Fenvikova drveta (BIT)

U nastavku ćemo razmotriti *Fenvikova drveta* tj. *binarno indeksirana drveta* (eng. 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 Fenvikovih drveta, ali ne i minimumi, nzd-ovi i slično). Dakle, 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 Sradi 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 $[1, n]$.

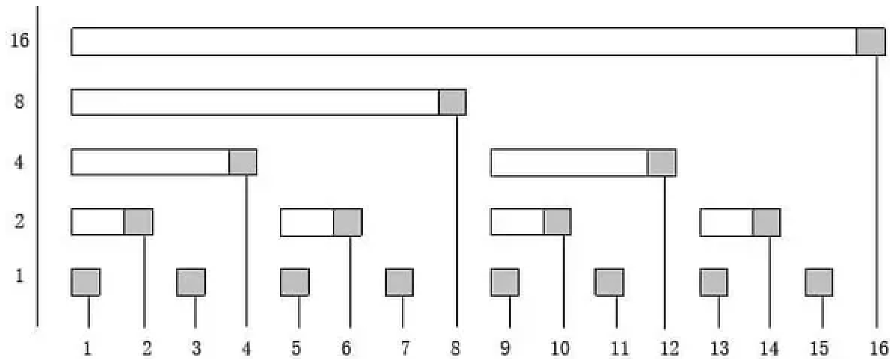
Ključna ideja Fenvikovog drveta je sledeća: *u drvetu se na poziciji k čuva zbir vrednosti polaznog niza iz segmenta pozicija oblika $(f(k), k]$ gde je $f(k)$ broj koji se dobije od broja k tako što se iz binarnog zapisa broja k obriše prva jedinica zdesna.*

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 najdesnije jedinice iz njegovog binarnog zapisa 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 krajnje desne jedinice iz njeovog binarnog zapisa 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 najdesnije jedinice iz binarnog zapisa broja 16 dobija 0 (važi da je $f(16) = 0$).

Na slici 1 je za niz dužine 16 prikazano koje se sume segmenata čuvaju na svakoj od pozicija. Primetimo da se neparne pozicije odgovaraju segmentima dužine 1, a da za stepene broja 2 segmenti predstavljaju prefikse niza do te pozicije.

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
```



Slika 1: Prikaz segmenata čije su sume smeštene u elementima Fenvikovog drveta dužine 16.

```
    0      0     2     0     4     4     6     0  f(k) dekadno
(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  Fenvikovo 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, a ne samo za broj 21. Broj elemenata čijim se sabiranjem dobija zbir prefiksa je samo $O(\log n)$. Naime, u svakom koraku se broj jedinica u binarnom zapisu tekućeg indeksa smanjuje, a broj n se binarno 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 vrednost $f(k)$. 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. Na primer, broju $k = 20$ odgovara binarni zapis 00010100, a broju $-k$ binarni zapis 11101100, te će `k & -k` biti jednako 00000100 i predstavlja broj koji sadrži samo jednu jedinicu i to na mestu poslednje jedinice u zapisu broja k. Oduzimanjem ove vrednosti od broja k dobijamo broj 16 kome odgovara binarna reprezentacija 00010000.

Drugi način da se izračuna vrednost $f(k)$ jeste računanjem vrednosti izraza `k & (k-1)`. Naime, broj `k-1` će imati sve iste bitove od najlevijeg do pozicije poslednjog postavljenog bita u broju k, a sve invertovane bitove posle najdesnijeg postavljenog bita u broju `k`. Na primer, binarni zapis broja $k = 20$ je 00010100, a broja $k-1 = 19$ je 00010011 i izrazu `k & (k-1)` odgovara binarni zapis 00010000.

Zbir prefiksa $[0, k]$ polaznog niza a smeštenog u Fenvikovo drvo *drvo* 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, pošto se radi o operaciji sabiranja, da bismo mogli da oduzimanjem dve vrednosti prefiksa dobijemo zbir proizvoljnog prefiksa[1]).

