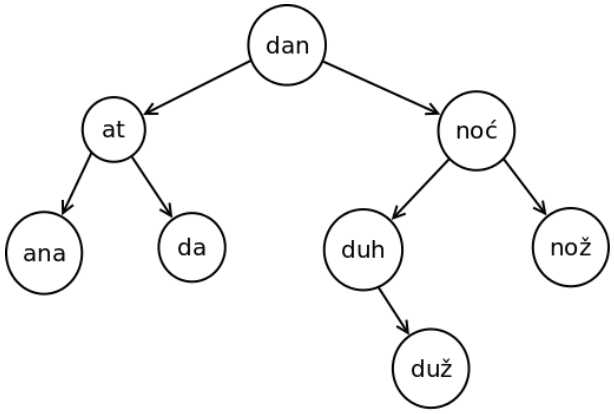
Napredne strukture podataka

Prefiksno drvo

Uređena binarna drveta i heš tabele omogućavaju efikasnu implementaciju struktura podataka sa asocijativnim pristupom[1] (skupova, mapa) kod kojih se pristup elementima vrši po ključu. Ključ ne mora biti celobrojna vrednost, već niska karaktera, niska bitova 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 drvo čiji su ključevi niske.

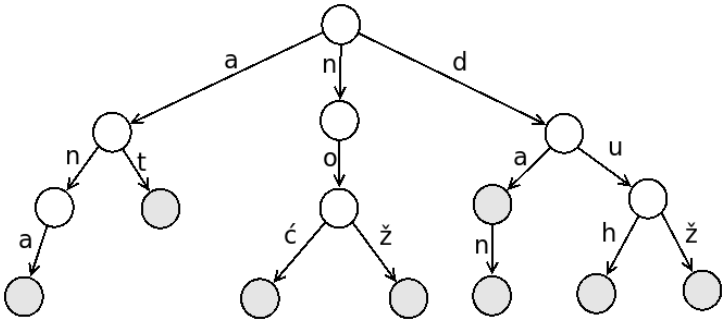
Prilikom svih operacija sa uređenim binanrnim drvetom (pretraga, brisanje, umetanje) u svakom čvoru se vrši poređenje ključeva (onog koji piše u čvoru i onog koji se obrađuje) i kada su ključevi niske, to poređenje može zahtevati dosta vremena i loše uticati na performanse.

Još jedna struktura podataka u vidu drveta koja omogućava efikasan asocijativni pristup kada su ključevi niske je *prefiksno drvo* (eng. prefix tree) takođe poznato pod engleskim nazivom *trie* (od engleske reči *reTRIEeval*).

Jedan primer prefiksnog drveta, kod kojeg su prikazane oznake pridružene granama, a ne čvorovima, dat je na slici 2.

Osnovna ideja ove strukture podataka je da se ključ pridružen svakom čvoru dobija nadovezivanjem karaktera koji se nalaze na granam duž putanje od korena do tog čvora. Koren sadrži praznu reč, a prelaskom preko svake grane se na do tada formiranu reč zdesna nadovezuje još jedan karakter. Pritom, zajednički

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

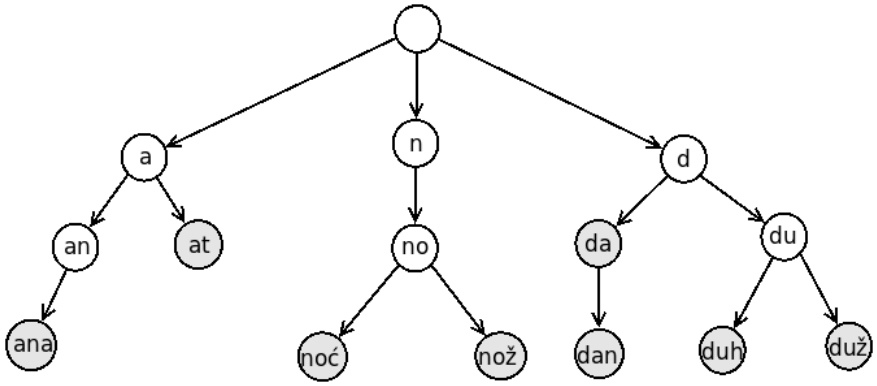


Slika 2: Primer prefiksnog drveta.

prefiksi različitih ključeva su predstavljeni istim putanjama od korena do tačke razlikovanja.

