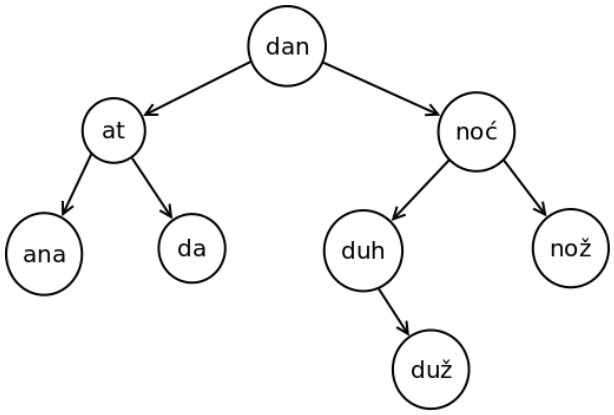
Napredne strukture podataka

Prefiksno drvo

Uređena binarna drveta omogućavaju efikasnu implementaciju struktura podataka sa asocijativnim pristupom[1] kod kojih se pristup elementima vrši po ključu koji nije celobrojna vrednost, već niska ili nešto drugo. Na slici 1 prikazano je uređeno binarno drvo koje sadrži niske *ana*, *at*, *noć*, *nož*, *da*, *dan*, *duh* i *duž* kao ključeve.



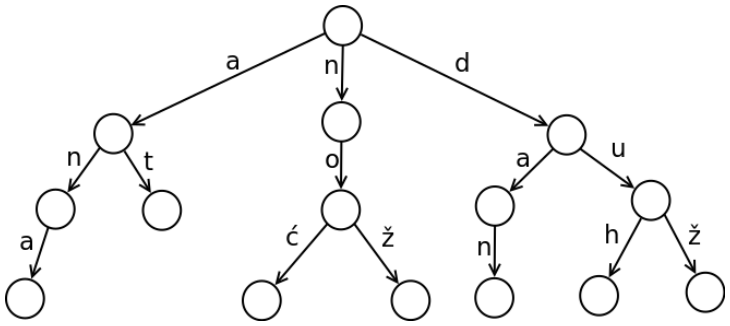
Slika 1: Uređeno binarno stablo čiji su ključevi niske.

Još jedna struktura podataka u vidu drveta koja omogućava efikasan asocijativni pristup je *prefiksno drvo* takođe poznato pod engleskim nazivom *trie*[2] (od engleske reči *reTRIEeval*). Osnovna ideja ove strukture podataka je da putanje od korena do listova ili do nekih od unutrašnjih čvorova drveta kodiraju ključeve, a da se podaci vezani za taj ključ čuvaju u čvoru do kojeg se dolazi pronalaženjem ključa duž putanje. 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 prefiksnog drveta, kod kojeg su prikazane oznake pridružene granama, a ne čvorovima, dat je na slici 2. Ključevi koje ovo prefiksno drvo čuva su *ana*, *at*, *noć*, *nož*, *da*, *dan*, *duh* i *duž*. Primetimo da se ključ *da* ne završava u listu i da stoga svaki čvor mora čuvati informaciju o tome da li se

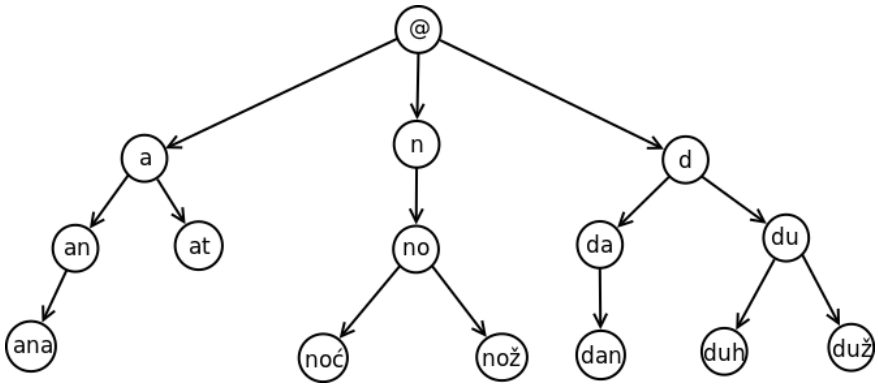
[1] Kod asocijativnih struktura podataka pristup elementima se vrši na osnovu vrednosti ključa, a ne na osnovu indeksa, odnosno pozicije elementa u strukturi podataka.

[2]



Slika 2: Primer prefiksnog drveta.

njime kompletira neki ključ (i u tom slučaju sadržati odgovarajući podatak) ili ne. Ilustracije radi na slici 3 na čvorovima prefiksnog drveta prikazane su oznake akumulirane do tih čvorova. Treba imati u vidu da ovaj prikaz ne ilustruje implementaciju, već samo prefikse duž grane. Simbol @ predstavlja praznu reč.

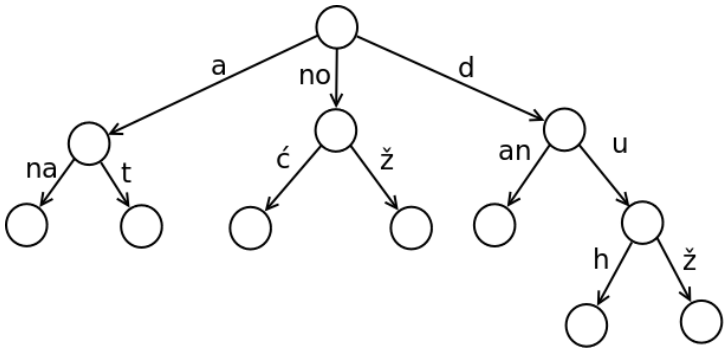


Slika 3: Primer prefiksnog drveta sa akumuliranim prefiksima na čvorovima.

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. Ovako se dobija kompaktnija reprezentacija prefiksnog drveta kod koje svaki unutrašnji čvor ima bar dva deteta (slika 4).

Pored opšteg asocijativnog pristupa podacima, očigledna primena ove strukture podataka 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 podataka nije rezervisana za čuvanje niski. Na primer, u slučaju celih brojeva ključevi se mogu sastojati od njihovih dekadnih cifara ili pak od bitova njihovih binarnih reprezentacija. Pored niski, najčešće se



Slika 4: Primer prefiksnog drveta sa granama označenim niskama.

u prefiksnom drvetu čuvaju binarne reprezentacije brojeva fiksne širine.