Osnovna prednost Fenvikovih drveta u odnosu na niz svih zbirova prefiksa je to što se mogu efikasno ažurirati. Naime, ažuriranje jednog elementa polaznog niza može rezultovati izmenom vrednosti velikog broja prefiksnih suma, u najgorem slučaju $O(n)$ elemenata ako je element koji menjamo blizu početka niza. Stoga je operacija ažuriranja elementa kada se koriste prefiksne sume u najgorem slučaju složenosti $O(n)$. Razmotrimo funkciju koja ažurira Fenvikovo 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, nakon toga sabiranjem sa 1 (brojem koji sadrži tačno jednu jedinicu na poziciji poslednje jedinice u binarnom zapisu datog broja) dobili bismo 100 što odgovara broju 4, a nakon toga bismo ovu vrednost sabrali sa 100 (ponovo brojem koji sadrži tačno jednu jedinicu u svom binarnom zapisu i to na poziciji poslednje jedinice) 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. S obzirom na to da se broj n binarno zapisuje sa $O(\log n)$ bitova i da se u svakom koraku broj krajnjih nula u zapisu broja povećava, ukupan broj koraka biće jednak $O(\log n)$. Dakle, broj elemenata Fenvikovog

[1] Obratimo pažnju da ako se Fenvikovim drvetom računaju proizvodi segmenata, elementi drveta moraju biti različiti od nule, zbog operacije deljenja.

drveta čije je vrednosti potrebno izmeniti iznosi $O(\log n)$.

```
// ažurira Fenvikovo drvo smešteno u niz drvo nakon što se
// u originalnom nizu element na poziciji k uveća za x
void uvecaj(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 po definiciji ne 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 sve brojeve $m > k$, sigurno važi desna nejednakost i potrebno je utvrditi da li važi leva. Razmotrimo koji je to najmanji broj strogo veći od k za koje važi ova nejednakost. Neka je $g(k)$ broj koji se dobija od broja 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 $g(k)$. Pokažimo najpre da $g(k)$ zadovoljava ovaj uslov. 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. Pokažimo sada da je $g(k)$ najmanji broj strogo veći od k koji zadovoljava dati uslov. 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. U primeru za $k = 101100$, kao što smo već videli važi $g(k) = 110000$ i jedini brojevi između k i $g(k)$ su 101101, 101110 i 101111 i brisanjem poslednje jedinice dobijaju se redom brojevi 101100, 101100 i 101110 i svi oni su veći ili jednaki od 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 po vrednosti 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 jednake n, pa je naša implementacija korektna.

Ostaje još pitanje kako u startu formirati Fenvikovo drvo. 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(drvo + 1, n, 0);
  for (int k = 1; k <= n; k++)
     uvecaj(drvo, n, k, a[k]);
}
```

Razmotrimo formiranje niza $3, 4, 1, 2, 6, 5, 1, 4$.

```
0  1  2  3  4  5  6  7  8
0  0  0  0  0  0  0  0  0

uvecaj(1,3)

0  1  2  3  4  5  6  7  8
0  3  3  0  3  0  0  0  3

uvecaj(2,4)

0  1  2  3  4  5  6  7  8
0  3  7  0  7  0  0  0  7

uvecaj(3,1)

0  1  2  3  4  5  6  7  8
0  3  7  1  8  0  0  0  8

uvecaj(4,2)

0  1  2  3   4  5  6  7   8
0  3  7  1  10  0  0  0  10

uvecaj(5,6)

0  1  2  3   4  5  6  7   8
0  3  7  1  10  6  6  0  16

uvecaj(6,5)

0  1  2  3   4  5   6  7   8
0  3  7  1  10  6  11  0  21
```

```
uvecaj(7,1)

0  1  2  3   4  5   6  7   8
0  3  7  1  10  6  11  1  22

uvecaj(8,4)

0  1  2  3   4  5   6  7   8
0  3  7  1  10  6  11  1  26
```

