dve funkcije se lako izražava preko one druge.

Implementacija segmentnog drveta za druge asocijativne operacije je skoro identična, osim što se operator `+` menja drugom operacijom.

Fenvikova drveta (BIT)

U nastavku ćemo razmotriti *Fenvikova drveta* tj. *binarno indeksirana drveta* (engl. binary indexed tree, BIT) koja koriste malo manje memorije i mogu biti za konstantni faktor brža od segmentnih drveta (iako je složenost operacija asimptotski jednaka). Sa druge strane, za razliku od segmentnih drveta koja su pogodna za različite operacije, Fenvikova drveta su specijalizovana samo za asocijativne operacije koje imaju inverz (npr. zbirovi ili proizvodi elemenata segmenata se mogu nalaziti uz pomoć BIT, ali ne i minimumi, nzd-ovi i slično). Segmentna drveta mogu da urade sve što i Fenvikova, dok obratno ne važi.

Iako se naziva drvetom, Fenvikovo drvo zapravo predstavlja niz vrednosti zbirova nekih pametno izabranih segmenata. Izbor segmenata je u tesnoj vezi sa binarnom reprezentacijom indeksa. Ponovo ćemo jednostavnosti radi pretpostaviti da se vrednosti u nizu smeštaju od pozicije 1 (vrednost na poziciji 0 je irelevantna) i to i u polaznom nizu i u nizu u kom se smešta drvo. Prilagođavanje koda situaciji u kojoj su u polaznom nizu elementi smešteni od pozicije nula, veoma je jednostavno (samo je na početku svake funkcije koja radi sa drvetom indeks

polaznog niza potrebno uvećati za jedan pre dalje obrade). Ako je polazni niz dužine n, elementi drveta će se smeštati u poseban niz na pozicije $[1, n]$.

Ključna ideja Fenvikovog drveta je sledeća: *u drvetu se na poziciji k čuva zbir vrednosti polaznog niza iz segmenta pozicija oblika $(f(k), k]$ gde je $f(k)$ broj koji se dobije od broja k tako što se iz binarnog zapisa broja k obriše prva jedinica sdesna.*

Na primer, na mestu $k = 21$ zapisuje se zbir elemenata polaznog niza na pozicijama iz intervala $(20, 21]$, jer se broj 21 binarno zapisuje kao 10101 i brisanjem jedinice dobija se binarni zapis 10100 tj. broj 20 (važi da je $f(21) = 20$). Na poziciji broj 20 nalazi se zbir elemenata sa pozicija iz intervala $(16, 20]$, jer se brisanjem jedinice dobija binarni zapis 10000 tj. broj 16 (važi da je $f(20) = 16$). Na poziciji 16 se čuva zbir elemenata sa pozicija iz intervala $(0, 16]$, jer se brisanjem jedinice iz binarnog zapisa broja 16 dobija 0 (važi da je $f(16) = 0$).

Za niz $3, 4, 1, 2, 6, 5, 1, 4$, Fenvikovo drvo bi čuvalo sledeće vrednosti.

```
0     1       2      3      4      5      6      7      8  k
      1      10     11    100    101    110    111   1000  k  binarno
      0       0     10      0    100    100    110      0  f(k) binarno
  (0,1]   (0,2]  (2,3]  (0,4]  (4,5]  (4,6]  (6,7]  (0,8]  interval
      3       4      1      2      6      5      1      4  niz
      3       7      1     10      6     11      1     26  drvo
```

Nadovezivanjem intervala $(0, 16]$, $(16, 20]$ i $(20, 21]$ dobija se interval $(0, 21]$ tj. prefiks niza do pozicije 21. Zbir elemenata u prefiksu se, dakle, može se dobiti kao zbir nekoliko elemenata zapisanih u Fenvikovom drvetu. Ovo, naravno, važi za proizvoljni indeks (ne samo za 21). Broj elemenata čijim se sabiranjem dobija zbir prefiksa je samo $O(\log n)$. Naime, u svakom koraku se broj broj jedinica u binarnom zapisu tekućeg indeksa smanjuje, a broj n se zapisuje sa najviše $O(\log n)$ binarnih jedinica.

