Upiti raspona

Određene strukture podataka su posebno pogodne za probleme u kojima se traži da se nad nizom elemenata izvršavaju upiti koji zahtevaju izračunavanje statistika nekih raspona tj. segmenata niza koje nazivamo *upiti raspona* (eng. range queries).

Problem: Implementirati strukturu podataka koja obezbeđuje efikasno izračunavanje zbirova segmenata datog niza određenih intervalima pozicija $[a, b]$.

Rešenje grubom silom koje bi smestilo sve elemente u klasičan niz x i pri svakom upitu iznova računalo zbir elemenata na pozicijama iz datog intervala imalo bi složenost $O(mn)$, gde je sa m označen broj upita, a sa n dužina niza, što je u slučaju dugačkih nizova i velikog broja upita nedopustivo neefikasno. Jednostavno rešenje je zasnovano na ideji da umesto da čuvamo elemente niza x, čuvamo niz zbirova prefiksa niza $P_{i+1} = \sum_{k=0}^{i} x_k, P_0 = 0$. Dakle, zbir P_i čuva zbir prvih i elemenata niza (pošto brojanje pozicija počinje od nule, zbir elemenata $x_0, \ldots, x_i$ sadrži $i+1$ sabirak i jednak je P_{i+1}). Zbir svakog segmenta $[a, b]$ onda možemo razložiti na razliku prefiksa niza x do elementa b i prefiksa do elementa $a-1$:

$$\sum_{k=a}^{b} x_k = \sum_{k=0}^{b} x_k - \sum_{k=0}^{a-1} x_k = P_{b+1} - P_a.$$

Svi prefiksi niza x mogu se 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 izvršavanja m upita računanja zbira elemenata nekog segmenta niza x jednaka $O(n+m)$.

Donekle srodan problem ovom problemu je i sledeći.

Problem: Dat je niz x dužine n koji sadrži samo nule. Nakon toga izvršavaju se upiti oblika $([a, b], c)$, koji podrazumevaju da se svi elementi niza x na pozicijama iz intervala $[a, b]$ uvećaju za vrednost c. Potrebno je odgovoriti kako izgleda niz x nakon izvršavanja svih tih upita.

Rešenje grubom silom bi u svakom koraku u petlji uvećavalo sve niza x elemente na pozicijama $[a, b]$. Složenost tog naivnog pristupa bila bi $O(mn)$, gde je sa m označen broj upita, a sa n dužina niza.

Mnogo bolje rešenje se može dobiti ako se umesto elemenata niza pamte razlike između svaka dva susedna elementa niza: $R_0 = x_0, R_i = x_i - x_{i-1}$ za $i > 0$. Ključni uvid je da se tokom uvećavanja svih elemenata segmenta $[a, b]$ niza x za vrednost c menjaju samo razlike između elemenata na pozicijama a i $a-1$ (razlika R_a se uvećava za c) kao i između elemenata na pozicijama $b+1$ i b (razlika R_{b+1} se umanjuje za c). Ako znamo sve elemente niza, tada niz razlika

susednih elemenata možemo veoma jednostavno izračunati u vremenu $O(n)$. Sa druge strane, ako znamo niz razlika, tada originalni niz možemo takođe veoma jednostavno rekonstruisati u vremenu $O(n)$. Jednostavnosti radi, možemo pretpostaviti da početni niz proširujemo sa po jednom nulom sa leve i desne strane.

Može se primetiti da se rekonstrukcija niza vrši zapravo izračunavanjem prefiksnih zbirova niza razlika susednih elemenata, što ukazuje na duboku vezu između ove dve tehnike. Zapravo, razlike susednih elemenata predstavljaju određeni diskretni analogon izvoda funkcije, dok prefiksni zbirovi onda predstavljaju analogiju određenog integrala. Izračunavanje zbira segmenta kao razlike dva zbira prefiksa odgovara Njutn-Lajbnicovoj formuli.

Dakle, niz zbirova prefiksa omogućava efikasno postavljanje upita nad segmentima niza, ali ne omogućava efikasno ažuriranje elemenata niza, jer je potrebno ažurirati sve zbirove prefiksa nakon ažuriranog elementa, što je naročito neefikasno kada se ažuriraju elementi blizu početka niza (složenost najgoreg slučaja je $O(n)$). Niz razlika susednih elemenata dopušta stalna ažuriranja niza, međutim, izvršavanje upita očitavanja stanja niza podrazumeva rekonstrukciju polaznog niza, što je složenosti $O(n)$.

Problemi koje ćemo razmatrati u ovom poglavlju su specifični po tome što omogućavaju da se upiti ažuriranja niza i očitavanja njegovih statistika javljaju isprepletano. Za razliku od prethodnih, *statičkih upita nad rasponima* (eng. static range queries), ovde ćemo razmatrati tzv. *dinamičke upite nad rasponima* (eng. dynamic range queries) koji dozvoljavaju da se niz menja tokom vremena, tako da je potrebno razviti naprednije strukture podataka koje omogućavaju izvršavanje oba tipa upita efikasno. Za početak razmotrimo malo jednostavniji problem.