Ova implementacija n puta poziva funkciju `uvecaj` koja je složenosti $O(\log n)$, te je ukupna složenost ovog algoritma $O(n \log n)$. Međutim, pokazuje se da možemo i bolje od ovog. Naime, svaki element Fenvikovog drveta sadrži sumu nekog segmenta polaznog niza, pa za izračunavanje elemenata Fenvikovog drveta možemo iskoristiti sume prefiksa. Izračunavanje suma svih prefiksa polaznog niza je vremenske složenosti $O(n)$, ali se nakon toga elementi Fenvikovog drveta računaju u vremenu $O(1)$, pa je ukupna vremenska složenost ovog pristupa $O(n)$. Naime, element na poziciji k u Fenvikovom drvetu dobijamo kao razliku sume prefiksa $(0, k]$ i sume prefiksa $(0, f(k)]$, ako je $f(k) > 0$, dok ako je $f(k) = 0$ element na poziciji k biće jednak baš sumi prefiksa $(0, k]$.

```
void formirajDrvoPrefiksneSume(int drvo[], int n, int a[]) {
  vector<int> sume_prefiksa(n+1,0);
  sume_prefiksa[1] = a[1];
  for (int k = 2; k <= n; k++){
    sume_prefiksa[k] = sume_prefiksa[k-1] + a[k];
  }

  fill_n(drvo + 1, n, 0);
  for (int k = 1; k <= n; k++){
    // racunamo f(k)
    int f_k = k - (k & -k);
    // od sume prefiksa do k oduzimamo sumu prefiksa do f(k)
    // ako je f(k)=0, onda ne vrsimo oduzimanje
    drvo[k] = sume_prefiksa[k] - (f_k > 0 ? sume_prefiksa[f_k] : 0);
  }
}
```

Prethodna implementacija zahteva dodatni memorijski prostor veličine $O(n)$ za smeštanje suma prefiksa. Primetimo da su za izračunavanje k-tog elementa Fenvikovog drveta potrebni samo elementi niza prefiksnih suma na pozicijama j, gde je $j \leq k$. Ovo nam omogućava da koristimo samo jedan niz koji ćemo inicijalizovati na niz prefiksnih suma, a zatim počev od poslednjeg elementa niza idući ka prvom elementu menjati element po element sa prefiksne sume na element Fenvikovog drveta.

```
void formirajDrvoLinearno(int drvo[], int n, int a[]) {
  fill_n(drvo+1, n, 0);
```

```
  drvo[1] = a[1];
  for (int k = 2; k <= n; k++){
    drvo[k] = drvo[k-1] + a[k];
  }

  for (int k = n; k >= 1; k--){
    int f_k = k - (k & -k);
    if (f_k > 0) drvo[k] -= drvo[f_k];
  }
}
```

Ovaj algoritam za formiranje Fenvikovog drveta dakle ima vremensku složenost $O(n)$, a prostornu složenost $O(1)$.

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* (eng. 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 uvecaj(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
    uvecaj(drvo, x, 1);
  }
  // vraćamo ukupan broj inverzija
  return broj;
}
```

Složenost sortiranja je $O(n \log n)$. 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 održavanjem dva Fenvikova drveta (koje ovde nećemo razmatrati).

Ako treba ažurirati sve elemente polaznog niza, to zahteva ažuriranje vrednosti

svih elemenata segmentnog stabla (kojih ima $O(n)$), i ne bi se moglo uraditi u vremenu manjem od $O(n)$. Međutim, moguće je ažurirati sve elemente proizvoljnog segmenta u vremenu $O(\log n)$ ako se primeni tehnika *lenje propagacije* (eng. lazy propagation). Ona odgovara strategiji lenjog izvršavanja kojim se izračunavanja odlažu sve dok odgovarajuće vrednosti nisu potrebne. 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 tom čvoru. Naime, ako 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 tom čvoru time biva ažurirana u konstantnom vremenu, ali vrednosti zbirova unutar poddrveta kojima je taj čvor koren (uključujući i vrednosti u listovima koje odgovaraju vrednostima polaznog niza) i dalje ostaju neažurne. Njihovo ažuriranje zahtevalo bi linearno vreme, što je nedopustivo skupo. 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. Naime, 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 ih treba 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 tekućeg čvora različita od nule i ako jeste, ažuriraćemo vrednost zbira u tom čvoru tako što ćemo ga uvećati za proizvod tog koeficijenta i broja elemenata koji taj čvor pokriva, 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 na ovaj način 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 vrednost v. Na kraju, ako se ova dva segmenta seku, tada se prelazi na rekurzivnu obradu oba deteta. Nakon izvršavanja funkcije nad njima, sigurni smo da će svi čvorovi u levom i desnom poddrvetu zadovoljavati uslov invarijante i da će koreni i levog i desnog poddrveta imati ažurne vrednosti. Ažurnu vrednost u korenu dobijamo sabiranjem vrednosti dva deteta.

Lenjo segmentno drvo čuvaćemo korišćenjem dva niza: `drvo` i `lenjo`: u prvom ćemo čuvati elemente segmentnog drveta, a u drugom koeficijente lenje propagacije svakog od čvorova.