Implementacija je veoma jednostavna, kada se pronađe način da se iz binarnog zapisa broja ukloni prva jedinica sleva tj. da se za dati broj k izračuna $f(k)$. Pod pretpostavkom da su brojevi zapisani u potpunom komplementu, izrazom `k & -k` može se dobiti broj koji sadrži samo jednu jedinicu i to na mestu poslednje jedinice u zapisu broja `k`. Oduzimanjem te vrednosti od broja `k` tj. izrazom `k - (k & -k)` dobijamo efekat brisanja poslednje jedince u binarnom zapisu broja `k` i to predstavlja implementaciju funkcije f. Drugi način da se to uradi je da se izračuna vrednost `k & (k-1)`.

Zbir prefiksa $[0, k]$ polaznog niza možemo onda izračunati narednom funkcijom.

```
// na osnovu Fenvikovog drveta smeštenog u niz drvo
// izračunava zbir prefiksa (0, k] polaznog niza
int zbirPrefiksa(int drvo[], int k) {
  int zbir = 0;
  while (k > 0) {
     zbir += drvo[k];
```

```
    k -= k & -k;
  }
}
```

Kada znamo zbir prefiksa, zbir proizvoljnog segmenta $[a, b]$ možemo izračunati kao razliku zbira prefiksa $(0, b]$ i zbira prefiksa $(0, a-1]$. Pošto se oba računaju u vremenu $O(\log n)$, i zbir svakog segmenta možemo izračunati u vremenu $O(\log n)$.

Osnovna prednost Fenvikovih drveta u odnosu na niz svih zbirova prefiksa je to što se mogu efikasno ažurirati. Razmotrimo funkciju koja ažurira drvo nakon uvećanja elementa u polaznom nizu na poziciji k za vrednost x. Tada je za x potrebno uvećati sve one zbirove u drvetu u kojima se kao sabirak javlja i element na poziciji k. Ti brojevi se izračunavaju veoma slično kao u prethodnoj funkciji, jedino što se umesto oduzimanja vrednosti `k & -k` broj `k` u svakom koraku uvećava za `k & -k`.

```
// Ažurira Fenvikovo drvo smešteno u niz drvo nakon što se
// u originalnom nizu element na poziciji k uveća za x
void uvećaj(int drvo[], int n, int k, int x) {
  while (k <= n) {
    drvo[k] += x;
    k += k & -k;
  }
}
```

Objasnimo i dokažimo korektnost prethodne implementacije. Potrebno je ažurirati sve one pozicije m čiji pridruženi segment sadrži vrednost k, tj. sve one pozicije m takve da je $k \in (f(m), m]$, tj. $f(m) < k \leq m$. Ovo nikako ne može da važi za brojeve $m < k$, a sigurno važi za broj $m = k$, jer je $f(k) < k$, kada je $k > 0$ (a mi pretpostavljamo da je $1 \leq k \leq n$). Za brojeve $m > k$, sigurno važi desna nejednakost i potrebno je utvrditi da važi leva. Neka je $g(k)$ broj koji se dobija od k tako što se k sabere sa brojem koji ima samo jednu jedinicu u svom binarnom zapisu i to na poziciji na kojoj se nalazi poslednja jedinica u binarnom zapisu broja k. Na primer, za broj $k = 101100$, broj $g(k) = 101100 + 100 = 110000$. U implementaciji se broj $g(k)$ lako može izračunati kao `k + (k & -k)`. Tvrdimo da je najmanji broj m koji zadovoljava uslov $f(m) < k < m$ upravo $g(k)$. Zaista, očigledno važi $k < g(k)$ i $g(k)$ ima sve nule od pozicije poslednje jedinice u binarnom zapisu broja k (uključujući i nju), pa do kraja, pa se brisanjem njegove poslednje jedinice tj. izračunavanjem $f(g(k))$ sigurno dobija broj koji je strogo manji od k. Nijedan broj m između k i $g(k)$ ne može da zadovolji uslov da je $f(m) < k$. Naime, svi ti brojevi se poklapaju sa brojem k na svim pozicijama pre krajnjih nula, a na pozicijama krajnjih nula broja k imaju bar neku jedinicu, čijim se brisanjem dobija broj koji je veći ili jednak k. Po istom principu zaključujemo da naredni traženi broj mora biti $g(g(k))$, zatim $g(g(g(k)))$ itd. sve dok se ne dobije neki broj koji prevazilazi n. Zaista, važi da je $k < g(k) < g(g(k))$. Važi da je $f(g(g(k))) < f(g(k)) < k$, pa $g(g(k))$ zadovoljava uslov. Nijedan broj između $g(k)$ i $g(g(k))$ ne može da zadovolji uslov, jer se svi oni poklapaju sa $g(k)$ u svim binarnim ciframa, osim na njegovim krajnjim nulama gde imaju