Problem: Implementirati strukturu podataka koja obezbeđuje efikasno izračunavanje zbirova segmenata datog niza određenih intervalima pozicija $[a, b]$, pri čemu se pojedinačni elementi niza često mogu menjati.

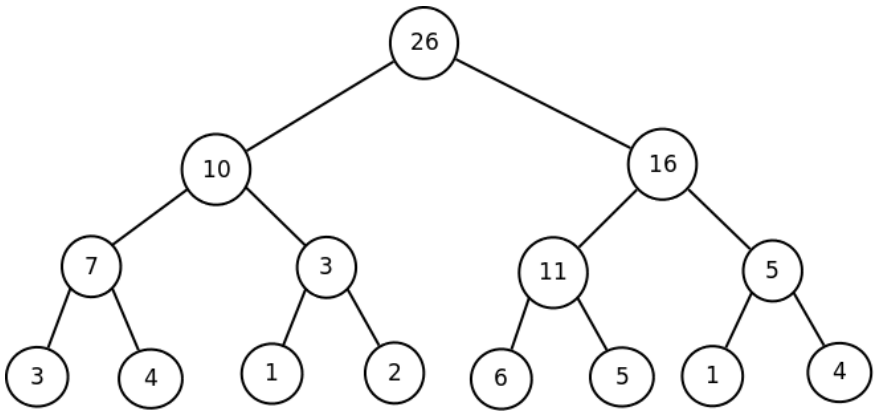
Dakle, pretpostavljamo da imamo dat niz od m operacija od kojih je svaka jednog od prethodno pomenuta dva tipa i cilj je minimizovati ukupno vreme izvršavanja ovog niza operacija. U ovom slučaju nam nije od koristi struktura podataka kod koje se jedna od ove dve operacije izvršava efikasno, a druga neefikasno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

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

Segmentna drveta

Jedna struktura podataka koja omogućava prilično jednostavno i efikasno rešavanje ovog problema su *segmentna drveta* (eng. segment tree). Opet se tokom faze pretprocesiranja izračunavaju zbirovi određenih pogodno odabranih segmenata polaznog niza, a onda se zbir elemenata proizvoljnog segmenta polaznog niza izražava u funkciji tih unapred izračunatih zbirova. Recimo i da segmentna drveta nisu specifična samo za sabiranje, već se mogu koristiti i za druge statistike segmenata koje se izračunavaju asocijativnim operacijama (na primer za određivanje najmanjeg ili najvećeg elementa, nzd-a ili nzs-a svih elemenata i slično).

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



Slika 1: Primer segmentnog drveta.

Pošto je drvo potpuno, najjednostavnija implementacija podrazumeva 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 i 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

[1]Ako se segmentnim drvetom računaju proizvodi segmenata, niz se može dopuniti do stepena dvojke jedinicama, ako se računa najveći zajenički delilac može se dopuniti nulama jer važi $nzd(0, x) = nzd(x, 0) = x$, a ako se računa najmanji zajednički sadržalac jedinicama jer važi $nzs(1, x) = nzs(x, 1) = x$

poziciji 1. Ako polazni niz sadrži n elemenata, onda se u segmentnom drvetu elementi polaznog niza nalaze se na pozicijama $[n, 2n-1]$. Element koji se u polaznom nizu nalazi na poziciji p, se u segmentnom drvetu nalazi na poziciji $p+n$. Levo dete čvora k nalazi se na poziciji $2k$, a desno na poziciji $2k+1$. Dakle, na parnim pozicijama se nalaze leva deca svojih roditelja, a na neparnim desna. Roditelj čvora k nalazi se na poziciji $\lfloor \frac{k}{2} \rfloor$. Na primer, na slici 1 na kojoj je predstavljeno segmentno drvo za niz od 8 elemenata možemo uočiti da se elementi polaznog niza nalaze na pozicijama $[8, 15]$. Levo dete čvora sa vrednošću 10 koji se nalazi na poziciji 2 je čvor sa vrednošću 7 koji se nalazi na poziciji $2 \cdot 2 = 4$, a desno dete je čvor sa vrednošću 3 koji se nalazi na poziciji $2 \cdot 2 + 1 = 5$. Roditelj i čvora na poziciji 4 i čvora na poziciji 5 je čvor koji se nalazi na poziciji $\lfloor \frac{4}{2} \rfloor = \lfloor \frac{5}{2} \rfloor = 2$.

