Napredne strukture podataka (nastavak) - čas 2

Fenvikova drveta (BIT)

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

Iako se naziva drvetom, Fenvikovo drvo zapravo predstavlja niz vrednosti zbirova nekih pametno izabranih segmenata. Izbor segmenata je u tesnoj vezi sa binarnom reprezentacijom indeksa. Ponovo ćemo jednostavnosti radi pretpostaviti da se vrednosti u nizu smeštaju od pozicije 1 (vrednost na poziciji 0 je irelevantna) i to i u polaznom nizu i u nizu u kom se smešta drvo. Prilagođavanje koda situaciji u kojoj su u polaznom nizu elementi smešteni od pozicije nula, veoma je jednostavno (samo je na početku svake funkcije koja radi sa drvetom indeks polaznog niza potrebno uvećati za jedan pre dalje obrade). Ako je polazni niz dužine n, elementi drveta će se smeštati u poseban niz na pozicije $[1, n]$.

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

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

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

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

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

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

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

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

Kada znamo zbir prefiksa, zbir proizvoljnog segmenta $[a, b]$ možemo izračunati kao razliku zbira prefiksa $(0, b]$ i zbira prefiksa $(0, a-1]$. Pošto se oba računaju u vremenu $O(\log n)$, i zbir svakog segmenta možemo izračunati u vremenu $O(\log n)$. Napomenimo i to da je zbog ove operacije važno da asocijativna operacija koja se koristi u Fenvikovom drvetu ima inverz (u ovom slučaju da bismo mogli da oduzimanjem dve vrednosti prefiksa dobijemo zbir proizvoljnog prefiksa).

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

Na primer, ako bi se u prethodnom primeru element na poziciji 3 uvećao za vrednost 4, bilo bi potrebno povećati za 4 vrednosti elemenata Fenvikovog drveta na pozicijama 3, 4 i 8. Do ovih pozicija bismo došli počev od binarnog zapisa broja 3 koji iznosi 11 sabiranjem sa 1 (broj koji sadrži tačno jednu jedinicu na poziciji poslednje jedinice u binarnom zapisu datog broja) čime bismo dobili

100 što odgovara broju 4, a nakon toga bismo ovu vrednost sabrali sa 100 čime bismo dobili 1000 (binarni zapis broja 8). Ovde se procedura završava s obzirom na to da smo stigli do poslednjeg elementa u Fenvikovom drvetu.

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

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

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

```
// Na osnovu niza a u kom su elementi smešteni
// na pozicijama iz segmenta [1, n] formira Fenvikovo drvo
// i smešta ga u niz drvo (na pozicije iz segmenta [1, n])
```

```
void formirajDrvo(int drvo[], int n, int a[]) {
  fill_n(a+1, n, 0);
  for (int k = 1; k <= n; k++)
     dodaj(drvo, n, k, a[k]);
}
```

Broj inverzija pomoću Fenvikovog drveta

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

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

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

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

Primećujemo da su segmenti čiji nas zbirovi zanimaju stalno sufiksi niza frekvencija. Ovo nas može asocirati na to da bismo obilaskom u suprotnom smeru mogli malo popraviti efikasnost, jer bismo umesto zbirova sufiksa koji zahtevaju dva obilaska izračunavali zbirove prefiksa koji zahtevaju samo jedan obilazak. Zaista, ako niz obrađujemo unatrag i računamo frekvencije svih ranije viđenih elemenata, onda inverzije u kojima učestvuje tekući element i oni elementi iza njega određujemo tako što prebrojimo koliko ima viđenih elemenata koji su strogo manji od tekućeg, a to možemo saznati ako saberemo frekvencije koje se nalaze na pozicijama iz intervala $[1, a_i - 1]$. Na kraju ne smemo zaboraviti da održimo induktivnu hipotezu tako što nakon obrade uvećamo frekvenciju pojavljivanja tekućeg elementa.

Prethodni pristup nije moguć ako se u nizu javljaju negativni elementi, a prilično je memorijski neefikasan ako je maksimalni element niza veliki (memorijska složenost BIT-a je $O(max)$). Nama nisu relevantne same vrednosti, već samo njihov međusobni odnos. Zato pre obrade pomoću BIT-a možemo uraditi takozvanu *kompresiju indeksa* (engl. index compression). Svaki element u nizu

ćemo zameniti njegovom pozicijom u sortiranom redosledu (krenuvši od 1). Iste elemente možemo zameniti istim pozicijama. Najlakši način da se to postigne je da se napravi sortirana kopija niza, i da se zatim za svaki element polaznog niza binarnom pretragom pronađe pozicija njegovog prvog pojavljivanja u sortiranoj kopiji (u jeziku C++ to možemo jednostavno postići bibliotečkim funkcijama `sort` i `lower_bound`).

```
// operacije za rad sa Fenvikovim drvetom

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

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

// izračunava broj parova i < j takvih da je a[i] > a[j]
int brojInverzija(const vector<int>& a) {
  // sortiramo niz
  int n = a.size();
  vector<int> a_sort = a;
  sort(begin(a_sort), end(a_sort));
  // Fenvikovo drvo
  vector<int> drvo(n+1, 0);
  // broj inverzija
  int broj = 0;
  for (int i = n-1; i >= 0; i--) {
    // na osnovu sortiranog niza a,
    // određujemo koji je po veličini u nizu a element a[i] -
    // broji se od 1
    auto it = lower_bound(begin(a_sort), end(a_sort), a[i]);
    int x = distance(begin(a_sort), it) + 1;
    // uvećavamo broj inverzija za broj do sada viđenih elemenata
    // koji su strogo manji od x
    broj += zbirPrefiksa(drvo, x-1);
    // ažuriramo frekvenciju pojavljivanja elementa x
    dodaj(drvo, x, 1);
```

```
  }
  // vraćamo ukupan broj inverzija
  return broj;
}
```