```
// 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];
      }
      // vrednost u korenu je sad ažurna
      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]
   // vrsimo uvecanje cvora za k * v
   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;
```

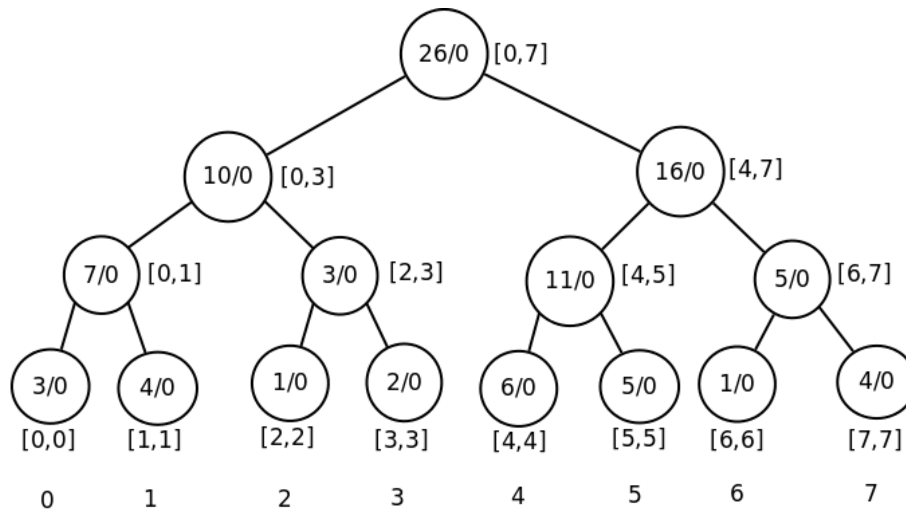
```
      }
   } else {
      // u suprotnom se intervali seku,
      // pa rekurzivno obilazimo poddrveta
      int s = (x + y) / 2;
      promeni(drvo, lenjo, 2*k, x, s, a, b, v);
      promeni(drvo, lenjo, 2*k+1, s+1, y, a, b, v);
      // azurnu vrednost u korenu dobijamo kao zbir vrednosti dece
      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);
}
```

Složenost funkcije ažuriranja svih elemenata datog segmenta u lenjom segmentnom drvetu iznosi $O(\log n)$. Naime, kao i u slučaju operacije računanja sume segmenta u segmentnom drvetu, na svakom nivou se razmatra najviše 4 čvora, te je maksimalni broj čvorova koji se posećuju jednak $4 \log n$.

Prikažimo rad ove funkcije na primeru lenjog segmentnog drveta sa slike 2.

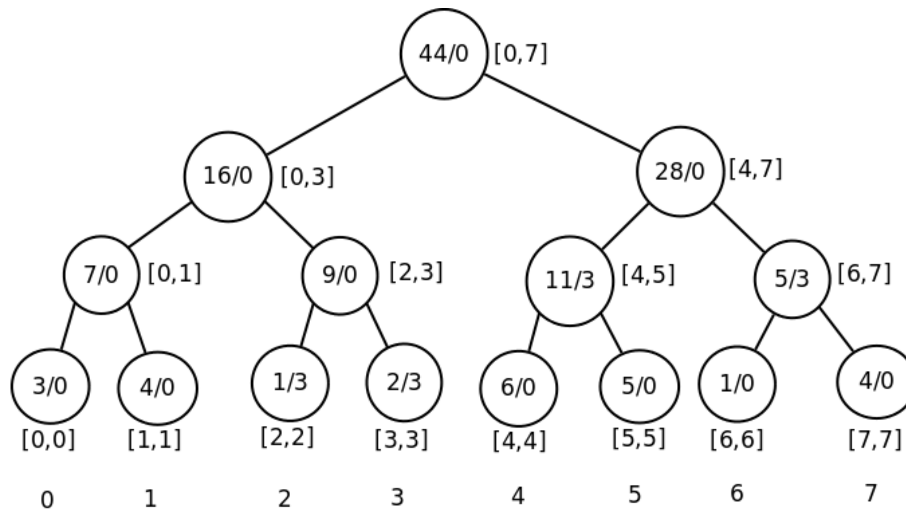
Prikažimo kako bismo sve elemente iz segmenta pozicija $[2, 7]$ uvećali za 3. Krećemo od korena koji pokriva segment $[0, 7]$. Segmenti $[0, 7]$ i $[2, 7]$ se seku, pa stoga ažuriranje prepuštamo naslednicima i nakon njihovog ažuriranja, pri povratku iz rekurzije vrednost određujemo kao zbir njihovih 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 ovaj čvor pokriva 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 direktno 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 $16 + 12 = 28$. Propagaciju ažuriranja kroz poddrvo odlažemo i samo njegovoj deci beležimo da je uvećanje za 3 potrebno izvršiti u