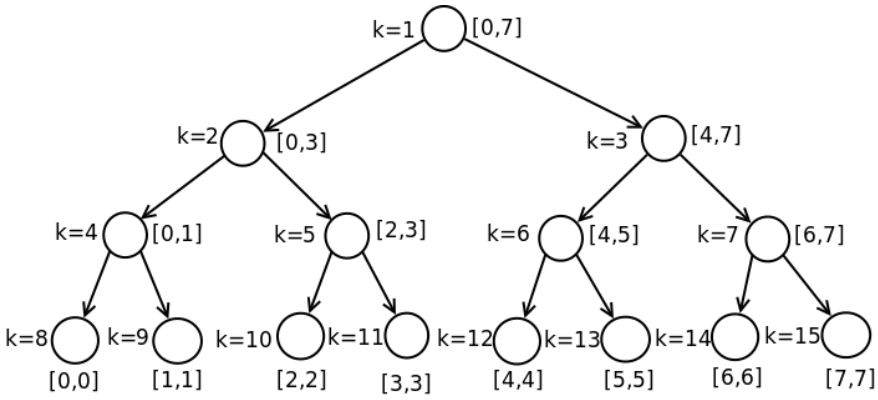
Formiranje segmentnog drveta Formiranje segmentnog drveta na osnovu datog niza je veoma jednostavno. Prvo se elementi polaznog niza prekopiraju u drvo, krenuvši od pozicije n. Zatim se svi unutrašnji čvorovi drveta (od pozicije $n-1$, pa unazad do pozicije 1) popunjavaju kao zbirovi svoje dece (na poziciju 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 u kom se elementi smeštaju
// u niz drvo krenuvši od pozicije 1
void formirajSegmentnoDrvo(int a[], int n, int drvo[]) {
  // kopiramo originalni niz u listove drveta drvo
  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 unutrašnji čvorovi sve dok se ne dođe do korena). Još jedan način je da se drvo formira je rekurzivno, odozgo naniže. Iako je ova implementacija komplikovanija i malo neefikasnija, pristup odozgo naniže je u nekim kasnijim operacijama neizbežan, pa ga ilustrujemo na ovom jednostavnom primeru. Svaki čvor drveta predstavlja zbir određenog segmenta pozicija polaznog niza. Segment je jednoznačno određen pozicijom k u nizu koji odgovara segmentnom drvetu, ali da bismo olakšali implementaciju granice tog segmenta možemo kroz rekurziju prosleđivati kao parametar funkcije, zajedno sa vrednošću k (neka je to segment $[x, y]$). Na slici 2 za svaki čvor segmentnog drveta dat je njegov indeks u odgovarajućem nizu `drvo` i granice segmenta koje taj čvor pokriva.

Drvo krećemo da gradimo od korena gde je $k = 1$ i $[x, y] = [0, n-1]$. Ako roditeljski čvor pokriva segment $[x, y]$, tada levo dete pokriva segment $[x, \lfloor \frac{x+y}{2} \rfloor]$,



Slika 2: Prikaz segmentnog drveta: za svaki čvor drveta prikazane su granice segmenta koji taj čvor pokriva.

a desno dete pokriva segment $[\lfloor \frac{x+y}{2} \rfloor + 1, y]$. Drvo popunjavamo rekurzivno, tako što prvo popunimo levo poddrvo, zatim desno poddrvo i na kraju vrednost u korenu izračunavamo kao zbir vrednosti u levom i desnom detetu. Izlaz iz rekurzije predstavljaju listovi, koje prepoznajemo po tome što pokrivaju segmente dužine 1, i u njih samo kopiramo elemente sa odgovarajućih pozicija polaznog niza.

```
// od elemenata niza a sa pozicija [x, y]
// formira se segmentno drvo i elementi mu se smeštaju u niz drvo
// krenuvši od pozicije k
void formirajSegmentnoDrvo(int a[], int drvo[], int k, int x, int y) {
   if (x == y)
      // u listove prepisujemo elemente polaznog niza
      drvo[k] = a[x];
   else {
      // rekurzivno formiramo levo i desno poddrvo
      int s = (x + y) / 2;
      formirajSegmentnoDrvo(a, drvo, 2*k, x, s);
      formirajSegmentnoDrvo(a, drvo, 2*k+1, s+1, y);
      // izračunavamo vrednost u korenu
      drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
}

// na osnovu datog niza a dužine n
// u kom su elementi smešteni od pozicije 0
// formira se segmentno drvo i elementi mu se smeštaju u niz
// drvo krenuvši od pozicije 1
void formirajSegmentnoDrvo(int a[], int n, int drvo[]) {
```

```
  // krećemo formiranje od korena koji se nalazi u nizu drvo
  // na poziciji 1 i pokriva elemente na pozicijama [0, n-1]
  formirajSegmentnoDrvo(a, drvo, 1, 0, n-1);
}
```

Vremensku složenost prethodne rekurzivne implementacije možemo opisati narednom rekurentnom jednačinom:

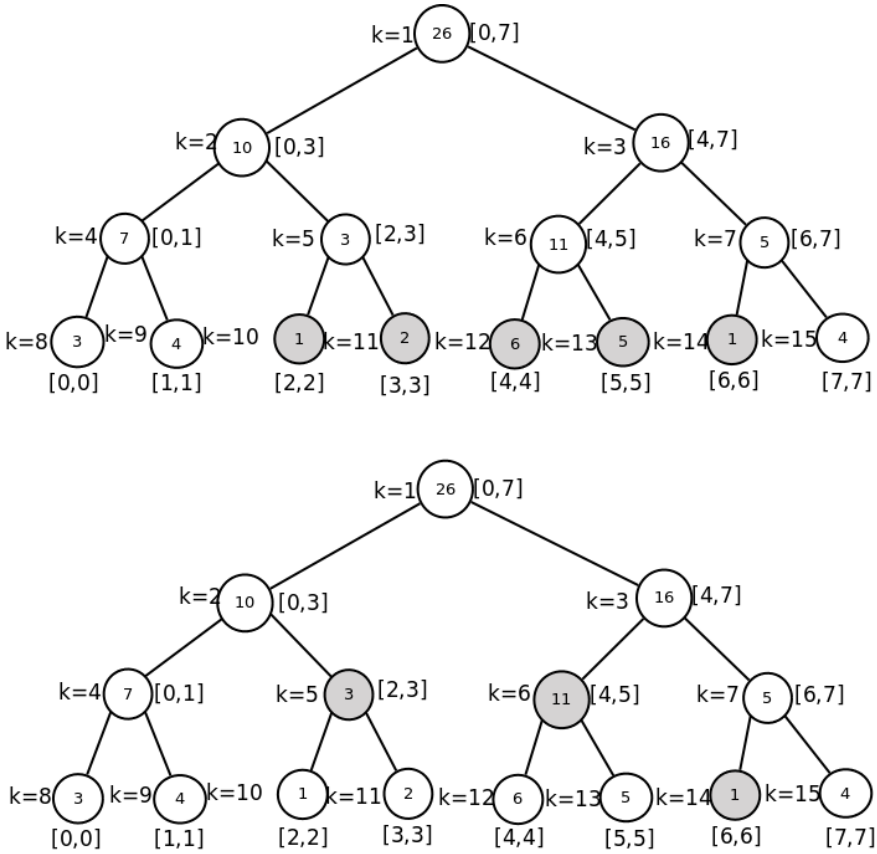
$$T(n) = 2T(n/2) + O(1), T(1) = O(1).$$

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

Računanje zbira elemenata segmenta Razmotrimo sada kako bismo našli zbir elemenata niza $3, 4, 1, 2, 6, 5, 1, 4$ na pozicijama iz segmenta $[2, 6]$, tj. zbir elemenata $1, 2, 6, 5$ i 1. U segmentnom drvetu taj segment je smešten na pozicijama $[2+8, 6+8] = [10, 14]$ (slika 3). Jasno je da ne treba ići do listova i redom sabirati elemente, već na pametan način iskoristiti već izračunate sume segmenata koje se nalaze u drvetu. Naime, zbir prva dva elementa $(1, 2)$ se nalazi u čvoru iznad njih (na poziciji 5), zbir naredna dva elemenata $(6, 5)$ takođe u čvoru iznad (na poziciji 6), dok se u roditeljskom čvoru elementa 1 nalazi njegov zbir sa elementom 4, koji ne pripada segmentu koji sabiramo. Zato zbir elemenata na pozicijama $[10, 14]$ u segmentnom drvetu možemo razložiti na zbir elemenata segmenta na pozicijama $[5, 6]$ i elementa na poziciji 14 koji zasebno dodajemo u traženi zbir. Na ovaj način penjemo se na nivo iznad tekućeg (dolazimo do roditelja tih čvorova) i nastavljamo algoritam na segmentu $[5, 6]$ na isti način. Na poziciji 5 nalazi se element 3, a na poziciji 6 element 11. U roditeljskom čvoru elementa 3 nalazi se zbir sa elementom levo od njega, koji ne pripada razmatranom segmentu. Slično, u roditeljskom čvoru elementa 11 nalazi se zbir ovog elementa sa elementom desno od njega koji ne pripada segmentu čiji zbir računamo. Stoga oba elementa dodajemo zasebno u zbir i ostajemo sa praznim segmentom, čime se postupak računanja sume završava.

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 čiji zbir računamo 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



Slika 3: Računanje zbira elemenata iz segmenta $[2, 6]$ pristupom odozdo naviše. Na prvoj slici su sivom bojom označeni listovi koji odgovaraju traženom segmentu, a na drugoj čvorovi čije se vrednosti sabiraju.