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

Ažuriranje celih raspona niza odjednom

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

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

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

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

Svaki čvor u segmentnom drvetu se odnosi na određeni segment elemenata polaznog niza i čuva zbir tog segmenta. Ako se taj segment u celosti sadrži unutar segmenta koji se ažurira, možemo unapred izračunati za koliko se povećava vrednost u korenu. Naime, pošto se svaka vrednost u segmentu povećava za v, tada se vrednost zbira tog segmenta povećava za $k \cdot v$, gde je k broj elemenata

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

Imajući ovo u vidu, razmotrimo kako se može implementirati funkcija koja vrši uvećanje svih elemenata nekog segmenata. Njena invarijanta će biti da svi čvorovi u drvetu ili sadrže ažurne vrednosti zbirova ili će biti ispravno obeleženi za kasnija ažuriranja (preko koeficijenta lenje propagacije), a da će nakon njenog izvršavanja koren drveta na kom je pozvana sadržati aktuelnu vrednost zbira. Nakon početnog obezbeđivanja da vrednost u tekućem čvoru postane ažurna, moguća su tri sledeća slučaja. Ako je segment u tekućem čvoru disjunktan u odnosu na segment koji se ažurira, tada su svi čvorovi u poddrvetu kojem je on koren već ili ažurni ili ispravno obeleženi za kasnije ažuriranje i nije potrebno ništa uraditi. Ako je segment koji odgovara tekućem čvoru potpuno sadržan u segmentu čiji se elementi uvećavaju, tada se njegova vrednost ažurira (uvećavanjem za $k \cdot v$, gde je k broj elemenata segmenta koji odgovara tekućem čvoru, a v vrednost uvećanja), a njegovoj deci se koeficijent lenje propagacije uvećava za v. Na kraju, ako se dva segmenta seku, tada se prelazi na rekurzivnu obradu dva deteta. Nakon izvršavanja funkcije nad njima, sigurni smo da će svi čvorovi u levom i desnom poddrvetu zadovoljavati uslov invarijante i da

će oba korena imati ažurne vrednosti. Ažurnu vrednost u korenu postizaćemo sabiranjem vrednosti dva deteta.

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

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

```
            int a, int b, int v) {
   promeni(drvo, lenjo, 1, 0, n-1, a, b, v);
}
```

Prikažimo rad ove funkcije na jednom primeru.

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

0      1     2     3     4      5     6     7
```

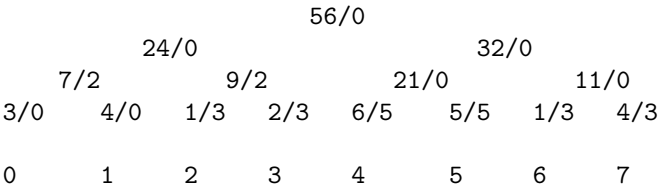
Prikažimo kako bismo sve elemente iz segmenta pozicija $[2, 7]$ uvećali za 3. Segmenti $[0, 7]$ i $[2, 7]$ se seku, pa stoga ažuriranje prepuštamo naslednicima i nakon njihovog ažuriranja, pri povratku iz rekurzije vrednost određujemo kao zbir njihovih ažuriranih vrednosti. Na levoj strani se segment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

```
                    44/0
          16/0                    28/0
    7/0         9/0         11/3         5/3
3/0    4/0   1/3   2/3   6/0    5/0   1/0   4/0