Operacije pretrage i umetanja elemenata u prefiksno drvo se pravolinijski implementiraju, dok je u slučaju brisanja nekada potrebno brisati više od jednog čvora.

U nastavku su date implementacije osnovnih operacija sa prefiksnim drvetom na primeru formiranja i pretrage rečnika. U ovom slučaju jedino je potrebno podržati dodavanje ključeva u strukturu podataka i uz ključeve nema potrebe čuvati neke pridružene vrednosti. Čvorovi prefiksnog drveta mogu imati različit broj dece, ali maksimalni broj dece određen je veličinom azbuke koja se koristi za kodiranje ključeva. Čvor prefiksnog drveta možemo implementirati korišćenjem niza pokazivača koji sadrži po jedan element za svaku moguću vrednost karaktera kojima se kodiraju ključevi (npr. možemo koristiti niz pokazivača dužine 26 ako se ključevi kodiraju kao niske engleskog alfabeta). Međutim, efikasnije je u svakom čvoru čuvati informacije samo o onim karakterima za koje postoji grana iz tog čvora, odnosno iskoristiti mape.

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

// osnovna struktura čvora prefiksnog drveta - u svakom čvoru čuvamo mapu
// grana obeleženih karakterima ka potomcima i informaciju da li
// je u ovom čvoru kraj neke reči
struct cvor {
  bool krajKljuca = false;
  unordered_map<char, cvor*> grane;
};

// tražimo sufiks reči w koji počinje od pozicije i
// u drvetu na čiji koren ukazuje pokazivač drvo
```

```
bool nadji(cvor *drvo, 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 drvo->krajKljuca;

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako je nađemo, rekurzivno tražimo ostatak sufiksa od pozicije i+1
  if(it != drvo->grane.end())
    return nadji(it->second, w, i+1);

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

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

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

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako takva grana ne postoji, dodajemo je kreirajući novi čvor
  if(it == drvo->grane.end())
    drvo->grane[w[i]] = new cvor();

  // sada znamo da grana sa w[i] sigurno postoji i preko te grane
  // nastavljamo dodavanje sufiksa koji počinje na i+1;
  umetni(drvo->grane[w[i]], w, i+1);
}

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

// program kojim testiramo gornje funkcije
int main() {
  cvor* drvo = new cvor();
  vector<string> reci
```

```
    {"ana", "at", "noc", "noz", "da", "dan", "duh", "duz"};
  vector<string> reci_neg
    {"", "a", "d", "ananas", "marko", "ptica"};
  for(auto w : reci)
    umetni(drvo, w);
  for(auto w : reci)
    cout << w << ": " << (nadji(drvo, w) ? "da" : "ne") << endl;
  for(auto w : reci_neg)
    cout << w << ": " << (nadji(drvo, w) ? "da" : "ne") << endl;
  return 0;
}
```

U slučaju konačne azbuke veličine m i kada se u svakom čvoru čuva neuređena mapa, složenost operacija prefiksnog drveta u najgorem slučaju je $O(mn)$, gde je n dužina reči koja se traži, umeće ili briše, dok je amortizovana složenost ovih operacija $O(n)$. Ukoliko su ključevi koji se čuvaju 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. Štaviše prostorna složenost prefiksnog drveta u najgorem slučaju iznosi $O(M \cdot N \cdot m)$, gde je sa N označen broj ključeva koji se čuvaju u prefiksnom drvetu, a sa M maksimalna dužina ključa. Naime, maksimalni broj čvorova prefiksnog drveta jednak je $O(M \cdot N)$ i dešavao bi se u slučaju kada ne bi bilo nikakvog preklapanja ključeva, dok je prostorna složenost svakog čvora jednaka $O(m)$ zbog potrebe čuvanja mape u svakom čvoru. Primetimo da je očekivana prostorna složenost manja jer će se u slučaju realne konačne azbuke, npr. engleske abecede prvo preklapanje javiti već nakon 26 ključeva.

Kada bi se umesto prefiksnog drveta koristilo balansirano uređeno binarno drvo koje bi čuvalo kompletne ključeve u čvorovima (slika 1), vremenska složenost ovih operacija bi u najgorem slučaju iznosila $O(M \log N)$, gde je sa N označen ukupan broj ključeva koji se čuvaju u drvetu, a sa M maksimalna dužina ključa. Prostorna složenost ovog pristupa bila bi $O(M \cdot N)$.

Disjunktni podskupovi (union-find)

Ponekad je u programu potrebno održavati nekoliko disjunktnih podskupova određenog skupa, pri čemu je potrebno moći za dati element efikasno pronaći kom podskupu 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 kom pripada i proverimo da li su one jednake.

Jedna moguća implementacija ovako osmišljene strukture podataka podrazumeva da se održava preslikavanje svakog elementa u oznaku podskupa kojem pripada. Ako pretpostavimo da razmatramo skup od n elemenata i 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 (ukoliko elementi nisu numerisani brojevima, mogli bismo umesto niza da koristimo mapu). 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 $O(n)$. Pritom argumenti operacije `union` ne moraju biti oznake podskupova čiju uniju treba kreirati, već se kao argumenti mogu proslediti proizvoljni elementi tih podskupova.

Pretpostavljaćemo da je početno stanje takvo da svaki element pripada zasebnom podskupu. Razmotrimo kako bi se menjao sadržaj odgovarajućeg niza nakon izvršavanja operacija unija.

```
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7

unija(3,4)

0 1 2 3 4 5 6 7
0 1 2 4 4 5 6 7

unija(1,5)

0 1 2 3 4 5 6 7
0 5 2 4 4 5 6 7

unija(1,2)

0 1 2 3 4 5 6 7
0 5 5 4 4 5 6 7

unija(2,3)

0 1 2 3 4 5 6 7
0 5 5 5 5 5 6 7
```

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

// na pocetku svaki element pripada zasebnom skupu
void inicijalizuj() {
  for (int i = 0; i < n; i++)
    id[i] = i;
}

// oznaku podskupa kome element pripada pamtimo na odgovarajucem mestu u nizu
int predstavnik(int x) {
```

```
   return id[x];
}

// elementi su u istom podskupu ako su im oznake iste
int u_istom_podskupu(int x, int y) {
  return predstavnik(x) == predstavnik(y);
}

// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int idx = id[x], idy = id[y];
  // oznake svih elemenata prvog podskupa menjamo u oznaku drugog podskupa
  for (int i = 0; i < n; i++)
    if (id[i] == idx)
      id[i] = idy;
}
```

Ako bismo u programu imali m operacija od kojih je svaka tipa `union` ili `find`, ukupno vreme izvršavanja ovog niza operacija bismo mogli da ocenimo kao $O(m \cdot n)$ jer iako se operacija `find` izvršava veoma efikasno – u vremenu $O(1)$, složenost operacije `union` je linearna u odnosu na broj elemenata koji održavamo. Stoga bismo želeli da razmotrimo neku drugačiju implementaciju ove strukture podataka u kojoj bi operacija `union` bila vremenski efikasnija.