neke jedinice. Brisanjem poslednje jedinice se dobija broj koji je veći ili jednak $g(k)$, pa dobijeni broj ne može biti manji od k. Otuda sledi da su jedine pozicije koje treba ažurirati upravo pozicije iz serije k, $g(k)$, $g(g(k))$ itd., sve dok su one manje ili jednak n, pa je naša implementacija korektna.

Ostaje još pitanje kako u startu formirati Fenvikovo drvo, međutim, formiranje se može svesti na to da se kreira drvo popunjeno samo nulama, a da se zatim uvećava vrednost jednog po jednog elementa niza prethodnom funkcijom.

```
// Na osnovu niza a u kom su elementi smešteni
// na pozicijama iz segmenta [1, n] formira Fenvikovo drvo
// i smešta ga u niz drvo (na pozicije iz segmenta [1, n])
void formirajDrvo(int drvo[], int n, int a[]) {
  fill_n(a+1, n, 0);
  for (int k = 1; k <= n; k++)
     dodaj(drvo, n, k, a[k]);
}
```

Broj inverzija pomoću Fenvikovog drveta

Prikažimo sada upotrebu Fenvikovog drveta u rešavanju jednog algoritamskog problema. Umesto Fenvikovog, moglo je biti upotrebljeno i segmentno drvo.

Problem: Odredi koliko različitih parova elemenata u nizu je takvo da je prvi element strogo veći drugog. Na primer, u nizu $5, 4, 3, 1, 2$ takvi su parovi $(5, 4)$, $(5, 3)$, $(5, 1)$, $(5, 2)$, $(4, 3)$, $(4, 1)$, $(4, 2)$, $(3, 1)$ i $(3, 2)$ i ima ih 9.

Rešenje grubom silom podrazumeva proveru svakog para elemenata i očigledno je složenosti $O(n^2)$. Možemo i efikasnije od toga.

Osnovna ideja je da radimo induktivno i da obrađujemo niz element po element. Pretpostavimo da smo ojačali induktivnu hipotezu i da za obrađeni prefiks niza znamo ne samo broj inverznih parova u tom prefiksu, već i frekvencije svih elemenata (ako su svi elementi u nizu različiti frekvencije će biti ili 0 ili 1). Da bismo odredili koliko inverzija tekući element pravi sa elementima ispred sebe, potrebno je da znamo koliko postoji elemenata koji su se javili pre tekućeg, a koji su strogo veći od njega. To možemo otkriti tako što saberemo frekvencije svih elemenata koji su strogo veći od tekućeg (ako je tekući element a_i, zanima nas zbir elemenata niza frekvencija koji se javljaju na pozicijama iz intervala $[a_i, max]$, gde je max najveći element niza. To možemo saznati jednostavno iz Fenvikovog drveta kao razliku zbirova dva prefiksa.

Primećujemo da su segmenti čiji nas zbirovi zanimaju stalno sufiksi niza frekvencija. Ovo nas može asocirati na to da bismo obilaskom u suprotnom smeru mogli malo popraviti efikasnost, jer bismo umesto zbirova sufiksa koji zahtevaju dva obilaska izračunavali zbirove prefiksa koji zahtevaju samo jedan obilazak. Zaista, ako niz obrađujemo unatrag i računamo frekvencije svih ranije viđenih elemenata, onda inverzije u kojima učestvuje tekući element i oni elementi iza

njega određujemo tako što prebrojimo koliko ima viđenih elemenata koji su strogo manji od tekućeg, a to možemo saznati ako saberemo frekvencije koje se nalaze na pozicijama iz intervala $[1, a-1]$. Na kraju ne smemo zaboraviti da održimo induktivnu hipotezu tako što nakon obrade uvećamo frekvenciju pojavljivanja tekućeg elementa.