0      1     2     3     4      5     6     7
```

Pretpostavimo da se sada elementi iz segmenta $[0, 5]$ uvećavaju za 2. Ponovo se kreće od vrha i kada se ustanovi da se segment $[0, 7]$ seče sa $[0, 5]$ ažuriranje se prepušta deci i vrednost se ažurira tek pri povratku iz rekurzije. Segment $[0, 3]$ je ceo sadržan u $[0, 5]$, pa se zato vrednost 16 uvećava za $4 \cdot 2 = 8$ i postavlja na 24. Poddrveta se ne ažuriraju odmah, već se samo njihovim korenima upisuje da je sve vrednosti potrebno ažurirati za 2. U desnom poddrvetu segment $[4, 7]$ se seče sa $[0, 5]$, pa se rekurzivno obrađuju poddrveta. Pri obradi čvora 11, primećuje se da je on trebao da bude ažuriran, međutim, još nije, pa se onda njegova vrednost

ažurira i uvećava za $2 \cdot 3$ i sa 11 menja na 17. Njegovi naslednici se ne ažuriraju odmah, već samo ako to bude potrebno i njima se samo upisuje lenja vrednost 3. Tek nakon toga, se primećuje da se segment $[4, 5]$ ceo sadrži u $[0, 5]$, pa se onda vrednost 17 uvećava za $2 \cdot 2 = 4$ i postavlja na 21. Poddrveta se ne ažuriraju odmah, već samo po potrebi tako što se u njihovim korenima postavi vrednost lenjog koeficijenta. Pošto je u njima već upisana vrednost 3, ona se sada uvećava za 2 i postavlja na 5. Prelazi se tada na obradu poddrveta u čijem je korenu vrednost 5 i pošto ono nije ažurno, prvo se ta vrednost 5 uvećava za $2 \cdot 3 = 6$ i postavlja na 11, a njegovoj deci se lenji koeficijent postavlja na 3. Nakon toga se primećuje da je segment $[6, 7]$ disjuktan sa $[0, 5]$ i ne radi se ništa. U povratku kroz rekurziju se ažuriraju vrednosti roditeljskih čvorova i dolazi se do narednog drveta.



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

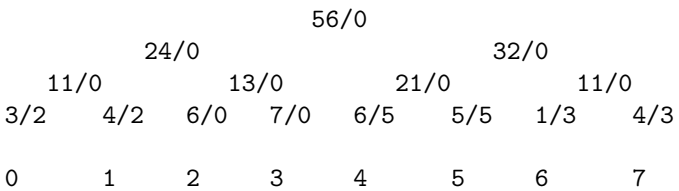
```
// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [x, y]
// izračunava se zbir elemenata polaznog niza sa pozicija
// iz segmenta [a, b]
int saberi(int drvo[], int lenjo[], int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

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

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

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

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

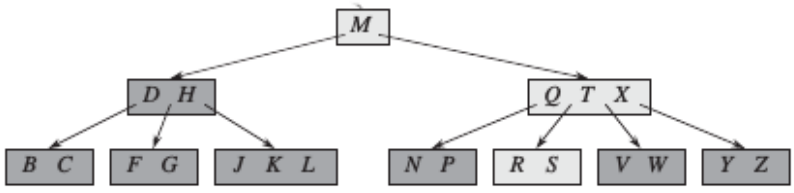


B-drvo

B-drveta su balansirana uređena drveta. U većini drugih balansiranih uređenih drveta kao što su recimo AVL drveta ili crveno-crna drveta pretpostavlja se da se svi podaci nalaze u glavnoj memoriji. B-drveta uskaču u pomoć u situaciji kada se čuva ogromna količina podataka koja ne može da stane u glavnu memoriju. Na primer, mnogi sistemi baza podataka koriste B-drveta za čuvanje podataka. B-drveta su dizajnirana tako da rade dobro na diskovima i drugim sekundarnim uređajima za skladištenje sa direktnim pristupom, čiji je kapacitet obično za red veličine veći. Međutim, vreme pristupa disku je dosta veće u odnosu na vreme pristupa glavnoj memoriji. Iz tog razloga, kada se podaci čitaju sa diska, obično se ne pristupa samo jednom podatku već većem broju podataka istovremeno. Podaci se dele na blokove jednake veličine i svako čitanje/pisanje diska pristupa jednom ili većem broju blokova (stranica). Osnovna ideja B-drveta je da se smanji broj pristupa disku, da bi se ubrzao rad sa podacima.

Osnovna razlika između B-drveta i crveno-crnih drveta je ta što čvor B-drveta može imati veći broj dece, čak i do nekoliko hiljada. Kao i kod crveno-crnih drveta svaki čvor u drvetu je visine $O(\log n)$, ali tačna visina B-drveta je značajno manja nego kod crveno-crnih drveta jer je faktor grananja, a samim tim i osnova logaritma kojom se izražava visina drveta, mnogo veći. Dakle, visina B-drveta se održava malom tako što se u jednom čvoru čuva veliki broj ključeva. Najčešće je čvor B-drveta veliki kao cela stranica na disku (koja se može sačuvati u glavnoj memoriji), te u stvari veličina stranice ograničava broj dece koji čvor B-drveta može da ima. S obzirom na to da u glavnoj memoriji možemo čuvati koren B-drveta, svaki ključ u drvetu možemo pronaći nakon malog broja pristupa disku.

Ukoliko unutrašnji čvor B-drveta čuva vrednosti n ključeva, onda on ima $n + 1$ dece. Vrednosti ključeva u unutrašnjem čvoru predstavljaju tačke podele opsega mogućih ključeva u $n + 1$ podopsega, pri čemu će svaki od tih podopsega biti obrađen od strane jednog deteta tog čvora. Prilikom potrage za ključem u B-drvetu, odluku gde dalje nastaviti potragu pravimo na osnovu poređenja sa n ključeva smeštenih u tom čvoru.



Slika 1: Primer B-drveta u kom su vrednosti ključeva slova engleske abecede.