Ključevi koje ovo prefiksno drvo čuva su isti oni koje čuva uređeno binarno drvo sa slike 1: *ana*, *at*, *noć*, *nož*, *da*, *dan*, *duh* i *duž*. Primetimo da se većina ključeva u koje prefiksno drvo ovom primeru čuva završava u nekom od listova. Međutim, to ne mora biti slučaj: ključ *da* se ne završava u listu. Stoga je potrebno da svaki čvor prefiksnog drveta čuva i informaciju o tome da li se njime kompletira neki ključ (i u tom slučaju sadržati odgovarajući podatak) ili ne. Na prethodnoj slici ti čvorovi su obojeni.

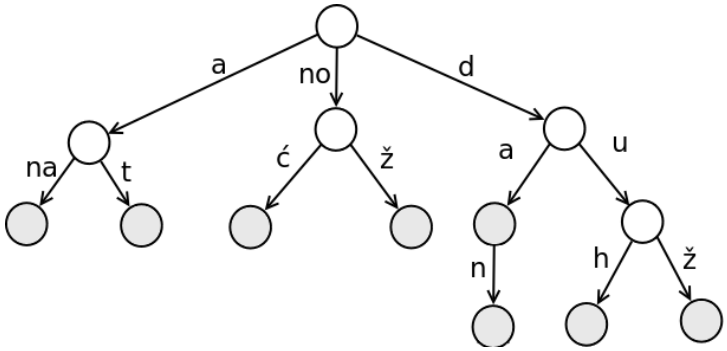
Ilustracije radi na slici 3 uz čvorove prefiksnog drveta prikazane su oznake akumulirane do tih čvorova. Treba imati u vidu da ovaj prikaz ne ilustruje implementaciju, već samo prefikse duž grane.



Slika 3: Primer prefiksnog drveta sa prikazanim akumuliranim prefiksima za svaki od čvorova.

Mana prefiksnih drveta može biti to što zauzimaju puno memorije i stoga je poželjno komprimovati ih. 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), pri čemu grane mogu biti označene niskama, a ne samo karakterima.



Slika 4: Primer prefiksnog drveta kod koga su grane označene niskama.

Pored opšteg asocijativnog pristupa podacima (implementacije skupova i mapa), očigledna primena prefiksnih drveta je i implementacija konačnih rečnika, na primer u svrhe automatskog kompletiranja ili provere ispravnosti reči koje korisnik kuca na računaru ili mobilnom telefonu.

Ključevi u prefiksnim drvetima ne moraju biti isključivo niske karaktera. Na primer, u prefiksnim drvetima možemo čuvati i ključeve koji su prirodni brojevi, pri čemu se tada koristi niz cifara u njihovom dekadnom ili binarnom zapisu. U praksi se pored niski, najčešće koriste binarne reprezentacije brojeva fiksne širine (zapisanih sa fiksnim brojem binarnih cifara).

Operacije pretrage i umetanja elemenata u prefiksno drvo se vrše na prilično očigledan način, 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 koji sadrži skup reči engleske abecede. U ovom slučaju jedino je potrebno podržati operaciju dodavanja ključeva u strukturu podataka i ključevima nema potrebe pridruživati 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 engleske abecede). Međutim, efikasnije je u svakom čvoru čuvati informacije samo o onim karakterima za koje postoji grana iz tog čvora, odnosno u ove svrhe iskoristiti mape. Dakle, u svakom čvoru prefiksnog drveta čuvaćemo neuređenu (heš) mapu grana koje kreću iz tog čvora, obeleženih karakterima ka deci, pri čemu koristimo bibliotečku implementaciju heš-mape. Implementacija umetanja i pretrage se