Slika 2: Lenjo segmentno drvo: u svakom čvoru čuvamo vrednost zbira odgovarajućeg segmenta i koeficijenta lenje propagacije.

nekom kasnijem trenutku. Pri povratku iz rekurzije vrednost u korenu ažuriramo sa 26 na $16 + 28 = 44$. Nakon izvršavanja ove operacije dobija se drvo prikazano na slici 3. Ovo lenjo segmentno drvo odgovara nizu $3, 4, 4, 5, 9, 8, 4, 7$.

Pretpostavimo da je sada potrebno elemente iz segmenta $[0, 5]$ uvećavati za vrednost 2. Ponovo se kreće od korena lenjog segmentnog stabla i kada se ustanovi da se segment $[0, 7]$ seče sa segmentom $[0, 5]$ ažuriranje se prepušta deci i vrednost se ažurira tek pri povratku iz rekurzije. Segment $[0, 3]$ je ceo sadržan u segmentu $[0, 5]$, 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 sa vrednošću 11, primećuje se da je on trebalo da bude ažuriran, međutim, još nije, pa se onda njegova vrednost najpre 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. Sada se prelazi 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 drveta prikazanog na slici 4. Ovo



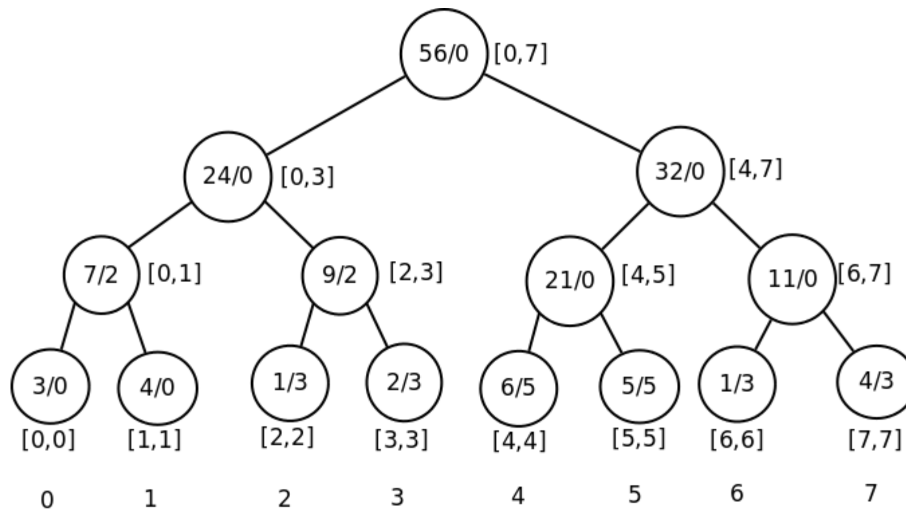
Slika 3: Lenjo segmentno drvo nakon ažuriranja svih elemenata iz segmenta $[2, 7]$ za vrednost 3.

lenjo segmentno drvo odgovara nizu $5, 6, 6, 7, 11, 10, 4, 7$.

Funkcija izračunavanja vrednosti zbira segmenta ostaje praktično nepromenjena, osim što se pri ulasku u svaki čvor vrši njegovo ažuriranje, ako je potrebno. Stoga i njena složenost ostaje nepromenjena i iznosi $O(\log n)$.