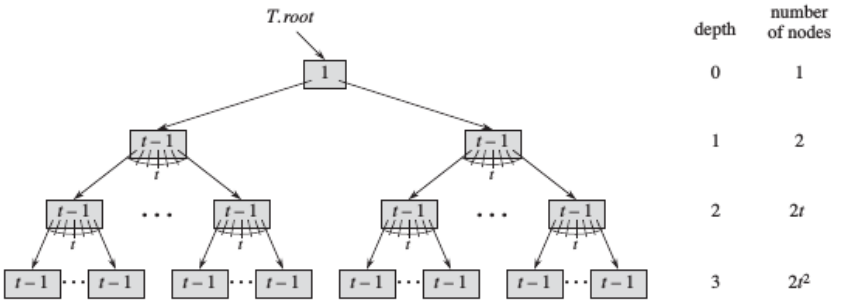
Pretpostavićemo da se satelitski podaci uz svaki ključ čuvaju u istom čvoru zajedno sa ključem. U praksi, moguće je da se uz ključ čuva samo pokazivač na drugu stranicu na disku koja sadrži satelitske podatke vezane za taj ključ.

B-drvo zadovoljava naredna svojstva:

1. Svaki čvor sadrži broj ključeva n koji se čuvaju u tom čvoru, n ključeva $k_1, k_2, \ldots, k_n$ koji se čuvaju u nerastućem poretku (važi $k_1 \leq k_2 \leq \ldots \leq k_n$) i oznaku da li je taj čvor list
2. Svaki unutrašnji čvor sadrži i $n+1$ pokazivača $c_1, c_2, \ldots, c_{n+1}$ ka svojoj deci
3. Ključevi k_i koji se čuvaju u datom čvoru razdvajaju opseg ključeva koji se čuvaju u poddrvetu čiji je koren dati čvor
4. Svi listovi se nalaze na istom nivou
5. Koren ima bar dva deteta, ukoliko nije list
6. Čvorovi imaju donju i gornju granicu na broj ključeva koji sadrže. Za svako B-drvo definišemo *minimalni stepen* t B-drveta i važi sledeće:
   1. Svaki čvor osim korena mora da ima bar $t-1$ ključeva, odnosno bar t dece. Ako je drvo neprazno, koren mora da sadrži bar jedan ključ.
   2. Svaki čvor može da sadrži najviše $2t-1$ ključeva, odnosno maksimalno $2t$ dece. Kažemo da je čvor pun ukoliko sadrži tačno $2t-1$ ključeva.

Najjednostavnije B-drvo dobijamo za vrednost $t = 2$; tada svaki unutrašnji čvor ima 2, 3 ili 4 deteta. Jasno je da veće vrednosti za t daju B-drveta manje visine.

Važi sledeće tvrđenje: za svako B-drvo T visine h minimalnog stepena $t \geq 2$ koje sadrži n ključeva važi da je $h \leq \log_t \frac{n+1}{2}$. Dokažimo ga.



Slika 2: B-drvo visine 3 koje sadrži najmanji mogući broj ključeva.

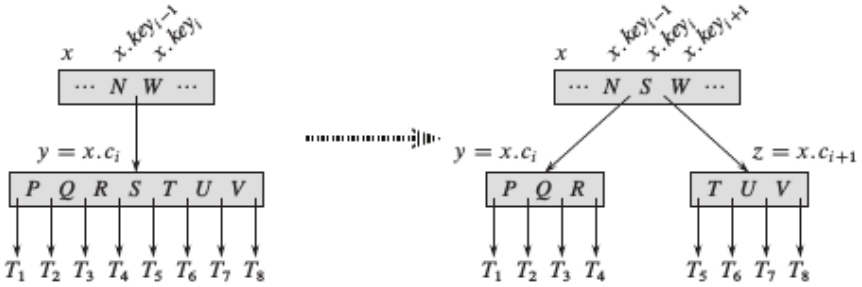
Koren B-drveta T sadrži bar jedan ključ, a svi ostali čvorovi bar $t-1$ ključeva. Dakle drvo T visine h sadrži bar 2 čvora dubine 1, bar $2t$ čvorova dubine 2, bar $2t^2$ čvorova dubine 3 itd sve do dubine h gde imamo bar $2t^{h-1}$ čvorova. Na slici 2 prikazano je ovakvo drvo za $h = 3$. Ako sa n označimo broj ključeva u drvetu, važi nejednakost:

$$n \geq 1 + (t-1)\sum_{i=0}^{h-1} 2t^i = 1 + 2(t-1)\frac{t^h-1}{t-1} = 2t^h - 1$$

Odavde dobijamo $t^h \leq (n+1)/2$ odakle direktno sledi tvrđenje.

Pretraga u B-drvetu je nalik pretrazi u uređenom binarnom drvetu, osim što umesto pravljenja binarne odluke u svakom čvoru, imamo višestruko grananje na osnovu broja dece tog čvora.