može jednostavno implementirati bilo rekurzivno bilo iterativno (u nastavku je prikazana rekurzivna implementacija).

```
#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 poziciji 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 kada je azbuka kojom se kodiraju ključevi veličine K i kada se u svakom čvoru čuva neuređena mapa grana, složenost operacija pretraživanja, umetanja i brisanja elementa iz prefiksnog drveta je u najgorem slučaju $O(MK)$, gde je M dužina reči koja se traži, umeće ili briše. Zaista, složenost najgoreg slučaja pretrage neuređene mape je $O(K)$, a prilikom obrade ključa dužine M vrši se M takvih pretraga. Sa druge strane, amortizovana složenost pretrage neuređene mape je $O(1)$, pa je amortizovana složenost ovih operacija $O(M)$. Ako se radi sa niskama karaktera i ako se neuređena mapa implementira pomoću niza od m elemenata onda je i složenost najgoreg slučaja $O(M)$.

Prednost prefiksnog drveta je što složenost zavisi od dužine zapisa ključa, a ne od broja elemenata koji se čuvaju u drvetu. Mana je potreba za čuvanjem pokazivača uz svaki čvor u drvetu. Štaviše prostorna složenost prefiksnog drveta u najgorem slučaju iznosi $O(M \cdot N \cdot K)$, 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 karaktera među ključevima, dok je prostorna složenost svakog čvora jednaka $O(K)$ zbog potrebe čuvanja mape u

svakom čvoru. Primetimo da je očekivana prostorna složenost manja jer će se u slučaju realne konačne azbuke (na primer, engleske abecede) prvo preklapanje javiti već nakon 26 ključeva.

Smanjenje memorijske složenosti se može postići i tako što se smanji veličina azbuke (po cenu povećanje dužine ključa). Na primer, umesto 256 različitih 8-bitnih karaktera, možemo svaki karakter podeliti na dve 4-bitne polovine čime se dobija 16 različitih 4-bitnih karaktera, ali se dužina svakog ključa dva puta povećava.

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

Kada bi se koristila heš tabela, prilikom svake operacije bi morala da bude izračunata heš vrednost za šta je potrebno vreme $O(M)$. U zavisnosti od broja kolizija, vršila bi se poređenja heš vrednosti (u najgorem slučaju njih $O(N)$, a amortizovano $O(1)$). Na kraju bi ključevi morali da budu eksplicitno upoređeni, za šta je takođe potrebno vreme $O(M)$, tako da bi složenost najgoreg slučaja operacija bila $O(N + M)$, a amortizovana složenost $O(M)$. Pošto ključ mora biti zapisan eksplicitno u svakom slogu tabele, prostorna složenost je $O(M \cdot N)$.

Disjunktni podskupovi – union-find struktura podataka

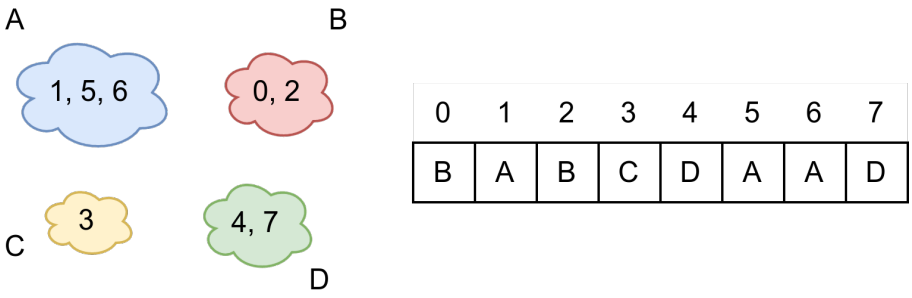
Ponekad je u programu potrebno održavati nekoliko disjunktnih podskupova određenog skupa, pri čemu je potrebno umeti za dati element efikasno pronaći kom podskupu pripada (tu operaciju zovemo `find`) i efikasno spojiti dva zadata podskupa u novi, veći podskup (tu operaciju zovemo `union`). Pritom argumenti operacije `union` ne moraju biti oznake podskupova čiju uniju treba kreirati, već se kao argumenti mogu proslediti proizvoljni elementi tih podskupova. 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 ove oznake jednake. Ovakva struktura podataka se često naziva *union-find* ili *DSU* (engl. disjoint set union).

Na primer, ovo je potrebno kada god radimo sa nekim relacijama ekvivalencije i kada je potrebno predstaviti klase ekvivalencije (koje su disjunktni podskupovi skupa na kom je relacija definisana). Provera da li su dva elementa u relaciji se zasniva na proveri da li pripadaju istoj klasi, a uspostavljanjem relacije između bilo koja dva elementa dovodi do spajanja njihovih klasa ekvivalencije.

Naivna implementacija

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 kojom bi se oznake elemenata preslikavale u oznake podskupa kom element pripada). Primer takve reprezentacije prikazan je na narednoj slici.



Slika 5: Predstavljanje podskupova običnim nizom

Operacija `find` je tada trivijalna: samo se iz niza pročita oznaka podskupa kom element pripada 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 $O(n)$. Ukoliko bi se umesto niza u za ovo koristila neuređena mapa i ako bismo pretpostavili da se nad njom operacije vrše u složenosti $O(1)$ (što je zaista amortizovana složenost operacija nad neuređenom mapom), operacija `find` bila bi složenosti $O(1)$ a operacija `union` složenosti $O(n)$ (jer je potrebno menjati oznake podskupova za veliki broj elemenata). 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.

Primer 1:

Ilustrujmo sprovođenje operacija nad ovakvom implementacijom strukture podataka za disjunktne podskupove na jednom primeru. Pretpostavimo da je početno stanje takvo da svaki element pripada zasebnom podskupu. Razmotrimo kako bi se menjao sadržaj odgovarajućeg niza nakon izvršavanja operacija unije. U prvom redu prikazani su indeksi elemenata u nizu a u drugom vrednosti elemenata niza koje predstavljaju oznake podskupova kojima elementi pripadaju. Pretpostavićemo da će prilikom izvršavanja operacije `union(x,y)` oznaka novog podskupa biti oznaka skupa kome pripada element `y`.

Prikažimo sada implementaciju ove tehnike.

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

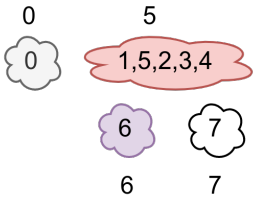
unija(1, 5)

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

unija(2, 1)

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

unija(3, 2)



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

Slika 6: Primer primene operacije unije