Prethodni pristup nije moguć ako se u nizu javljaju negativni elementi, a prilično je memorijski neefikasan ako je maksimalni element niza veliki (memorijska složenost BIT-a je $O(max)$). Nama nisu relevantne same vrednosti, već samo njihov međusobni odnos. Zato pre obrade pomoću BIT-a možemo uraditi takozvanu kompresiju indeksa (engl. index compression). Svaki element u nizu ćemo zameniti njegovom pozicijom u sortiranom redosledu (krenuvši od 1). Iste elemente možemo zameniti istim pozicijama. Najlakši način da se to postigne je da se napravi sortirana kopija niza, i zatim da za se svaki element polaznog niza binarnom pretragom pronađe pozicija njegovog prvog pojavljivanja u sortiranoj kopiji (u jeziku C++ to možemo jednostavno postići bibliotečkim funkcijama `sort` i `lower_bound`).

```
// operacije za rad sa Fenvikovim drvetom

void dodaj(vector<int>& drvo, int k, int v) {
  while (k < drvo.size()) {
    drvo[k] += v;
    k += k & -k;
  }
}

int zbirPrefiksa(const vector<int>& drvo, int k) {
  int zbir = 0;
  while (k > 0) {
    zbir += drvo[k];
    k -= k & -k;
  }
  return zbir;
}

// izračunava broj parova i < j takvih da je a[i] > a[j]
int brojInverzija(const vector<int>& a) {
  // sortiramo niz
  int n = a.size();
  vector<int> a_sort = a;
  sort(begin(a_sort), end(a_sort));
  // Fenvikovo drvo
  vector<int> drvo(n+1, 0);
  // broj inverzija
  int broj = 0;
  for (int i = n-1; i >= 0; i--) {
```

```
    // na osnovu sortiranog niza a,
    // određujemo koji je po veličini u nizu a element a[i] -
    // broji se od 1
    auto it = lower_bound(begin(a_sort), end(a_sort), a[i]);
    int x = distance(begin(a_sort), it) + 1;
    // uvećavamo broj inverzija za broj do sada viđenih elemenata
    // koji su strogo manji od x
    broj += zbirPrefiksa(drvo, x-1);
    // ažuriramo frekvenciju pojavljivanja elementa x
    dodaj(drvo, x, 1);
  }
  // vraćamo ukupan broj inverzija
  return broj;
}
```

Sortiranje nosi složenost $O(n \log n)$. Nakon toga se za svaki od n elemenata izvršava jedna binarna pretraga čija je složenost $O(\log n)$, jedno izračunavanje zbira prefiksa Fenvikovog drveta čija je složenost takođe $O(\log n)$ i na kraju jedno ažuriranje vrednosti elementa u originalnom nizu i Fenvikovom drvetu čija je složenost takođe $O(\log n)$, tako da je ukupna složenost $O(n \log n)$.

Rešenje zasnovano na principu dekompozicije (prilagođenoj varijanti sortiranja objedinjavanjem) iste složenosti prikazaćemo kasnije.

Ažuriranje celih raspona niza odjednom

Videli smo da i segmentna i Fenvikova drveta podržavaju efikasno izračunavanje statistika određenih segmenata (raspona) niza i ažuriranje pojedinačnih elemenata niza. Ažuriranje celih segmenata niza odjednom nije direktno podržano. Ako se ono svede na pojedinačno ažuriranje svih elemenata unutar segmenta, dobija se loša složenost.