Umetanje u B-drvo je znatno komplikovanije: kao i kod uređenih binarnih drveta, tražimo list u koji bi trebalo da unesemo novu vrednost ključa. Međutim, kod B-drveta ne možemo samo kreirati novi list i umetnuti ga jer rezultujuće drvo neće biti B-drvo (zbog ograničenja na minimalni broj ključeva koji se čuvaju u svakom čvoru). Stoga, ključ umećemo u postojeći list. S obzirom na to da ne možemo umetnuti ključ u puni čvor, nekad je potrebno podeliti puni čvor (sa $2t-1$ ključeva) oko njegove medijane u dva čvora koja imaju po $t-1$ čvorova. Ključ medijane se penje u roditeljski čvor da bi predstavljao tačku podele između dva nova drveta. Međutim, ukoliko je njegov roditelj isto pun čvor, moramo ga podeliti pre umetanja novog ključa, i na ovaj način je moguće vršiti podelu duž cele putanje do korena drveta. Međutim, operaciju umetanja je moguće izvršiti i u jednom prolasku kroz drvo od korena do lista tako što dok tražimo poziciju umetanja, delimo svaki puni čvor na koji naiđemo. Stoga, uvek kada treba da podelimo neki čvor, sigurni smo da njegov otac nije takođe puni čvor.



Slika 3: Podela čvora B-drveta za koje važi $t = 4$.

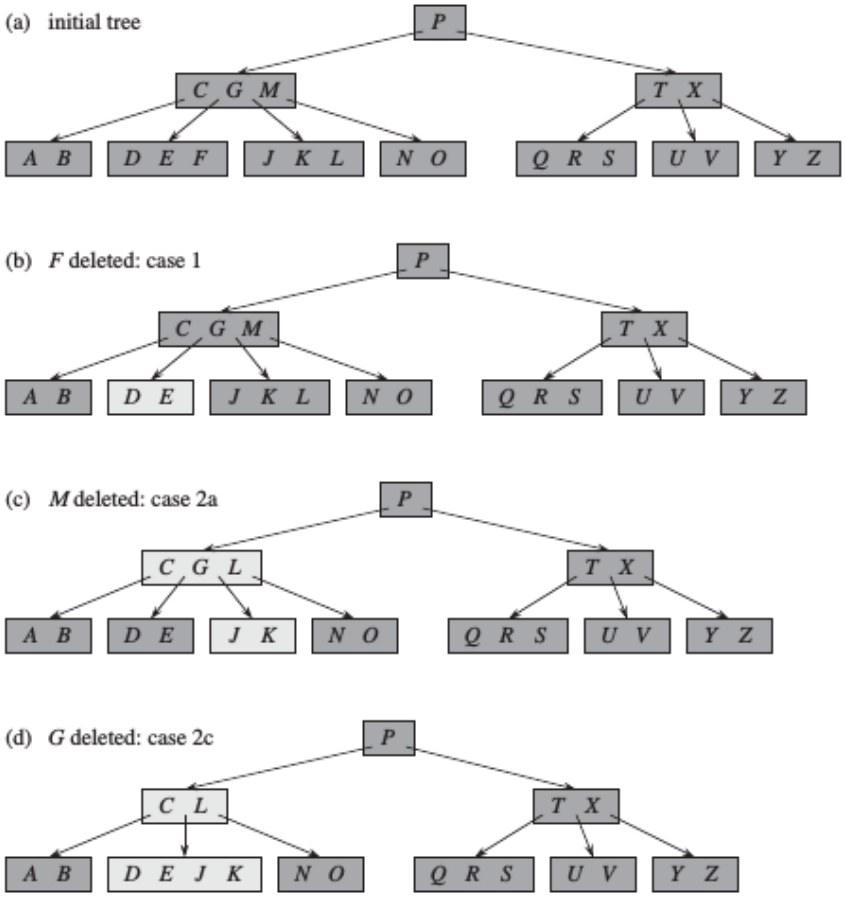
Brisanje čvora iz B-drveta je nalik umetanju, ali donekle komplikovanije jer se ključ može brisati iz svakog od čvorova, a ne nužno iz lista drveta. U situaciji kada brišemo ključ iz unutrašnjeg čvora, potrebno je da preuredimo decu tog čvora. Slično kao i kod umetanja, moramo voditi računa da brisanjem ne narušimo neki od uslova B-drveta, odnosno da čvor ne ostane sa premalim brojem ključeva. Brisanje ključa sa vrednošću k iz datog B-drveta se sprovodi na sledeći način:

1. ako je ključ k u čvoru x i x je list, obrisati ključ k iz čvora x
2. ako je ključ k u čvoru x i x je unutrašnji čvor, raditi sledeće:
   a. ako dete y čvora x koje prethodi ključu k sadrži barem t ključeva, pronaći prethodnika k' ključa k u poddrvetu sa korenom y. Rekurzivno obrisati k' i zameniti k sa k'.
   b. ako y sadrži manje od t ključeva, onda simetrično ispitati dete z čvora