```
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 pripada element x citamo sa pozicije x iz niza
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;
}
```

Obratimo pažnju da se prilikom izvođenja unije moraju koristiti pomoćne promenljive (razmislite zašto je naredna implementacija neispravna).

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

Efikasna implementacija

Razmotrimo nešto drugačiju implementaciju ove strukture podataka u kojoj je operacija `union` 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 predstavljenih tim drvetom. Da bismo za proizvoljni element saznali oznaku podskupa kom pripada, potrebno je da počev od tog elementa prođemo kroz niz roditeljskih čvorova 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 dva podskupa u ovom pristupu možemo jednostavno realizovati tako što koren jednog podskupa usmerimo ka korenu drugog.

Prvi od opisanih algoritama unije 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 `union` i `find` $O(n)$. 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, u ovom pristupu složenost svake od operacija `union` i `find` je $O(\log n)$. Na ovaj način se postiže da se niz od m operacija tipa `union` ili `find` izvršava u vremenu $O(m \log n)$, dok bi u prvom pristupu ovaj niz operacija u slučaju kada je većina operacija tipa `union` bilo složenosti $O(mn)$. Da bi se na ovoj ideji mogla izgraditi efikasna struktura podataka ključna stvar je na neki način obezbediti 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]++;
  }
}
```

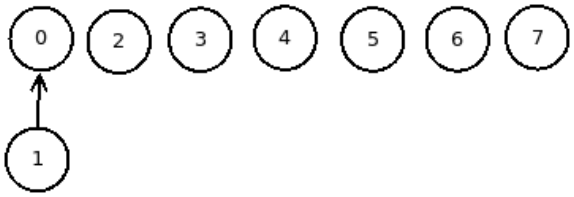
Primetimo da su nam samo relevantne vrednosti ranga korena podskupova.