```
// 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]
```



Slika 4: Lenjo segmentno drvo nakon ažuriranja svih elemenata iz segmenta $[0, 5]$ za 2.

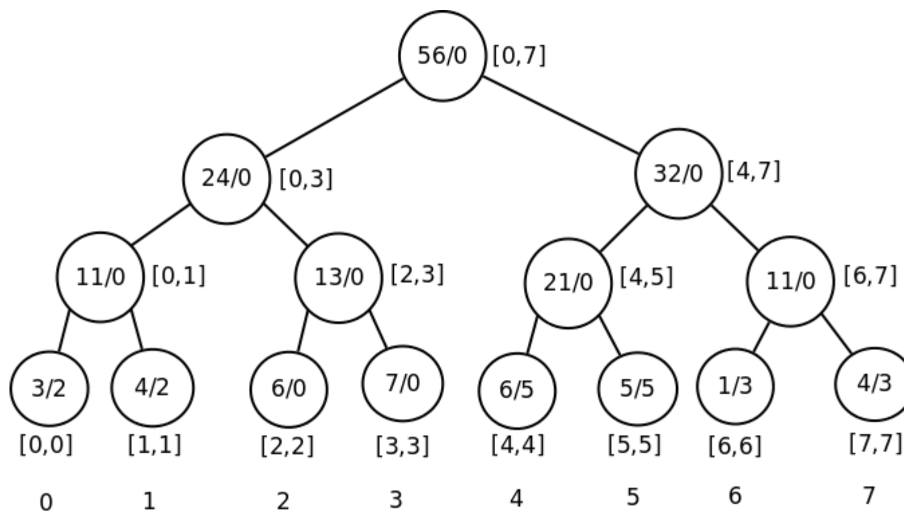
```
  if (a <= x && y <= b)
      return drvo[k];
  // intervali [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return saberi(drvo, lenjo, 2*k, x, s, a, b) +
         saberi(drvo, lenjo, 2*k+1, s+1, y, a, b);
}

// 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 rad prethodne funkcije na tekućem primeru. Razmotrimo kako se za stablo prikazano na slici 4 izračunava zbir elemenata iz segmenta $[3, 5]$. Krećemo od korena drveta koje sadrži sumu segmenta $[0, 7]$. 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 postavlja na 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 uvećava za 2, odnosno postaje 5. 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 $7 + 21 = 28$. Dakle, zbir segmenta $[3, 5]$ u tekućem drvetu je 28. Stanje drveta nakon izvršavanja upita je prikazano na slici 5.



Slika 5: Lenjo segmentno drvo nakon računanja zbira elemenata iz segmenta $[3, 5]$.

Dakle, u lenjom segmentnom drvetu ne možemo analizom pojedinačnog čvora (npr. lista) zaključiti koja je tačna vrednost tog čvora, međutim, kada nam bude bila potrebna vrednost nekog čvora u stablu, mi ćemo se od korena spustiti do tog čvora i, idući tom putanjom, sve vrednosti na toj putanji ažurirati. Na taj način, kada budemo stigli do željenog čvora, imaćemo njegovu ažurnu vrednost, što je jedino i važno.