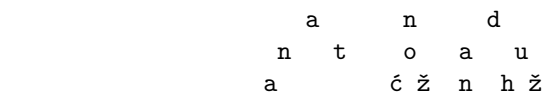
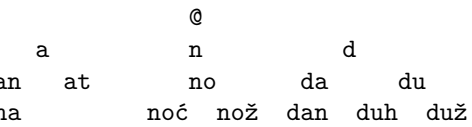
Napredne strukture podataka - čas 1

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 putanje od korena do listova ili do nekih od unutrašnjih čvorova, 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 ovakvog drveta, kod kojeg su prikazane oznake pridružene granama, a ne čvorovima, je sledeći:



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 ilustruje implementaciju, već samo prefikse duž grane. Simbol @ predstavlja praznu reč.



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.

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, dok je složenost ovih operacija u prosečnom slučaju $O(n)$. Kada bi se umesto prefiksnog drveta koristilo balansirano uređeno binarno drvo, složenost ovih operacija bi u najgorem slučaju bila $O(M \log N)$, gde je sa N označen ukupan broj ključeva koji se čuvaju u drvetu, a sa M maksimalna dužina ključa. 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. Štaviše prostorna složenost prefiksnog drveta je $O(M \cdot m \cdot N)$, gde je sa N označen broj ključeva koji se čuvaju u prefiksnom drvetu, a sa M maksimalna dužina ključa.

Pretraga i umetanje 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.

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
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 cvor {
  bool krajKljuca = false;
  unordered_map<char, node*> 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);
}

// umeće sufiks reči w od pozicije i u drvo na čiji koren ukazuje
// pokazivač trie
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]].second, w, i+1);
}

// umeće reč w u drvo na čiji koren ukazuje pokazivač w
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;
}
```

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 (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 $O(n)$.

```
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 u_istom_podskupu(int x, int y) {
  return predstavnik(x) == predstavnik(y);
}

void 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 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.

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.

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

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

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

void 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 $O(n)$. Ilustrujmo ovo jednim primerom.

```
0  1  2  3  4  5  6  7

unija 7 6

0  1  2  3  4  5  6  6

unija 6 5

0  1  2  3  4  5  5  6

unija 5 4

0  1  2  3  4  4  5  6
```

```
unija 4 3

0  1  2  3  3  4  5  6

unija 3 2

0  1  2  2  3  4  5  6

unija 2 1

0  1  1  2  3  4  5  6

unija 1 0

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, 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 nazvati `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;
```

```
}

void 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[fy]++;
  }
}
```

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 početni slučaj u kome je svaki čvor svoj predstavnik. Visina svih čvorova je tada nula i sva drveta imaju $2^0 = 1$ čvor. Pokažimo da svaka unija održava ovu invarijantu. Po induktivnoj hipotezi pretpostavljamo da oba drveta koja 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. 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 čvora; da bi se dobilo drvo visine 2 bar četiri čvora (jer se spajaju dva drveta visine 1 koja imaju bar po dva čvora). Da bi se dobilo drvo visine h potrebno je bar 2^h čvorova. Odavde sledi da će visine svih drveta u ovoj strukturi biti visine $O(\log n)$.

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 broj čvorova u podskupu, onda se prilikom kompresije putanje ta statistika i ne menja, pa je postupak korektan. Ako 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.

```
int predstavnik(int x) {
  while (x != roditelj[x]) {
    tmp = roditelj[x];
    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.

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

Primetimo da je ovim dodata samo jedna linija koda u prvobitnu implementaciju. Ovom jednostavnom promenom amortizovana 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 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 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 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. 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, 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 (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 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 $[x, y]$). 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 dva 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 na pozicijama $[10, 14]$ u segmentnom drvetu 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, 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 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;
   int 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 $[a, b]$ 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 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.