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. Konačno, ako se krajnji desni element nalazi u desnom čvoru (ako je na neparnoj poziciji), tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom levo od njega, koji pripada segmentu koji sabiramo (osim eventualno u slučaju jednočlanog segmenta).

```
// izračunava se zbir elemenata polaznog niza dužine n koji se
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drveta
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
int saberi(int drvo[], int n, int a, int b) {
   a += n; b += n;
   int zbir = 0;
   // sve dok je segment neprazan
   while (a <= b) {
      // ako je levi kraj segmenta desno dete, dodajemo ga posebno u zbir
      if (a % 2 == 1) zbir += drvo[a++];
      // ako je desni kraj segmenta levo dete, dodajemo ga posebno u zbir
      if (b % 2 == 0) zbir += drvo[b--];
      // penjemo se na nivo iznad, na roditeljske cvorove
      a /= 2;
      b /= 2;
   }
   return zbir;
}
```

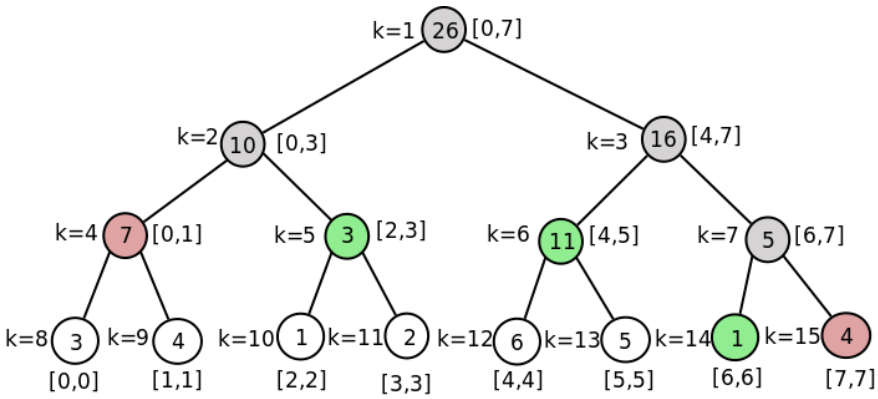
Pošto se u svakom koraku dužina segmenta $[a, b]$ polovi, a ona je inicijalno sigurno manja ili jednaka n, složenost ove operacije je $O(\log n)$.

Prethodna implementacija vrši izračunavanje odozdo naviše. I za ovu operaciju možemo napraviti rekurzivnu implementaciju koja vrši izračunavanje odozgo naniže. U opštem slučaju, neki elementi traženog segmenta su levo od tekućeg čvora, a neki desno (ili su svi levo ili svi desno). Za svaki čvor u segmentnom drvetu funkcija vraća koliki je doprinos segmenta koji odgovara tom čvoru i njegovim naslednicima traženom zbiru elemenata na pozicijama iz segmenta $[a, b]$ u polaznom nizu. Na početku krećemo od korena i računamo doprinos celog drveta zbiru elemenata iz segmenta $[a, b]$. Postoje tri različita moguća odnosa između segmenta $[x, y]$ koji odgovara tekućem čvoru i segmenta $[a, b]$ čiji zbir elemenata tražimo. Ako su disjunktni, doprinos tekućeg čvora zbiru segmenta $[a, b]$ je nula. Ako je $[x, y]$ u potpunosti sadržan u $[a, b]$, tada je doprinos potpun, tj. ceo zbir segmenta $[x, y]$ (a to je broj upisan u nizu na poziciji k) doprinosi zbiru elemenata na pozicijama iz segmenta $[a, b]$. Na kraju, ako se segmenti seku, tada je doprinos tekućeg čvora jednak zbiru doprinosa njegovog levog i desnog deteta. Odatle sledi naredna implementacija.

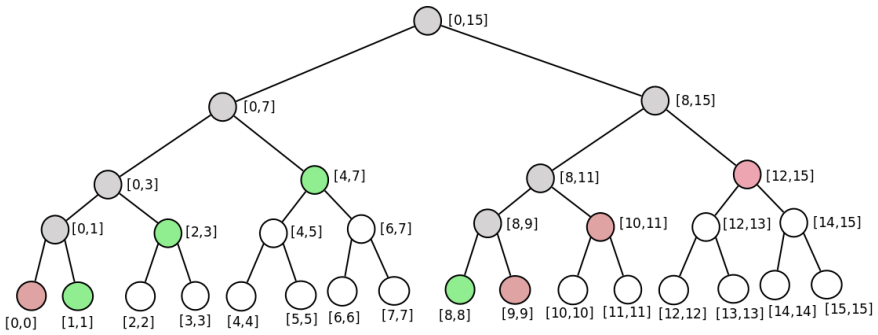
```
// izračunava se zbir onih elemenata polaznog niza koji se
// nalaze na pozicijama [a, b] u polaznom nizu
// a koji se nalaze u segmentnom drvetu koje čuva elemente polaznog niza
// koji se nalaze na pozicijama iz segmenta [x, y]
// i smešteno je u nizu drvo od pozicije k
int saberi(int drvo[], int k, int x, int y, int a, int b) {
  // segmenti [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // segment [x, y] je potpuno sadržan unutar segmenta [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // segmenti [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  // a i b se ne menjaju kroz rekurzivne pozive, dok se k, x i y menjaju
  return saberi(drvo, 2*k, x, s, a, b) +
         saberi(drvo, 2*k + 1, s+1, y, a, b);
}

// izračunava se zbir elemenata polaznog niza dužine n
// koji se nalaze na pozicijama iz segmenta [a, b]
// na osnovu segmentnog drveta koje je smešteno u nizu drvo,
// krenuvši od pozicije 1
int saberi(int drvo[], int n, int a, int b) {
  // krećemo od drveta smeštenog od pozicije 1 koje
  // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
  return saberi(drvo, 1, 0, n - 1, a, b);
}
```