Moguće je jednostavno upotrebiti Fenvikovo drvo tako da se efikasno podrži uvećavanje svih elemenata iz datog segmenta odjednom, ali onda se gubi mogućnost efikasnog izračunavanja zbirova elemenata segmenata, već je samo moguće efikasno vraćati vrednosti pojedinačnih elemenata niza. Osnovna ideja je da se održava niz razlika susednih elemenata polaznog niza i da se taj niz razlika čuva u Fenvikovom drvetu. Uvećavanje svih elemenata segmenata polaznog niza za neku vrednost x, svodi se na promenu dva elementa niza razlika, dok se rekonstrukcija elementa polaznog niza na osnovu niza razlika svodi na izračunavanje zbira odgovarajućeg prefiksa, što se pomoću Fenvikovog drveta može uraditi veoma efikasno. Na ovaj način i ažuriranje celih segmenata niza odjednom i očitavanje pojedinačnih elemenata možemo postići u složenosti $O(\log n)$, što nije bilo moguće samo uz korišćenje niza razlika (uvećavanja svih elemenata nekog segmenta je tada bilo složenosti $O(1)$, ali je očitavanje vrednosti iz niza bilo složenosto $O(n)$).

Efikasno uvećanje svih elemenata u datom segmentu za istu vrednost i izračunavanje zbirova segmenata moguće je implementirati pomoću održavanja dva Fenvikova drveta (čime se nećemo baviti).

Ove operacije se mogu implementirati nad segmentnim drvetima ako se primeni takozvana tehnika *lenje propagacije* (engl. lazy propagation). Za lenju propagaciju nam je bitno da znamo rad sa segmentnim drvetom odozgo naniže.

Svaki čvor u segmentnom drvetu se odnosi na određeni segment elemenata polaznog niza i čuva zbir tog segmenta. Ako se taj segment u celosti sadrži unutar segmenta koji se ažurira, možemo unapred izračunati za koliko se povećava vrednost u korenu. Naime, pošto se svaka vrednost u segmentu povećava za v, tada se vrednost zbira tog segmenta povećava za $k \cdot v$, gde je k broj elemenata u tom segmentu. Vrednost zbira u korenu time biva ažurirana u konstantnom vremenu, ali vrednosti zbirova unutar poddrveta kojima je to koren (uključujući i vrednosti u listovima koje odgovaraju vrednostima polaznog niza) i dalje ostaju neažurne. Njihovo ažuriranje zahtevalo bi linearno vreme, što nam je nedopustivo. Ključna ideja je da se ažuriranje tih vrednosti odloži i da se one ne ažuriraju odmah, već samo tokom neke kasnije posete tim čvorovima, do koje bi došlo i inače (ne želimo da te čvorove posećujemo samo zbog ovog ažuriranja, već ćemo ažuriranje uraditi usput, tokom neke druge posete tim čvorovima koja bi se svakako morala desiti). Postavlja se pitanje kako da signaliziramo da vrednosti zbirova u nekom poddrvetu nisu ažurne i dodatno ostavimo uputstvo na koji način se mogu ažurirati. U tom cilju proširujemo čvorove i u svakom od njih pored vrednosti zbira segmenta čuvamo i dodatni *koeficijente lenje propagacije*. Ako drvo u svom korenu ima koeficijent lenje propagacije c koji je različit od nule, to znači da vrednosti zbirova u celom tom drvetu nisu ažurne i da je svaki od listova tog drveta potrebno povećati za c i u odnosu na to ažurirati i vrednosti zbirova u svim unutrašnjim čvorovima tog drveta (uključujući i koren). Ažuriranje se može odlagati sve dok vrednost zbira u nekom čvoru ne postane zaista neophodna, a to je tek prilikom upita izračunavanja vrednosti zbira nekog segmenta. Ipak, vrednosti zbirova u čvorovima ćemo ažurirati i češće i to zapravo prilikom svake posete čvoru - bilo u sklopu operacije uvećanja vrednosti iz nekog segmenta pozicija polaznog niza, bilo u sklopu upita izračunavanja zbira nekog segmenta. Na početku obe rekurzivne funkcije ćemo proveravati da li je vrednost koeficijenta lenje propagacije različita od nule i ako jeste, ažuriraćemo vrednost zbira tako što ćemo ga uvećati za proizvod tog koeficijenta i broja elemenata u segmentu koji taj čvor predstavlja, a zatim koeficijente lenje propagacije oba njegova deteta uvećati za taj koeficijent (time koren drveta koji trenutno posećujemo postaje ažuran, a njegovim poddrvetima se daje uputstvo kako ih u budućnosti ažurirati). Primetimo da se izbegava ažuriranje celog drveta odjednom, već se ažurira samo koren, što je operacija složenosti $O(1)$.