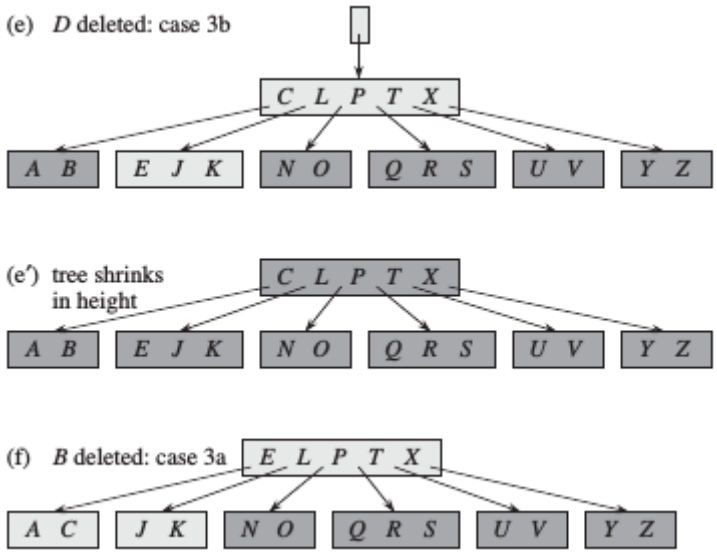
x koje sledi nakon ključa k u čvoru x. Ako z ima bar t ključeva, pronađi sledbenika k' ključa k u poddrvetu sa korenom z. Rekurzivno obrisati k' i zameniti k sa k'.

   c. ako i y i z imaju samo $t-1$ ključeva, spajamo ključ k i sve ključeve iz čvora z u čvoru y, tako da x izgubi i k i pokazivač na z, a y sada sadrži $2t-1$ ključeva. Nakon toga oslobađamo z i rekurzivno brišemo k iz y.

3. ako se ključ k ne nalazi u tekućem unutrašnjem čvoru x, tražimo koren odgovarajućeg poddrveta koje mora da sadrži k, ako se k uopšte nalazi u drvetu. Ako koren tog poddrveta sadrži samo $t-1$ čvorova, izvršavamo jedan od narednih koraka da bismo postigli da se spuštamo u čvor sa bar t ključeva, a onda rekurzivno nastavljamo sa odgovarajućim sinom čvora x
   a. ako odgovarajuće dete čvora x sadrži samo $t-1$ ključeva, ali ima direktnog brata sa bar t ključeva, dodajemo mu još jedan ključ tako što ključ iz čvora x spuštamo u to dete, jedan od ključeva direktnog brata stavljamo umesto ključa u čvor x i pomeramo odgovarajući pokazivač sa brata na njega
   b. ako i odgovarajuće dete čvora x i njegov brat sadrže po $t-1$ ključeva, spajamo ih i pomeramo ključ iz x u novi spojeni čvor da bi predstavljao ključ medijane za taj čvor.



Slika 4: Ilustracija slučajeva 1, 2a i 2c prilikom brisanja čvorova iz B-drveta. Minimalni stepen ovog drveta je $t = 3$, te čvor ne može imati manje od 2 ključa.



Slika 5: Ilustracija slučajeva 3b i 3a prilikom brisanja čvorova iz B-drveta.

Fibonačijev hip

Fibonačijev hip je struktura podataka kod koje operacije umetanja novog elementa u strukturu i izdvajanja minimalnog elementa iz strukture imaju konstantnu amortizovanu složenosti, te predstavlja pogodnu strukturu podataka u situacijama kada su ove operacije česte.

Preciznije, operacije koje se korišćenjem Fibonačijevog hipa mogu efikasno izvršiti jesu:

- `make_heap()` - pravi novi prazni hip
- `insert(H,x)` - umeće element x u Fibonačijev hip H
- `minimum(H)` - vraća element Fibonačijevog hipa H sa minimalnom vrednošću ključa
- `extract-min(H)` - briše element Fibonačijevog hipa H sa minimalnom vrednošću ključa i vraća ga kao povratnu vrednost
- `union(H1,H2)` - od dva Fibonačijeva hipa $H1$ i $H2$ pravi novi Fibonačijev hip koji sadrži sve elemente polaznih hipova
- `decrease_key(H,x,k)` - elementu x Fibonačijevog hipa H menja vrednost ključa novom vrednošću k, pritom je nova vrednost ključa manja ili jednaka staroj

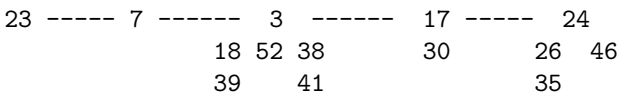
Amortizovana složenost operacija pravljenja hipa, umetanja elementa, određivanja minimuma, pravljenja unije i smanjivanja vrednosti ključa je $O(1)$, dok je amortizovana složenost operacije brisanja minimalnog elementa iz hipa $O(\log n)$. Dakle, operacije umetanja elementa, pravljenja unije i smanjivanja vrednosti ključa se efikasnije izvršavaju nad Fibonačijevim hipom, nego nad klasičnim binarnim hipom.

Fibonačijev hip je koristan kada je broj operacija brisanja minimalnog elementa iz hipa mali u odnosu na ostale pomenute operacije. Na primer, u grafovskim algoritmima kao što je određivanje razapinjućeg (povezujućeg) drveta minimalne cene ili najkraćih puteva iz zadatog čvora, ako je graf gust (sadrži veliki broj grana), operacija smanjivanja vrednosti ključa se može često javljati, te ubrzanje sa $O(\log n)$ kod binarnog hipa na $O(1)$ kod Fibonačijevog hipa može doneti pristojno ubrzanje. Nedostatak Fibonačijevog hipa je to što je u odnosu na veoma jednostavnu implementaciju binarnog hipa dosta teži za implementaciju.