Ključna ideja na kojoj se zasniva efikasnije rešenje je da elemente ne preslikavamo u oznake podskupova, već 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 za proizvoljni element saznali oznaku podskupa kom pripada, potrebno je da počev od tog elementa 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.

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

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.

Iako ovako opisana struktura podataka ima drvoliku strukturu, možemo je implementirati korišćenjem statičkog niza. Naime, pošto svaki element u drvetu ima jedinstvenog roditelja (osim korena) na poziciji nekog elementa u nizu možemo čuvati identifikator njegovog roditelja. U slučaju da je element koren nekog drveta njegov roditelj biće on sam.

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

```
// na pocetku svaki element pripada zasebnom skupu
void inicijalizuj() {
  for (int i = 0; i < n; i++)
    roditelj[i] = i;
}

// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int predstavnik(int x) {
  // sve dok ne stignemo do korena
  while (roditelj[x] != x)
    // penjemo se u roditeljski cvor
    x = roditelj[x];
  return x;
}

// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int fx = predstavnik(x), fy = predstavnik(y);
  // postavljamo da je koren prvog podskupa sin korena drugog podskupa
  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 $O(n)$. Ilustrujmo ovo jednim primerom: u prvom redu dati su indeksi elemenata u nizu, a u drugom vrednosti niza `roditelj`.

```
0  1  2  3  4  5  6  7
0  1  2  3  4  5  6  7

unija(7,6)

0  1  2  3  4  5  6  7
0  1  2  3  4  5  6  6

unija(6,5)

0  1  2  3  4  5  6  7
0  1  2  3  4  5  5  6

unija(5,4)

0  1  2  3  4  5  6  7
0  1  2  3  4  4  5  6

unija(4,3)
```

```
0  1  2  3  4  5  6  7
0  1  2  3  3  4  5  6

unija(3,2)

0  1  2  3  4  5  6  7
0  1  2  2  3  4  5  6

unija(2,1)

0  1  2  3  4  5  6  7
0  1  1  2  3  4  5  6

unija(1,0)

0  1  2  3  4  5  6  7
0  0  1  2  3  4  5  6
```

Upit kojim se traži predstavnik skupa kojem pripada element 7 se realizuje nizom koraka kojima se prelazi preko sledećih elemenata 7, 6, 5, 4, 3, 2, 1, 0. Iako ovo deluje lošije od prethodnog pristupa, gde je bar pronalaženje podskupa koštalo $O(1)$, kada su drveta izbalansirana, složenost svake od operacija je $O(\log n)$ 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 čija se unija pravi (visinu čvora 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 nazvati `rang`.

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

// na pocetku svaki element pripada zasebnom skupu
// i visina svakog drveta je 0
void inicijalizuj() {
  for (int i = 0; i < n; i++) {
    roditelj[i] = i;
    rang[i] = 0;
  }
}

// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
```

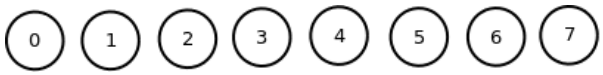
```
int predstavnik(int x) {
  // sve dok ne stignemo do korena
  while (roditelj[x] != x)
    // penjemo se u roditeljski čvor
    x = roditelj[x];
  return x;
}

// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int fx = predstavnik(x), fy = predstavnik(y);
  // postavljamo da je koren podskupa manjeg ranga
  // sin korena podskupa većeg ranga
  if (rang[fx] < rang[fy])
     roditelj[fx] = fy;
  else if (rang[fy] < rang[fx])
     roditelj[fy] = fx;
  else {
     roditelj[fx] = fy;
     // ako su podskupovi istog ranga
     // unija ce biti za jedan veceg ranga
     rang[fy]++;
  }
}
```

Primetimo da su nam samo relevantne vrednosti ranga korena podskupova.

Prikažimo rad algoritma na jednom primeru. Podskupove ćemo predstavljati drvetima. Pretpostavimo da je potrebno izvršiti niz narednih operacija:

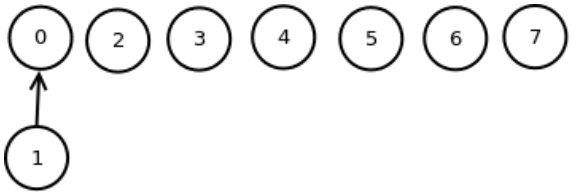
```
unija(0,1)
unija(5,6)
unija(3,6)
unija(4,7)
unija(0,2)
unija(4,3)
unija(2,6)
```

Na slikama 5, 6, 7, 8, 9, 10, 11 i 12 prikazano je stanje nakon izvršenja jedne po jedne operacije unije.

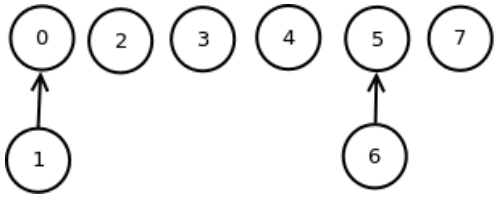


Slika 5: Ilustracija rada sa union-find strukturom (polazno stanje).