Imajući ovo u vidu, razmotrimo kako se može implementirati funkcija koja vrši uvećanje svih elemenata nekog segmenata. Njena invarijanta će biti da su svi čvorovi u drvetu koje ili sadržati ažurne vrednosti zbirova ili će biti ispravno obeleženi za kasnija ažuriranja (preko koeficijenta lenje propagacije), a da će

nakon njenog izvršavanja koren drveta na kom je pozvana sadržati aktuelnu vrednost zbira. Nakon početnog obezbeđivanja da vrednost u tekućem čvoru postane ažurna, moguća su tri sledeća slučaja. Ako je segment u tekućem čvoru disjunktan u odnosu na segment koji se ažurira, tada su svi čvorovi u poddrvetu kojem je on koren već ili ažurni ili ispravno obeleženi za kasnije ažuriranje i nije potrebno ništa uraditi. Ako je segment koji odgovara tekućem čvoru potpuno sadržan u segmentu čiji se elementi uvećavaju, tada se njegova vrednost ažurira (uvećavanjem za $k \cdot v$, gde je k broj elemenata segmenta koji odgovara tekućem čvoru, a v vrednost uvećanja), a njegovoj deci se koeficijent lenje propagacije uvećava za v. Na kraju, ako se dva segmenta seku, tada se prelazi na rekurzivnu obradu dva deteta. Nakon izvršavanja funkcije nad njima, sigurni smo da će svi čvorovi u levom i desnom poddrvetu zadovoljavati uslov invarijante i da će oba korena imati ažurne vrednosti. Ažurnu vrednost u korenu postizaćemo sabiranjem vrednosti dva deteta.

```
// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [x, y]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void promeni(int drvo[], int lenjo[], int k, int x, int y,
             int a, int b, int v) {
   // ažuriramo vrednost u korenu, ako nije ažurna
   if (lenjo[k] != 0) {
      drvo[k] += (y - x + 1) * lenjo[k];
      // ako nije u pitanju list propagaciju prenosimo na decu
      if (x != y) {
         lenjo[2*k] += lenjo[k];
         lenjo[2*k+1] += lenjo[k];
      }
      lenjo[k] = 0;
   }
   // ako su intervali disjunktni, ništa nije potrebno raditi
   if (b < x || y < a) return;
   // ako je interval [x, y] ceo sadržan u intervalu [a, b]
   if (a <= x && y <= b) {
     drvo[k] += (y - x + 1) * v;
     // ako nije u pitanju list propagaciju prenosimo na decu
     if (x != y) {
        lenjo[2*k] += v;
        lenjo[2*k+1] += v;
     }
   } else {
     // u suprotnom se intervali seku,
     // pa rekurzivno obilazimo poddrveta
     int s = (x + y) / 2;
```

```
    promeni(drvo, lenjo, 2*k, x, s, a, b, v);
    promeni(drvo, lenjo, 2*k+1, x, s, a, b, v);
    drvo[k] = drvo[2*k] + drvo[2*k+1];
  }
}

// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [0, n-1]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void promeni(int drvo[], int lenjo[], int n,
             int a, int b, int v) {
  promeni(drvo, lenjo, 1, 0, n-1, a, b, v);
}
```

Prikažimo rad ove funkcije na jednom primeru.

```
                      26/0
          10/0                    16/0
    7/0         3/0         11/0         5/0
3/0    4/0   1/0   2/0   6/0    5/0   1/0   4/0

0      1     2     3     4      5     6     7
```