Primer 2: Prikažimo rad algoritma na jednom primeru. Podskupove ćemo predstavljati drvetima. Pretpostavimo da je potrebno izvršiti niz narednih operacija:

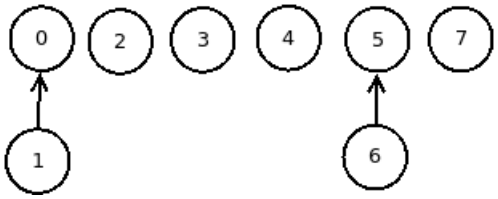
```
unija(0,1)
unija(5,6)
unija(3,6)
unija(4,7)
unija(0,2)
unija(4,3)
unija(2,6)
```

Na slikama 7, 8, 9, 10, 11, 12, 13 i 14 prikazano je stanje nakon izvršenja jedne po jedne operacije unije.

Slika 7: Ilustracija rada sa union-find strukturom (polazno stanje).

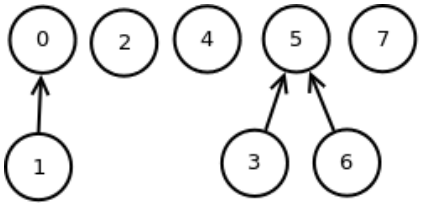


Slika 8: Stanje nakon izvršene operacije `unija(0,1)`.

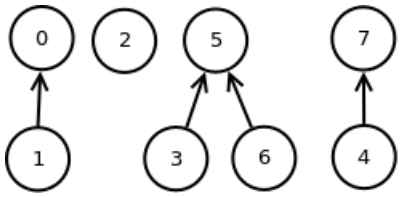


Slika 9: Stanje nakon izvršene operacije `unija(5,6)`.

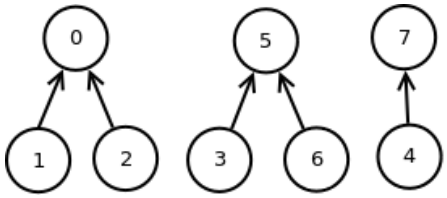
Dokažimo matematičkom indukcijom da se u drvetu čiji je koren na visini h nalazi bar 2^h čvorova. Baza indukcije odgovara polaznom stanju u kome je svaki čvor svoj predstavnik. Visina svih čvorova je tada nula i sva drveta imaju $2^0 = 1$ čvor pa tvrđenje važi. Pokažimo da operacija unije održava ovu 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



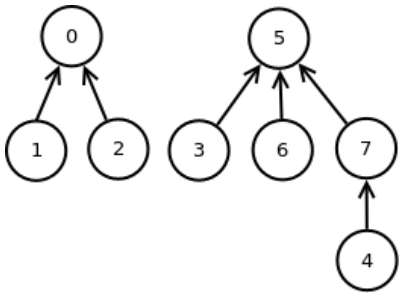
Slika 10: Stanje nakon izvršene operacije `unija(3,6)`.



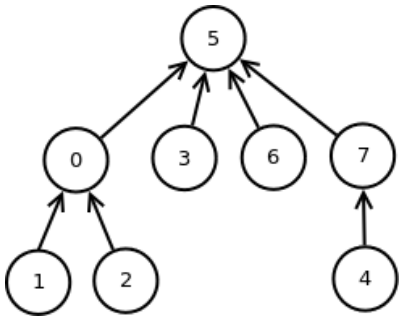
Slika 11: Stanje nakon izvršene operacije `unija(4,7)`.



Slika 12: Stanje nakon izvršene operacije `unija(0,2)`.



Slika 13: Stanje nakon izvršene operacije `unija(4,3)`.



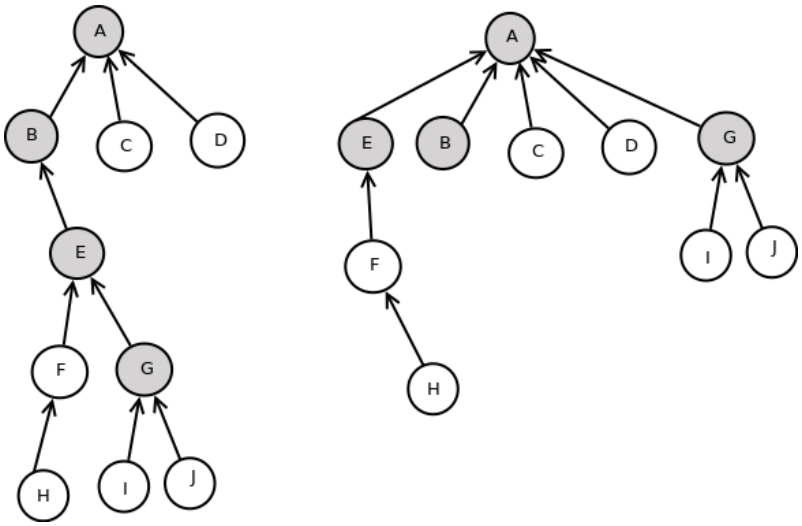
Slika 14: Stanje nakon izvršene operacije `unija(2,6)`.