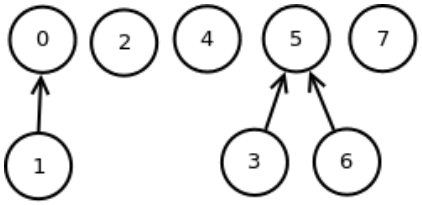
Dokažimo indukcijom da se u drvetu čiji je koren na visini h nalazi bar 2^h čvorova. Baza je početni slučaj u kome je svaki čvor svoj predstavnik. Visina svih čvorova je tada nula i sva drveta imaju $2^0 = 1$ čvor pa tvrđenje važi. Pokažimo da



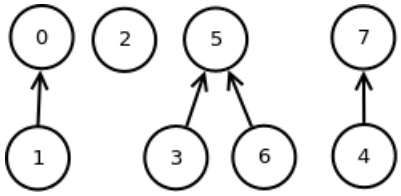
Slika 6: Stanje nakon izvršenje operacije `unija(0,1)`.



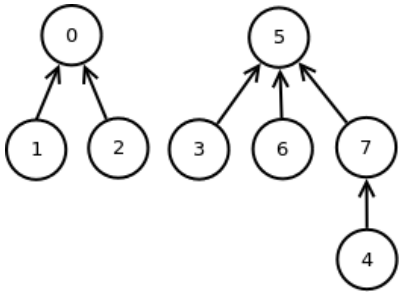
Slika 7: Stanje nakon izvršenje operacije `unija(5,6)`.



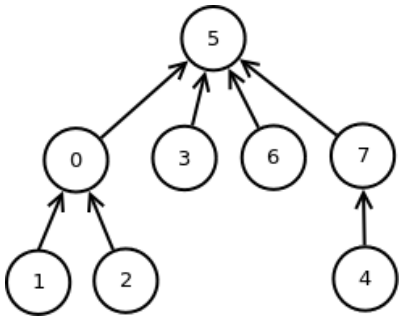
Slika 8: Stanje nakon izvršenje operacije `unija(3,6)`.



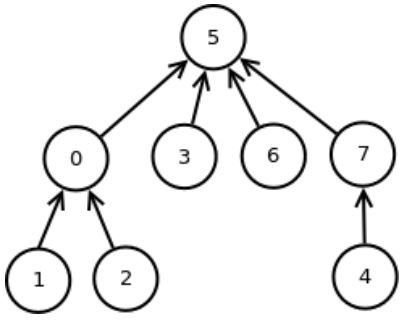
Slika 9: Stanje nakon izvršenje operacije `unija(4,7)`.



Slika 10: Stanje nakon izvršenje operacije `unija(0,2)`.



Slika 11: Stanje nakon izvršenje operacije `unija(4,3)`.



Slika 12: Stanje nakon izvršenje operacije `unija(2,6)`.

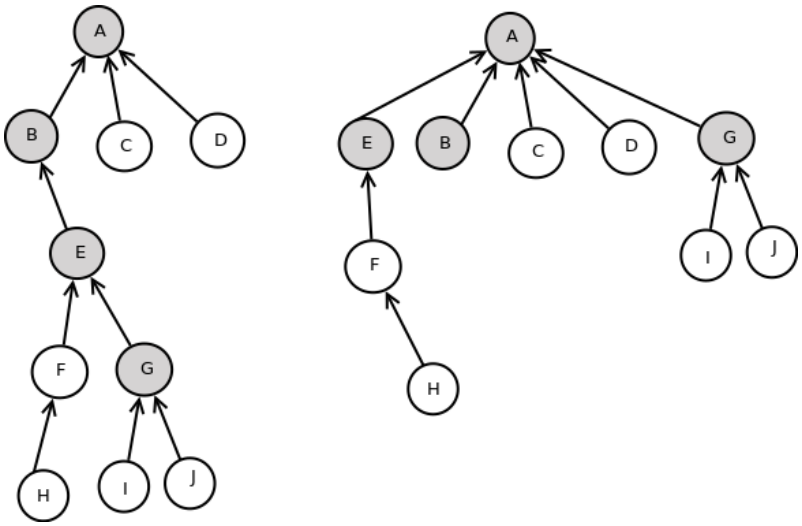
operacija unije održava ovu invarijantu. Po induktivnoj hipotezi pretpostavljamo da oba drveta koja predstavljaju podskupove koji se uniraju imaju visine h_1 i h_2 i redom bar 2^{h_1} i 2^{h_2} čvorova. Ukoliko se uniranjem visina ne poveća, invarijanta je trivijalno očuvana jer se broj čvorova uvećao, a visina je ostala ista. 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 $O(\log n)$.

Dakle, održavanje visina pod kontrolom nam garantuje logaritamsku složenost osnovnih operacija. Međutim, umesto visine moguće je održavati i broj čvorova u svakom od podskupova. Ako uvek usmeravamo predstavnika podskupa sa manjim brojem elemenata ka predstavniku podskupa sa većim brojem elemenata, ponovo ćemo dobiti logaritamsku složenost najgoreg slučaja za obe osnovne operacije. Ovo važi zato što i ovaj način pravljenja unije garantuje da ne možemo imati visoko drvo sa malim brojem čvorova. Da bi se dobilo drvo visine 1, potrebna su bar dva podskupa visine 0, odnosno bar 2 čvora; da bi se dobilo drvo visine 2 potrebna su bar dva drveta visine 1 koja imaju bar po dva čvora, odnosno drvo visine dva imaće bar 4 čvora. U opštem slučaju drvo visine h sadržaće bar 2^h čvorova. Odavde sledi da će visine svih drveta u ovoj strukturi podataka biti visine $O(\log n)$.