Prikažimo kako bismo sve elemente iz segmenta pozicija $[2, 7]$ uvećali za 3. Segmenti $[0, 7]$ i $[2, 7]$ se seku, pa stoga ažuriranje prepuštamo naslednicima i nakon njihovog ažuriranja, pri povratku iz rekurzije vrednost određujemo kao zbir njihovih ažuriranih vrednosti. Na levoj strani se segment $[0, 3]$ seče sa $[2, 7]$ pa i on prepušta ažuriranje naslednicama i ažurira se tek pri povratku iz rekurzije. Segment $[0, 1]$ je disjunktan u odnosu na $[2, 7]$ i tu onda nije potrebno ništa raditi. Segment $[2, 3]$ je ceo sadržan u $[2, 7]$, i kod njega direktno možemo da znamo kako se zbir uvećava. Pošto imamo dva elementa i svaki se uvećava za 3, zbir se uvećava ukupno za $2 \cdot 3 = 6$ i postavlja se na 9. U ovom trenutku izbegavamo ažuriranje svih vrednosti u drvetu u kom je taj element koren, već samo naslednicima upisujemo da je potrebno propagirati uvećanje za 3, ali samu propagaciju odlažemo za trenutak kada ona postane neophodna. U povratku iz rekurzije, vrednost 10 uvećavamo na $7 + 9 = 16$. Što se tiče desnog poddrveta, segment $[4, 7]$ je ceo sadržan u segmentu $[2, 7]$, pa i tu možemo izračunati vrednost zbira. Pošto se 4 elementa uvećavaju za po 3, ukupan zbir se uvećava za $4 \cdot 3 = 12$. Zato se vrednost 16 menja u 28. Propagaciju ažuriranja kroz poddrvo odlažemo i samo njegovoj deci beležimo da je uvećanje za 3 potrebno izvršiti u nekom kasnijem trenutku. Pri povratku iz rekurzije vrednost u korenu ažuriramo sa 26 na $16 + 28 = 44$. Nakon toga dobija se sledeće drvo.

```
                      44/0
          16/0                    28/0
    7/0         9/0         11/3         5/3
```

```
3/0     4/0    1/3    2/3    6/0     5/0    1/0    4/0

0       1      2      3      4       5      6      7
```

Pretpostavimo da se sada elementi iz segmenta $[0, 5]$ uvećavaju za 2. Ponovo se kreće od vrha i kada se ustanovi da se segment $[0, 7]$ seče sa $[0, 5]$ ažuriranje se prepušta deci i vrednost se ažurira tek pri povratku iz rekurzije. Segment $[0, 3]$ je ceo sadržan u $[0, 5]$, pa se zato vrednost 16 uvećava za $4 \cdot 2 = 8$ i postavlja na 24. Poddrveta se ne ažurira odmah, već se samo njihovim korenima upisuje da je sve vrednosti potrebno ažurirati za 2. U desnom poddrvetu segment $[4, 7]$ se seče sa $[0, 5]$, pa se rekurzivno obrađuju poddrveta. Pri obradi čvora 11, primećuje se da je on trebao da bude ažuriran, međutim, još nije, pa se onda njegova vrednost ažurira i uvećava za $2 \cdot 3$ i sa 11 menja na 17. Njegovi naslednici se ne ažuriraju odmah, već samo ako to bude potrebno i njima se samo upisuje lenja vrednost 3. Tek nakon toga, se primećuje da se segment $[4, 5]$ ceo sadrži u $[0, 5]$, pa se onda vrednost 17 uvećava za $2 \cdot 2 = 4$ i postavlja na 21. Poddrveta se ne ažuriraju odmah, već samo po potrebi tako što se u njihovim korenima postavi vrednost lenjog koeficijenta. Pošto je u njima već upisana vrednost 3, ona se sada uvećava za 2 i postavlja na 5. Prelazi se tada na obradu poddrveta u čijem je korenu vrednost 5 i pošto ono nije ažurno, prvo se ta vrednost 5 uvećava za $2 \cdot 3 = 6$ i postavlja na 11, a njegovoj deci se lenji koeficijent postavlja na 3. Nakon toga se primećuje da je segment $[6, 7]$ disjuktan sa $[0, 5]$ i ne radi se ništa. U povratku kroz rekurziju se ažuriraju vrednosti roditeljskih čvorova i dolazi se do narednog drveta.

```
                      56/0
          24/0                    32/0
    7/2          9/2          21/0          11/0
3/0     4/0    1/3    2/3    6/5     5/5    1/3    4/3

0       1      2      3      4       5      6      7
```