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 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 uniranja dva podskupa je $O(\log n)$.

Dakle, održavanje visina podskupova 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. Naime, 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 m operacija unije je $O(m \log n)$), može se dodatno poboljšati veoma jednostavnom tehnikom poznatom kao *kompresija putanje* (eng. path compression). 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 15). Na ovaj način se postiže da buduće operacije nad tim podskupom budu 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
```



Slika 15: Ilustracija postupka kompresije putanje u dva prolaza nakon traženja predstavnika čvora G.

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

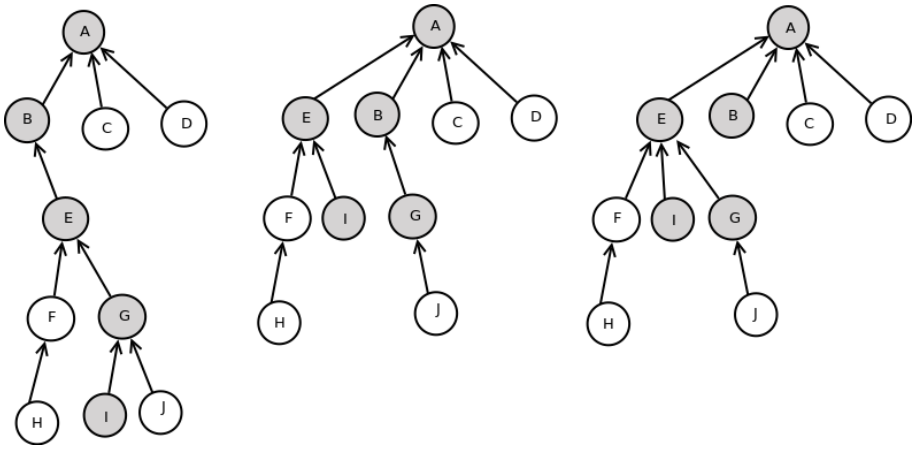
Za sve čvorove koji se obilaze od polaznog čvora do korena, dužine putanja do korena se nakon ovoga smanjuju na 1. Ako rangove tumačimo kao broj čvorova u podskupu, onda se prilikom kompresije putanje ta statistika ne menja, pa je postupak korektan. Ako pak rangove tumačimo kao visine, jasno je da prilikom 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 x 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 kroz koji se prolazi tokom pronalaženja predstavnika 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 16, 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 16, 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 16: 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 pored operacije `union` i operacija `find` menja strukturu drveta kojim je predstavljen taj podskup.

Primer 3:

Logička matrica dimenzije $n \times n$ u početku sadrži sve nule. Nakon toga se u nju 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).

Ideja na kojoj se zasniva efikasno rešenje je da se formiraju svi podskupovi elemenata između kojih postoji put. Kada se uspostavi veza između dva elementa različitih podskupova, podskupovi se uniraju. Provera da li postoji put između dva polja matrice 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 proizvoljnog elementa u prvoj vrsti matrice do proizvoljnog elementa u dnu matrice. To dovodi do toga da u svakom koraku algoritma moramo da proveravamo za sve parove elemenata iz gornje i donje vrste matrice da li pripadaju istom podskupu. Međutim, pokazuje se da možemo i bolje. Dodaćemo veštački početni čvor (nazovimo ga izvor) koji ćemo spojiti sa svim poljima iz prve vrste matrice i završni čvor (nazovimo ga ušće) koji ćemo spojiti sa svim poljima iz poslednje vrste matrice. Tada je u svakom koraku dovoljno 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 dimenzije 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;
}
```