Fibonačijev hip je dobio naziv prema Fibonačijevim brojevima koji se koriste u analizi složenosti operacija. Takođe, važi da je veličina svakog poddrveta Fibonačijevog hipa čiji je koren stepena k bar F_{k+2}, gde je sa F_k označen k-ti Fibonačijev broj. Ovu strukturu podataka osmislili su Fredman i Tardžan sa ciljem da se poboljša vreme izvršavanja Dajkstrinog algoritma za određivanje najkraćih puteva iz zadatog čvora. Dajkstrin algoritam koji koristi binarni hip radi u vremenskoj složenosti $O((|E| + |V|) \log |V|)$, a složenost algoritma koji koristi Fibonačijev hip umesto binarnog hipa je $O(|E| + |V| \log |V|)$.

Fibonačijev hip predstavlja kolekciju min-hipova. Koreni svih drveta u Fibonačijevom hipu se povezuju u kružnu, dvostruko povezanu listu koju nazivamo *lista korenova* hipa. Redosled drveta u Fibonačijevom hipu je proizvoljan. Fibonačijevom hipu pristupamo putem pokazivača na koren drveta sa minimalnom vrednošću – ovaj čvor zovemo *minimalnim čvorom* Fibonačijevog hipa (ukoliko postoji više čvorova sa minimalnom vrednošću bilo koji od njih se može proglasiti minimalnim čvorom). Svaki čvor sadrži pokazivač na svog roditelja i na jednog sina. Svi sinovi jednog čvora su međusobno povezani u kružnu, dvostruko povezanu listu, što omogućava umetanje i brisanje čvora u vremenu $O(1)$. Druga pogodnost korišćenja dvostruko povezane kružne liste je ta što dve ovakve liste možemo objediniti u novu listu takođe u vremenu $O(1)$.

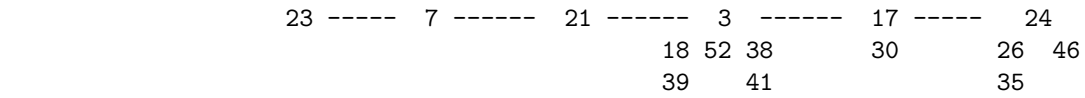
Na narednoj slici prikazan je Fibonačijev hip koji se sastoji od pet min-hipova. Isprekidanim linijama označena je lista korenova. Minimalni element u hipu je minimum svih korenova, a to je element sa ključem 3.



Svaki čvor sadrži i informaciju o broju svoje dece, kao i oznaku da li je "izgubio" dete od poslednjeg trenutka kada je postao sin nekog drugog čvora. Čvorovi se inicijalno ne označavaju i svaki put kada čvor postane sin nekog drugog čvora, ukoliko je bio označen, oznaka se uklanja.

Jedna od osnovnih karakteristika operacija nad Fibonačijevim hipom je lenjo izvršavanje operacija, tj posao se odlaže za što je moguće kasnije. Na primer, kada se vrši umetanje novog elementa u Fibonačijev hip, on se dodaje u listu korenova, što je konstantne vremenske složenosti. Dakle, umetanjem k elemenata u prazan Fibonačijev hip dobija se Fibonačijev hip čija lista korenova ima k elemenata. Ukoliko je vrednost ključa umetnutog elementa manja od vrednosti ključa tekućeg minimalnog elementa hipa, vrši se ažuriranje.

Na primer, umetanjem elementa sa ključem 21 u prethodni Fibonačijev hip dobio bi se naredni hip:



Određivanje minimalnog elementa u Fibonačijevom hipu je trivijalno, s obzirom na to da se u svakom trenutku čuva referenca na minimalni element u strukturi.

Unija dva Fibonačijeva hipa H_1 i H_2 se pravi nadovezivanjem listi korenova ova dva hipa i određivanjem novog minimalnog čvora kao manjeg od minimuma dva polazna hipa.

Operacija brisanja najmanjeg elementa iz Fibonačijevog hipa se izvodi na sledeći način: prvo se sva njegova deca umeću u listu korenova, on se briše iz liste korenova i vrši se ažuriranje minimuma hipa. Zatim se vrši *konsolidacija* hipa, odnosno hip se dovodi u stanje u kom nijedna dva korena nemaju isti stepen. To se postiže na sledeći način: sve dok neka dva čvora iz liste korenova imaju isti sstepen radi se sledeće:

1. pronalaze se dva čvora x i y u listi korenova koja su istog stepena (bez smanjenja opštosti neka važi da je vrednost ključa čvora x manja ili jednaka od vrednosti ključa čvora y)
2. y se izbacuje iz liste korenova i proglašava se detetom čvora x. Pritom, ukoliko je čvor y bio označen, briše se njegova oznaka. Na ovaj način drvo sa korenom x ostaje hip i stepen čvora x se povećava za jedan.

Za realizovanje ove operacije potreban je pomoćni niz u koji ćemo smeštati pokazivače na čvorove iz liste korenova, tako da se na poziciji i čuva pokazivač na prvi čvor iz liste korenova koji je stepena i. Dimenzija ovog pomoćnog niza jednaka je maksimalnom stepenu $D(n)$ proizvoljnog čvora u Fibonačijevom hipu – može se pokazati da je on jednak $O(\log n)$, gde je sa n označen broj elemenata hipa. Koreni koji su istog stepena se detektuju na sledeći način: prolazi se redom kroz listu korenova i ukoliko je tekući koren stepena i, a i-ti element niza je još uvek prazan, on se inicijalizuje pokazivačem na dati koren, a ukoliko je i-ti element već postavljen, otkriven je konflikt.