Razmotrimo kako se ovim pristupom određuje zbir elemenata na pozicijama iz segmenta $[2, 6]$, prikazan na slici 4. Izvršavanje kreće od korena. Segment $[0, 7]$ se seče sa segmentom $[2, 6]$ te će zbir biti jednak sumi doprinosa segmenata $[0, 3]$ i $[4, 7]$. Segment $[0, 3]$ se seče sa segmentom $[2, 6]$ te opet startujemo dva rekurzivna poziva za segmente $[0, 1]$ i $[2, 3]$. Segment $[0, 1]$ je disjunktan sa segmentom $[2, 6]$ pa je njegov doprinos traženoj sumi 0, a segment $[2, 3]$ je sadržan u segmentu $[2, 8]$ te je njegov doprinos potpun – jednak je vrednosti 3 koju taj čvor čuva. S druge strane, segment $[4, 7]$ se seče sa segmentom $[2, 6]$ te opet startujemo dva rekurzivna poziva za segmente $[4, 5]$ i $[6, 7]$. Segment $[4, 5]$ je u potpunosti sadržan u segmentu $[2, 8]$ te je njegov doprinos potpun i iznosi 11, a segment $[6, 7]$ se seče sa segmentom $[2, 6]$, pa iz njega startujemo dva rekurzivna poziva: za segmente $[6, 6]$ i $[7, 7]$: prvi je u potpunosti sadržan u traženom segmentu, te je njegov doprinos potpun i iznosi 1, a drugi je disjunktan pa je njegov doprinos jednak nula. Dakle, ukupna vrednost sume segmenta biće jednaka $3 + 11 + 1 = 15$.

I ova implementacija će imati složenost $O(\log n)$. Naime može se pokazati da se za svaki nivo segmentnog drveta obilaze najviše po četiri čvora, a pošto je visina segmentnog drveta $O(\log n)$, dobijamo datu ocenu složenosti. Dokažimo



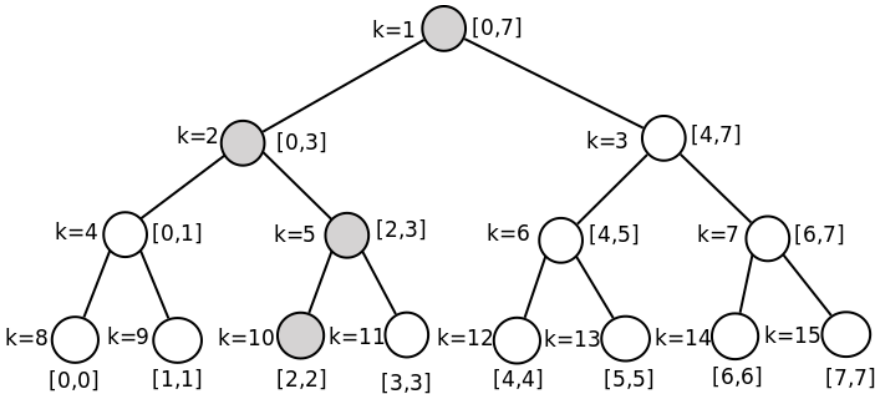
Slika 4: Računanje zbira elemenata iz segmenta $[2, 6]$ pristupom odozgo naniže. Sivom bojom su označeni čvorovi koji odgovaraju segmentima koji se seku sa traženim segmentom, roze bojom čvorovi koji su disjunktni, a zelenom bojom čvorovi koji odgovaraju segmentima koji su u potpunosti sadržani u traženom segmentu.



Slika 5: Računanje zbira elemenata iz segmenta $[1, 8]$ pristupom odozgo naniže. Sivom bojom su označeni čvorovi koji odgovaraju segmentima koji se seku sa traženim segmentom, roze bojom čvorovi koji su disjunktni, a zelenom bojom čvorovi koji odgovaraju segmentima koji su u potpunosti sadržani u traženom segmentu.