Iako je ova složenost sasvim prihvatljiva (složenost izvršavanja n operacija unije 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 (slika 13). Na ovaj način će buduće operacije biti efikasnije.

```
int predstavnik(int x) {
  int koren = x;
  // nalazimo oznaku podskupa kao koreni element podskupa
  while (koren != roditelj[koren])
    koren = roditelj[koren];
  // svim cvorovima na putanji od x do korena
  // postavljamo da je roditeljski cvor koren tog podskupa
  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. Ako rangove tumačimo kao broj čvorova



Slika 13: Ilustracija postupka kompresije putanje u dva prolaza nakon traženja predstavnika čvora G.

u podskupu, onda se prilikom kompresije putanje ta statistika ne menja, pa je postupak korektan. Ako pak rangove tumačimo kao visine, jasno je da prilikom kompresije putanje niz visina postaje neažuran. Međutim, interesantno je da ni u ovom slučaju nema potrebe da se on ažurira. Naime, brojevi koji se sada čuvaju u tom nizu ne predstavljaju više visine čvorova, već gornje granice visina čvorova. Ovi brojevi se nadalje smatraju rangovima čvorova 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.

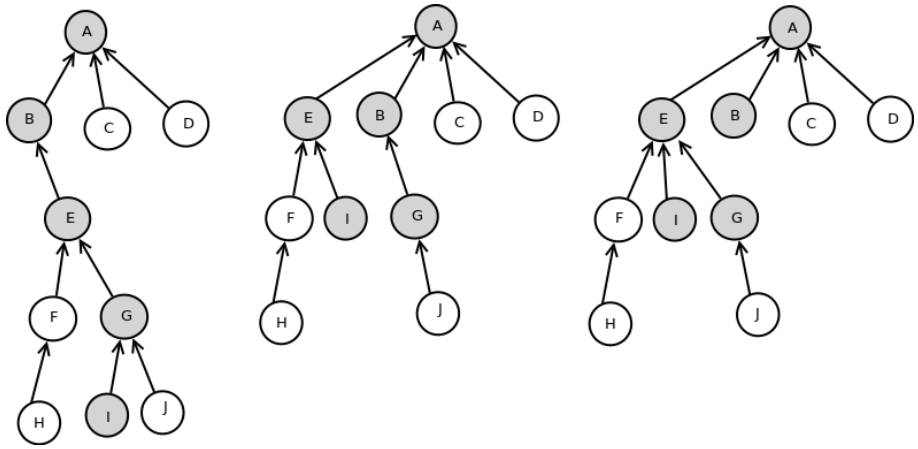
U prethodnoj implementaciji se dva puta prolazi kroz putanju od čvora do korena. Ipak, slične performanse se mogu dobiti i samo u jednom prolazu. Postoje dva načina na koji se ovo može uraditi: jedan od njih je da se svaki čvor usmeri ka roditelju svog roditelja. Za sve čvorove koji se obilaze od polaznog čvora do korena, dužine putanja do korena se nakon ovoga smanjuju dvostruko, što je dovoljno za odlične performanse (slika 14 sredina).

```
int predstavnik(int x) {
  int tmp;
  // za sve cvorove na putanji od x do korena
  while (x != roditelj[x]) {
    tmp = roditelj[x];
    // novi roditelj od x je roditelj njegovog roditelja
    roditelj[x] = roditelj[roditelj[x]];
    x = tmp;
  }
  return x;
```

```
}
```

Drugi način podrazumeva da se prilikom prolaska od čvora ka korenu svaki drugi čvor na putanji usmeri ka roditelju svog roditelja (slika 14 desno).