Pogledajmo kako bi se izmenio prethodni hip nakon izvođenja operacije brisanja elementa sa minimalnom vrednošću ključa.

```
                                H.min
                                  |
    23 ----- 7 ------  21 ------  3  ------  17 -----  24
                             18 52 38       30      26  46
                             39    41               35
```

```
    23 ----- 7 ------ 21 ----- 18 --- 52 --- 38 ---- 17 -----   24
                              39            41      30       26  46
                                                             35
```

17 je stepena 1,
24 je stepena 2,
23 je stepena 0,
7 je isto stepena 0, pa se 23 postavlja za dete cvora 7

```
  7 ------ 21 ----- 18 --- 52 --- 38 ---- 17 ----- 24
 23                39            41      30     26  46
                                                 35
```

7 je sada stepena 1,
sada 17 i 7 imaju isti stepen, pa se 17 postavlja za dete cvora 7

```
  7 ------ 21 ----- 18 --- 52 --- 38 ------  24
17 23               39            41       26  46
30                                          35
```

7 je sada stepena 2
sada 24 i 7 imaju isti stepen, pa se 24 postavlja za dete cvora 7

```
     7  ------ 21 ----- 18 ---- 52 ---- 38
 24   17  23             39              41
26 46  30
35
```

sada 21 i 52 imaju isti stepen, pa se 52 postavlja za dete cvora 21

```
     7   ------ 21 ----- 18 ---- 38
 24  17  23      52       39      41
26 46 30
35
```

```
sada 21 i 18 imaju isti stepen, pa se 21 postavlja za dete cvora 18

H.min
  |
  7   -----  18  ---- 38
 24  17  23    21  39     41
26 46 30        52
35
```

Umetanje sinova minimalnog čvora u listu korenova je složenosti $O(D(n))$, ažuriranje vrednosti minimuma je složenosti $O(D(n) + t(H))$, gde je sa $t(H)$ označen broj korenova Fibonačijevog hipa, a konsolidacija je složenosti $O(D(n) + t(H))$. Dakle, složenost ove operacije je $O(D(n) + t(H))$.

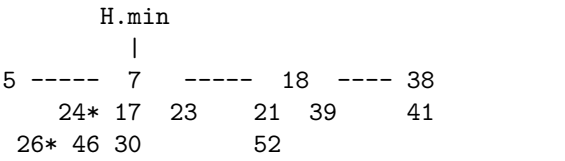
Još jedna važna operacija nad Fibonačijevim hipom jeste smanjivanje vrednosti ključa elementu x. Ona se izvodi na sledeći način: prvo se datom elementu x promeni vrednost ključa. Ukoliko je x koren ili je vrednost ključa roditelja čvora x manja od nove vrednosti ključa čvora x, onda uslov hipa nije narušen te nisu potrebne nikakve dalje izmene. Ukoliko ovo nije slučaj, uslov hipa je narušen te je hip potrebno preurediti. Najpre se vrši *odsecanje* grane između čvora x i njegovog roditelja i poddrvo sa korenom x se dodaje u listu korenova. Označavanje čvorova je važno za ovu operaciju: naime, "reagujemo" ako jedan čvor izgubi dva sina tako što vršimo odsecanje i roditelja. Preciznije vrši se sledeći niz koraka:

- smanjuje se vrednost ključa čvora x
- odseca se drvo čiji je x koren, dodaje se u listu korenova i uklanja se oznaka (ukoliko je čvor bio označen)
- ako je roditelj p čvora x neoznačen (nije mu do sada bio odsečen nijedan sin), on se označava
- u protivnom, odseca se i roditeljski čvor p, dodaje se u listu korenova i sa njega se uklanja oznaka
- rekurzivno se ponavlja prethodni korak za sve roditelje kojima su dvaput odsečeni sinovi

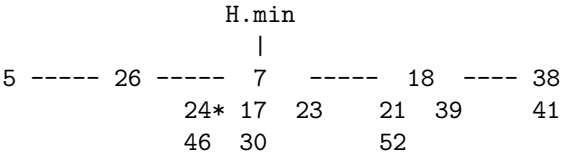
Na primer, ukoliko u datom hipu (u kome su označeni elementi označeni simbolom *) želimo da smanjimo vrednost ključa sa 35 na 5, to povlači još dva dodatna odsecanja:

```
      H.min
        |
        7   -----  18  ---- 38
     24* 17  23    21  39     41
  26* 46 30        52
  35

  35 se menja u 5, vrsi se odsecanje i dodaje se u listu korenova
```



26 je bio oznacen i odseceno mu je dete, te se i on odseca i dodaje u listu korenova



24 je bio oznacen i odseceno mu je dete, te se i on odseca i dodaje u listu korenova
novi minimum je ili stari minimum ili nova (smanjena) vrednost cvora