ovo tvrđenje principom matematičke indukcije. Na prvom nivou se posećuje samo jedan čvor, koren drveta, tako da se na ovom nivou posećuje manje od četiri čvora i baza indukcije važi. Razmotrimo sada proizvoljni nivo drveta: prema induktivnoj hipotezi na njemu se posećuje najviše četiri čvora. Ako se posećuje najviše dva čvora, u narednom nivou se posećuje najviše četiri čvora jer svaki čvor može da proizvede najviše dva rekurzivna poziva. Pretpostavimo da se na tekućem nivou posećuju tri ili četiri čvora. Oni mogu biti susedni (slika 4, nivo 2), ali i ne moraju (slika 5, nivo 3). Pošto čvorovi na jednom nivou segmentnog drveta sadrže sume disjunktnih segmenata, jedini čvorovi iz kojih se mogu pokrenuti rekurzivni pozivi su oni koji sadrže granice segmenta čiju sumu računamo: ostali segmenti će biti ili u potpunosti sadržani ili disjunktni sa segmentom čiju sumu računamo. Stoga, na svakom nivou postoje samo dva čvora iz kojih se mogu startovati rekurzivni pozivi, tako da će i naredni nivo zadovoljavati polazno tvrđenje. Dakle, posećuje se ukupno najviše $4 \log n$ čvorova segmentnog drveta, pa je vremenska složenost i ovog pristupa $O(\log n)$.

Ažuriranje vrednosti elementa Prilikom ažuriranja nekog elementa potrebno je ažurirati sve čvorove na putanji od tog lista do korena (slika 6). S obzirom na to da znamo poziciju roditelja svakog čvora, ova operacija se može veoma jednostavno implementirati.



Slika 6: Ažuriranje elementa na poziciji 2 u polaznom nizu. Sivom bojom su označeni čvorovi čije se vrednosti ažuriraju.

```
// 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 prethodne implementacije možemo opisati rekurentnom jednačinom:

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

i njeno rešenje iznosi $O(\log n)$. Naime dužina intervala $[x, y]$ se u svakom narednom pozivu smanjuje dva puta, a njegova početna dužina je maksimalno jednaka n.

Umesto funkcije `promeni` često se razmatra funkcija `uvecaj` koja element na

poziciji `i` polaznog niza uvećava za datu vrednost `v` i u skladu sa tim ažurira segmentno drvo. Svaka od ove dve funkcije se jednostavno izražava preko one druge.

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

Fenvikova drveta (BIT)

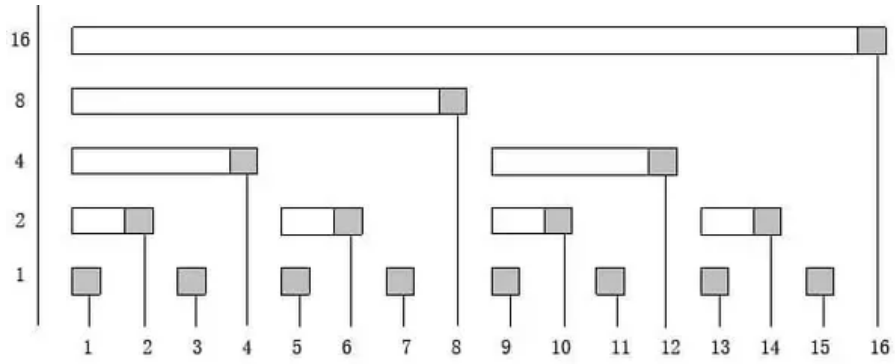
U nastavku ćemo razmotriti još jednu strukturu podataka koja omogućava efikasno računanje statistika nad segmentima niza i ažuriranje pojedinačnih elemenata: to su *Fenvikova drveta* (eng. Fenwick tree), tj. *binarno indeksirana drveta* (eng. binary indexed tree, BIT) koja koriste malo manje memorije od segmentnih drveta i mogu od njih biti za konstantni faktor brža (iako je složenost operacija asimptotski jednaka). S 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, najveći zajednički delioci i slično). Potrebu da za razmatranu operaciju postoji inverz ćemo detaljnije diskutovati kada budemo govorili o realizaciji samih operacija. 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 radi sa drvetom indeks polaznog niza potrebno uvećati za jedan pre dalje obrade. Dakle, 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 dobija od broja k tako što se iz binarnog zapisa broja k obriše prva jedinica zdesna.*

Na primer, u drvetu se na mestu $k = 21$ zapisuje 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 u drvetu nalazi se zbir elemenata polaznog niza sa pozicija iz intervala $(16, 20]$, jer se brisanjem krajnje desne jedinice iz njegovog binarnog zapisa dobija binarni zapis 10000 tj. broj 16 (važi da je $f(20) = 16$). U drvetu se na poziciji 16 čuva zbir elemenata polaznog niza 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 7 je za niz dužine 16 prikazano koje se sume segmenata polaznog niza čuvaju na svakoj od pozicija u Fenvikovom drvetu. Primetimo da neparne pozicije odgovaraju segmentima dužine 1, a da za stepene broja 2 segmenti predstavljaju prefikse polaznog niza do te pozicije.



Slika 7: Prikaz segmenata čije su sume smeštene u elementima Fenvikovog drveta dužine 16.

Primer 1: Razmotrimo niz elemenata sa vrednostima $3, 4, 1, 2, 6, 5, 1, 4$. Odgovarajuće 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      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
```

Primetimo da se nadovezivanjem intervala $(0, 16]$, $(16, 20]$ i $(20, 21]$ dobija interval $(0, 21]$ tj. prefiks niza do pozicije 21. Zbir elemenata u nekom prefiksu polaznog niza se, dakle, može dobiti kao zbir nekoliko elemenata zapisanih u Fenvikovom drvetu. Ovo svojstvo, naravno, važi za proizvoljni indeks, a ne samo za broj 21. Broj elemenata Fenvikovog drveta čijim se sabiranjem dobija zbir prefiksa polaznog niza 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 Fenvikovog drveta 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.