```
int predstavnik(int x) {
  // penjemo se do korena tako sto preskacemo po jedan cvor
  while (x != roditelj[x]) {
    // novi roditelj od x je roditelj njegovog roditelja
    roditelj[x] = roditelj[roditelj[x]];
    x = roditelj[x];
  }
  return x;
}
```



Slika 14: Ilustracija dva različita načina kompresije putanje u jednom prolazu tokom traženja predstavnika čvora I.

Primetimo da je na ovaj način dodata samo jedna linija koda u prvobitnu implementaciju operacije predstavnik. Ovom jednostavnom promenom amortizovana složenost operacija postaje samo $O(\alpha(n))$, gde je $\alpha(n)$ inverzna Akermanova funkcija koja jako 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.

Napomenimo da kada se koristi kompresija putanja i operacija `predstavnik` menja strukturu stabla.

Problem: Logička matrica dimenzije $n \times n$ 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 matrice (sa proizvoljnog polja prve vrste do proizvoljnog polja poslednje vrste matrice).

Osnovna ideja je da se formiraju svi podskupovi elemenata između kojih postoji put. Kada se uspostavi veza između dva elementa različitih podskupova, 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 matrice postoji ako postoji putanja od bilo kog elementa u prvoj vrsti matrice do bilo kog elementa u dnu matrice. To bi dovelo do toga da u svakom koraku moramo da proveravamo sve parove elemenata iz gornje i donje vrste matrice. 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) koji ćemo spojiti 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 dimenzije nxn
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, n), usce);

  // broj obrađenih jedinica
  int k = 0;
  while (k < jedinice.size()) {
    // čitamo narednu jedinicu
    int x = jedinice[k].first, y = jedinice[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 < n && 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 koje nazivamo *upiti raspona* (eng. 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 sa m označen broj upita, a sa n dužina niza, što je u slučaju dugačkih nizova i velikog broja upita nedopustivo neefikasno. Jednostavno rešenje je zasnovano na ideji da umesto da čuvamo elemente niza x, čuvamo niz zbirova prefiksa niza $P_i = \sum_{k=0}^{i} x_k$. Zbir svakog segmenta $[a, b]$ onda možemo razložiti na razliku prefiksa do elementa b i prefiksa do elementa $a - 1$: $\sum_{k=a}^{b} x_k = \sum_{k=0}^{b} x_k - \sum_{k=0}^{a-1} x_k = P_b - p_{a-1}$. 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 je i sledeći.

Problem: Dat je niz dužine n koji sadrži samo nule. Nakon toga izvršavaju se upiti oblika $([a, b], c)$, koji podrazumevaju da se svi elementi na pozicijama iz intervala $[a, b]$ uvećaju za vrednost c. 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 sa m označen broj upita, a sa n dužina niza.

Mnogo bolje rešenje se može dobiti ako se umesto elemenata niza pamte razlike između svaka dva susedna elementa niza: $R_0 = x_0, R_i = x_i - x_{i-1}$ za $i > 0$. Ključni uvid je da se tokom uvećavanja svih elemenata niza menjaju samo razlike između elemenata na pozicijama a i $a - 1$ (razlika R_a se uvećava za c) kao i između elemenata na pozicijama $b + 1$ i b (razlika R_{b-1} se umanjuje za c). 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 je $O(n)$). 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* (eng. static range queries), ovde ćemo razmatrati tzv. *dinamičke upite nad rasponima* (eng. dynamic range queries) koji dozvoljavaju da se niz menja tokom vremena, tako da je potrebno razviti naprednije strukture podataka koje omogućavaju izvršavanje oba tipa upita efikasno. Za početak razmotrimo malo jednostavniji 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.

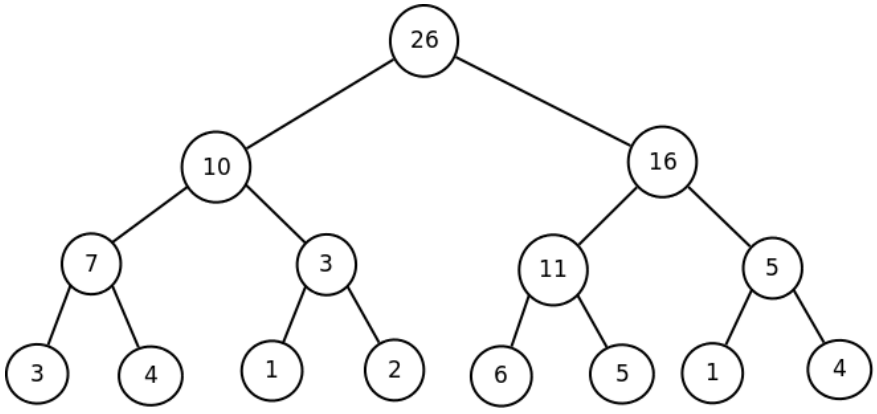
Dakle, pretpostavljamo da imamo dat niz od m operacija od kojih je svaka jednog od prethodno pomenuta dva tipa i da je cilj minimizovati ukupno vreme izvršavanja ovog niza operacija. U ovom slučaju nam nije od koristi struktura podataka kod koje se jedna od ove dve operacije izvršava efikasno, a druga neefikasno jer se može desiti da je većina (ili sve) od m operacija baš tog drugog tipa.

U nastavku ćemo videti dve različite, ali donekle slične strukture podataka koje daju efikasno rešenje prethodnog problema i njemu sličnih. Ideja obe ove strukture podataka na neki način nalikuje korišćenju prefiksnih suma: čuvaju se zbirovi nekih unapred pogodno izabranih segmenata i oni se koriste za efikasno računanje zbira proizvoljnog segmenta.

Segmentna drveta

Jedna struktura podataka koja omogućava prilično jednostavno i efikasno rešavanje ovog problema su *segmentna drveta* (eng. segment tree). Opet se tokom faze pretprocesiranja izračunavaju zbirovi određenih pogodno odabranih 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 ili nzs-a svih elemenata i slično).

Pretpostavimo da je dužina niza stepen broja 2; ako nije, niz se može dopuniti do najbližeg stepena broja 2, u slučaju računanja zbirova nulama. [3]. Članovi niza predstavljaju listove drveta. Grupišemo dva po dva susedna čvora i na svakom prethodnom nivou drveta čuvamo roditeljske čvorove koji čuvaju zbirove svoja dva deteta. Segmentno drvo za zbirove za niz $3, 4, 1, 2, 6, 5, 1, 4$, prikazano je na slici 15.



Slika 15: Primer segmentnog drveta.

Pošto je drvo potpuno, najjednostavnija implementacija je da se čuva implicitno u nizu (slično kao u slučaju hipa). Pretpostavićemo da elemente drveta smeštamo

[3] Ako se segmentnim stablom računaju proizvodi niz se može dopuniti do stepena dvojke jedinicama, ako se računa nzd može se dopuniti nulama jer važi $nzd(0, x) = nzd(x, 0) = x$, a za nzs jedinicama jer važi $nzs(1, x) = nzs(x, 1) = x$

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. Ako polazni niz sadrži n elemenata, onda se u segmentnom drvetu 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 $\lfloor \frac{k}{2} \rfloor$. Na primer, na prethodnoj slici kojom je predstavljeno segmentno stablo za niz od 8 elemenata možemo uočiti da se elementi polaznog niza nalaze na pozicijama $[8, 15]$. Levo dete čvora sa vrednošću 10 koji se nalazi na poziciji 2 je čvor sa vrednošću 7 koji se nalazi na poziciji $2 \cdot 2 = 4$, a desno dete je čvor sa vrednošću 3 koji se nalazi na poziciji $2 \cdot 2 + 1 = 5$. Roditelj i čvora na poziciji 4 i čvora na poziciji 5 je čvor koji se nalazi na poziciji $\lfloor \frac{4}{2} \rfloor = \lfloor \frac{5}{2} \rfloor = 2$.

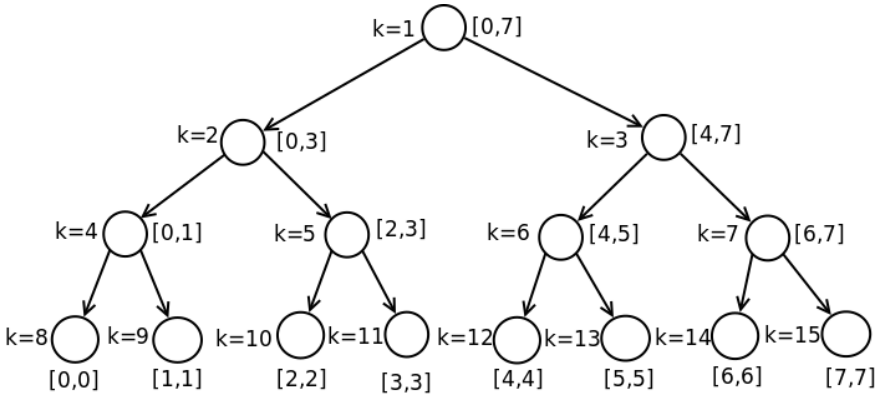
Formiranje segmentnog drveta na osnovu datog niza je veoma jednostavno. Prvo se elementi polaznog niza prekopiraju u drvo, krenuvši od pozicije n. Zatim se svi unutrašnji čvorovi drveta (od pozicije $n-1$, pa unazad do pozicije 1) popunjavaju kao zbirovi svoje dece (na poziciju k upisujemo zbir elemenata 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 polaznog niza. Segment je jednoznačno određen pozicijom k u nizu koji odgovara segmentnom drvetu, ali da bismo olakšali implementaciju granice tog segmenta možemo kroz rekurziju prosleđivati kao parametar funkcije, zajedno sa vrednošću k (neka je to segment $[x, y]$). Na slici 16 za svaki čvor segmentnog drveta dat

je njegov indeks u odgovarajućem nizu `drvo` i granice segmenta koje taj čvor pokriva.



Slika 16: Prikaz segmentnog drveta i za svaki čvor segmenta koji taj čvor pokriva.

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 poddrvo 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);
}
```

Vremensku složenost prethodne rekurzivne implementacije možemo opisati narednom rekurentnom jednačinom:

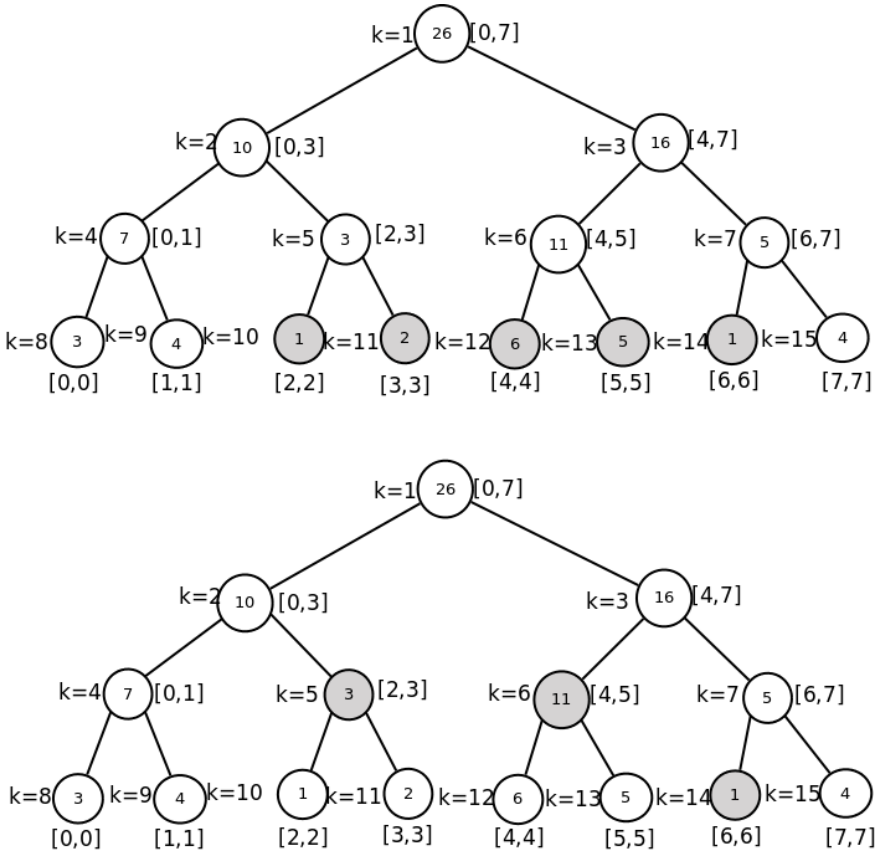
$$T(n) = 2T(n/2) + O(1), T(1) = O(1).$$

Njeno rešenje se dobija direktno iz master teoreme i iznosi $O(n)$.

Razmotrimo sada kako bismo našli zbir elemenata niza $3, 4, 1, 2, 6, 5, 1, 4$ na pozicijama iz segmenta $[2, 6]$, tj. zbir elemenata $1, 2, 6, 5, 1$. U segmentnom drvetu taj segment je smešten na pozicijama $[2 + 8, 6 + 8] = [10, 14]$ (slika 17). Jasno je da ne treba ići do listova i redom sabirati elemente, već na pametan način iskoristiti već izračunate sume segmenata koje se nalaze u stablu. Naime, zbir prva dva elementa traženog segmenta $(1, 2)$ se nalazi u čvoru iznad njih (na poziciji 5), zbir naredna dva elemenata $(6, 5)$ takođe u čvoru iznad (na poziciji 6), dok se u roditeljskom čvoru elementa 1 nalazi njegov zbir sa elementom 4, koji ne pripada segmentu koji sabiramo. Zato zbir elemenata na pozicijama $[10, 14]$ u segmentnom drvetu možemo razložiti na zbir elemenata segmenta na pozicijama $[5, 6]$ i elementa na poziciji 14. Na ovaj način penjemo se na nivo iznad tekućeg (dolazimo do roditelja tih čvorova) i nastavljamo algoritam na segmentu $[5, 6]$ na isti način. 5 je desni sin svog roditelja, a 6 levi sin svog roditelja, te oba elementa dodajemo zasebno u zbir i ostajemo sa praznim segmentom.

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, a zbir sva četiri data elementa u čvoru iznad 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 segmenta levo dete (ako se nalazi na parnoj poziciji), tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom desno od njega, koji ne



Slika 17: Računanje zbira elemenata iz segmenta $[2, 6]$ pristupom odozdo naviše. Na prvoj slici su sivom bojom označeni listovi koji odgovaraju traženom segmentu, a na drugoj čvorovi čije se vrednosti sabiraju.

pripada segmentu koji sabiramo. I u toj situaciji, taj element ćemo posebno dodati na zbir i isključiti iz segmenta koji sabiramo pomoću roditeljskih čvorova. Konačno, 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;
   int zbir = 0;
   // sve dok je segment neprazan
   while (a <= b) {
      // ako je levi kraj segmenta desno dete, dodajemo ga posebno u zbir
      if (a % 2 == 1) zbir += drvo[a++];
      // ako je desni kraj segmenta levo dete, dodajemo ga posebno u zbir
      if (b % 2 == 0) zbir += drvo[b--];
      // penjemo se na nivo iznad, na roditeljske cvorove
      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 $O(\log n)$.