Funkcija izračunavanja vrednosti zbira segmenta ostaje praktično nepromenjena, osim što se pri ulasku u svaki čvor vrši njegovo ažuriranje.

```
// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [x, y]
// izračunava se zbir elemenata polaznog niza sa pozicija
// iz segmenta [a, b]
int saberi(int drvo[], int lenjo[], int k, int x, int y,
           int a, int b) {
  // ažuriramo vrednost u korenu, ako nije ažurna
  if (lenjo[k] != 0) {
     drvo[k] += (y - x + 1) * lenjo[k];
     if (x != y) {
       lenjo[2*k] += lenjo[k];
       lenjo[2*k+1] += lenjo[k];
```

```
    }
    lenjo[k] = 0;
  }

  // intervali [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // interval [x, y] je potpuno sadržan unutar intervala [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // intervali [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return saberi(drvo, lenjo, 2*k, x, s, a, b) +
         saberi(drvo, lenjo, 2*k+1, s+1, y, a, b);
}

// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [0, n-1]
// izračunava se zbir elemenata polaznog niza sa pozicija
// iz segmenta [a, b]
int saberi(int drvo[], int lenjo[], int n, int a, int b) {
  // računamo doprinos celog niza,
  // tj. elemenata iz intervala [0, n-1]
  return saberi(drvo, lenjo, 1, 0, n-1, a, b);
}
```

Prikažimo sada rad prethodne funkcije na tekućem primeru. Razmotrimo kako se sada izračunava zbir elemenata u segmentu $[3, 5]$. Krećemo od vrha. Segment $[0, 7]$ se seče sa $[3, 5]$, pa se rekurzivno obrađuju deca. U levom poddrvetu segment $[0, 3]$ takođe ima presek sa $[3, 5]$ pa prelazimo na naredni nivo rekurzije. Prilikom posete čvora u čijem je korenu vrednost 7 primećuje se da njegova vrednost nije ažurna, pa se koristi prilika da se ona ažurira, tako što se uveća za $2 \cdot 2 = 4$, a naslednicima se lenji koeficijent uveća za 2. Pošto je segment $[0, 1]$ disjunktan sa $[3, 5]$, vraća se vrednost 0. Prilikom posete čvora u čijem je korenu vrednost 9 primećuje se da njegova vrednost nije ažurna, pa se koristi prilika da se ona ažurira, tako što se uveća za $2 \cdot 2 = 4$, a naslednicima se lenji koeficijent uveća za 2. Segment $[2, 3]$ se seče sa $[3, 5]$, pa se rekurzivno vrši obrada poddrveta. Vrednost 1 se prvo ažurira tako što se poveća za $1 \cdot 5 = 5$, a onda, pošto je $[2, 2]$ disjunktno sa $[3, 5]$ vraća se vrednost 0. Vrednost 2 se takođe prvo ažurira tako što se poveća za $1 \cdot 5 = 5$, a pošto je segment $[3, 3]$ potpuno sadržan u $[3, 5]$ vraća se vrednost 7. U desnom poddrvetu je čvor sa vrednošću 32 ažuran, segment $[4, 7]$ se seče sa $[3, 5]$, pa se prelazi na obradu naslednika. Čvor sa vrednošću 21 je ažuran, segment $[4, 5]$ je ceo sadržan u $[3, 5]$, pa se vraća vrednost 21. Čvor sa vrednošću 11 je takođe ažuran, ali je segment $[6, 7]$ disjunktan u odnosu na $[3, 5]$, pa se vraća vrednost 0. Dakle, čvorovi 13 i 24 vraćaju vrednost 7, čvor 32 vraća vrednost 21, pa čvor 56 vraća vrednost 28.

Stanje drveta nakon izvršavanja upita je sledeće.

```
                      56/0
          24/0                         32/0
   11/0         13/0         21/0         11/0
3/2    4/2   6/0   7/0   6/5    5/5   1/3    4/3

0      1     2     3     4      5     6      7
```