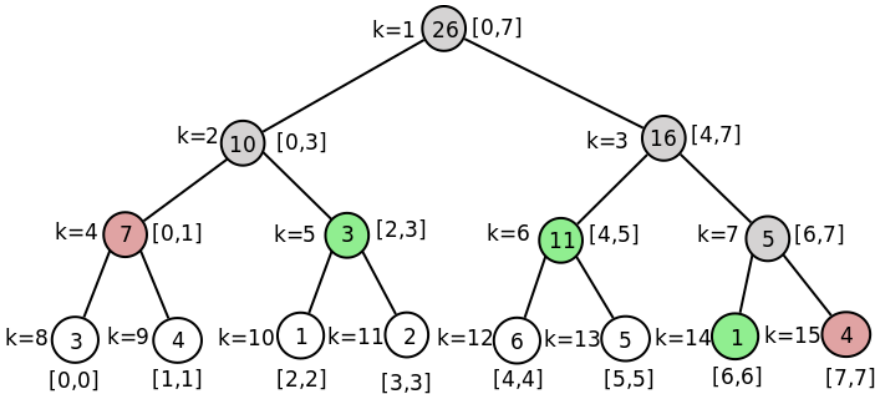
Prethodna implementacija vrši izračunavanje odozdo naviše. I za ovu operaciju možemo napraviti i rekurzivnu implementaciju koja vrši izračunavanje odozgo naniže. U opštem slučaju, neki elementi traženog segmenta su levo od tekućeg čvora, a neki desno (ili su svi levo ili svi desno). Za svaki čvor u segmentnom drvetu funkcija vraća koliki je doprinos segmenta koji odgovara tom čvoru i njegovim naslednicima traženom zbiru elemenata na pozicijama iz segmenta $[a, b]$ u polaznom nizu. Na početku krećemo od korena i računamo doprinos celog drveta zbiru elemenata iz segmenta $[a, b]$. Postoje tri različita moguća odnosa između segmenta $[x, y]$ koji odgovara tekućem čvoru i segmenta $[a, b]$ čiji zbir elemenata tražimo. 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 [a, b] u polaznom nizu
// a 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;
  // a i b se ne menjaju kroz rekurzivne pozive, dok se k, x i y menjaju
  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);
}
```

Razmotrimo kako se ovim pristupom određuje zbir elemenata na pozicijama iz segmenta $[2, 6]$. Izvršavanje kreće od korena. Segment $[0, 7]$ se seče sa segmentom $[2, 6]$ te će zbir biti jednak sumi doprinosa segmenata $[0, 3]$ i $[4, 7]$. Segment $[0, 3]$ se seče sa segmentom $[2, 6]$ te opet startujemo dva rekurzivna poziva za segmente $[0, 1]$ i $[2, 3]$. Segment $[0, 1]$ je disjunktan sa segmentom $[2, 6]$ pa je njegov doprinos traženoj sumi 0, a segment $[2, 3]$ je sadržan u segmentu $[2, 8]$ te je njegov doprinos potpun. S druge strane, segment $[4, 7]$ se seče sa segmentom $[2, 6]$ te opet startujemo dva rekurzivna poziva za segmente $[4, 5]$ i $[6, 7]$. Segment $[4, 5]$ je u potpunosti sadržan u segmentu $[2, 8]$ te je njegov doprinos potpun, a segment $[6, 7]$ se seče sa segmentom $[2, 6]$, pa iz njega startujemo dva rekurzivna poziva: za segmente $[6, 6]$ i $[7, 7]$: prvi je u potpunosti sadržan u traženom segmentu, te je njegov doprinos potpun, a drugi je disjunktan pa je njegov doprinos jednak nula.

I ova implementacija će imati složenost $O(\log n)$. Naime može se pokazati da se za svaki nivo segmentnog drveta obilaze najviše po četiri čvora, a pošto je visina segmentnog drveta $O(\log n)$, dobijamo datu ocenu složenosti. Dokažimo ovo tvrđenje principom matematičke indukcije. Na prvom nivou se posećuje samo jedan čvor, koren drveta, tako da se na ovom nivou posećuje manje od četiri čvora i baza indukcije važi. Razmotrimo sada proizvoljni nivo drveta: prema induktivnoj hipotezi na njemu se posećuje najviše četiri čvora. Ako se posećuje najviše dva čvora, u narednom nivou se posećuje najviše četiri čvora jer svaki čvor može da proizvede najviše dva rekurzivna poziva. Pretpostavimo da se na tekućem nivou posećuju tri ili četiri čvora. Oni moraju biti susedni (slika 18,



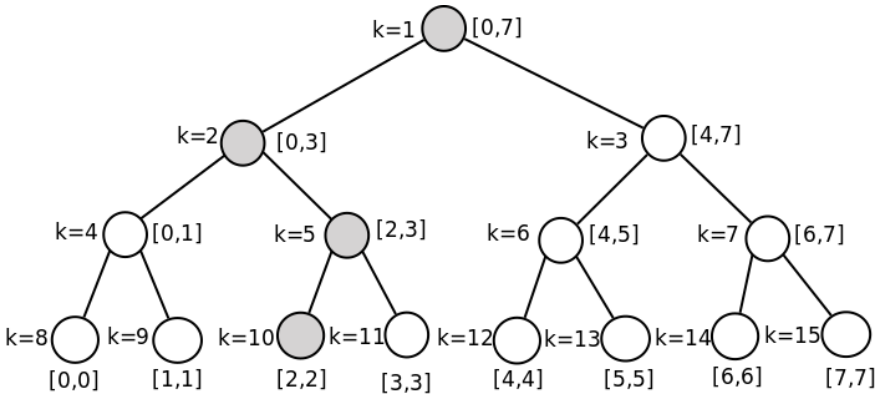
Slika 18: Računanje zbira elemenata iz segmenta $[2, 6]$ pristupom odozgo naniže. Sivom bojom su označeni čvorovi koji odgovaraju segmentima koji se seku sa traženim segmentom, roze bojom čvorovi koji su disjunktni, a zelenom bojom čvorovi koji odgovaraju segmentima koji su u potpunosti sadržani u traženom segmentu.

nivo 2). S obzirom na to da se traži suma segmenta (niza uzastopnih elemenata), možemo zaključiti da će od ta najviše četiri posećena čvora na tekućem nivou središnji čvorovi biti u potpunosti sadržani u traženom segmentu, te se iz njih neće startovati rekurzivni poziv. Stoga, samo krajnji levi i krajnji desni čvor imaju mogućnost da generišu dva nova rekurzivna poziva, tako da će i naredni nivo zadovoljavati polazno tvrđenje. Dakle, posećuje se ukupno najviše $4 \log n$ čvorova segmentnog drveta, pa je vremenska složenost i ovog pristupa $O(\log n)$.

Prilikom ažuriranja nekog elementa potrebno je ažurirati sve čvorove na putanji od tog lista do korena (slika 19). S obzirom na to 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 $O(\log n)$.



Slika 19: Ažuriranje elementa na poziciji 2 u polaznom nizu. Sivom bojom su označeni čvorovi čije se vrednosti ažuriraju.

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 prethodne implementacije možemo opisati rekurentnom jednačinom:

$$T(n) = T(n/2) + O(1), T(1) = O(1).$$

i njeno rešenje iznosi: $O(\log n)$. Naime dužina intervala $[x, y]$ se u svakom nardnom pozivu smanjuje dva puta.

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 jednostavno izražava preko one druge.

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