Računanje zbira elemenata segmenta 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 polaznog niza, zbir proizvoljnog segmenta $[a, b]$ polaznog niza možemo izračunati kao razliku zbira prefiksa $(0, b]$ i zbira prefiksa $(0, a-1]$. Pošto se oba prefiksa računaju u vremenu $O(\log n)$, i zbir svakog segmenta možemo izračunati u vremenu $O(\log n)$. Istaknimo na ovom mestu 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[2]).

Ažuriranje vrednosti elementa Ideju računanja zbira elemenata proizvoljnog segmenta kao razlike dve prefiksne sume smo videli već ranije, kada smo u te svrhe održavali niz svih prefiksnih suma. Osnovna prednost Fenvikovih drveta u odnosu na niz svih prefiksnih suma je to što se mogu efikasno ažurirati. Naime, ažuriranje jednog elementa polaznog niza može rezultovati izmenom vrednosti velikog broja prefiksnih suma, u najgorem slučaju $O(n)$ prefiksnih suma, 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

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

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 vrednost izraza `k & -k`.

Primer 2: Na primer, ako bi se u prethodnom primeru element na poziciji 3 u polaznom nizu uvećao za vrednost 4, bilo bi potrebno povećati za 4 vrednosti elemenata Fenvikovog drveta na pozicijama 3, 4 i 8. Do niza ovih pozicija bismo mogli da dođemo na sledeći način: binarnom zapisu broja 3 koji iznosi 11, dodajemo broj 1 (broj koji sadrži tačno jednu jedinicu na poziciji poslednje jedinice u binarnom zapisu datog broja) i dobijamo binarni zapis 100 koji odgovara broju 4; 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 pošto 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 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. pozicije k za koje važi

$$f(m) < k \leq m. \tag{1}$$

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 $1 \leq k \leq n$). Dakle prva pozicija u drvetu koju treba ažurirati je pozicija k.

Za sve brojeve $m > k$, sigurno važi desna nejednakost i jedino je potrebno utvrditi da li važi leva. Razmotrimo koji je to najmanji broj strogo veći od k za koji 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 po formuli `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 vrednosti $f(g(k))$ 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, odnosno da nijedan broj m između k i $g(k)$ ne može da zadovolji uslov $f(m) < k$. Naime, svi brojevi iz intervala $(k, g(k))$ 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 prethodno razmatranom 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. Brisanjem poslednje jedinice u binarnom zapisu ovih brojeva 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, takođe, 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 (1), jer se svi oni poklapaju sa $g(k)$ u svim binarnim ciframa, osim na njegovim krajnjim nulama gde imaju neke jedinice. Brisanjem poslednje jedinice iz binarnog zapisa dobija se broj koji je veći ili jednak $g(k)$, pa dobijeni broj ne može biti manji od k. Otuda sledi da su prilikom ažuriranja elementa na poziciji k u polaznom nizu 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 prethodna implementacija korektna.

Formiranje Fenvikovog drveta 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]);
}
```

Primer 3: Razmotrimo problem formiranja Fenvikovog drveta za niz vrednosti $3, 4, 1, 2, 6, 5, 1, 4$. Krenuvši od niza koji sadrži samo nule, uvećavamo jedan po jedan element niza.

```
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. 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);
  }
}
```

Izračunavanje suma svih prefiksa polaznog niza je vremenske složenosti $O(n)$, ali se nakon toga elementi Fenvikovog drveta računaju u vremenu $O(1)$, te je ukupna vremenska složenost ovog pristupa $O(n)$. Ova implementacija pak 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 opažanje 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];
  }
}
```

Ova implementacija algoritma za formiranje Fenvikovog drveta je vremenske složenosti $O(n)$, a prostorne složenosti $O(1)$.

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, dok ažuriranje celih segmenata niza odjednom nije direktno podržano. Naime, ako se ono svede na pojedinačno ažuriranje svih elemenata unutar segmenta, dobija se loša složenost.

Moguće je upotrebiti Fenvikovo drvo (ili, analogno, segmentno drvo) tako da se efikasno podrži uvećavanje svih elemenata iz datog segmenta odjednom, ali se onda 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 segment elemenata koji treba ažurirati predstavlja ceo niz, tj. ako treba ažurirati sve elemente polaznog niza, to zahteva ažuriranje vrednosti svih elemenata segmentnog drveta (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. Naime, 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, a koeficijent lenje propagacije tog čvora postaviti na nula (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 funkcija 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 ili već 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 njegova dva deteta.

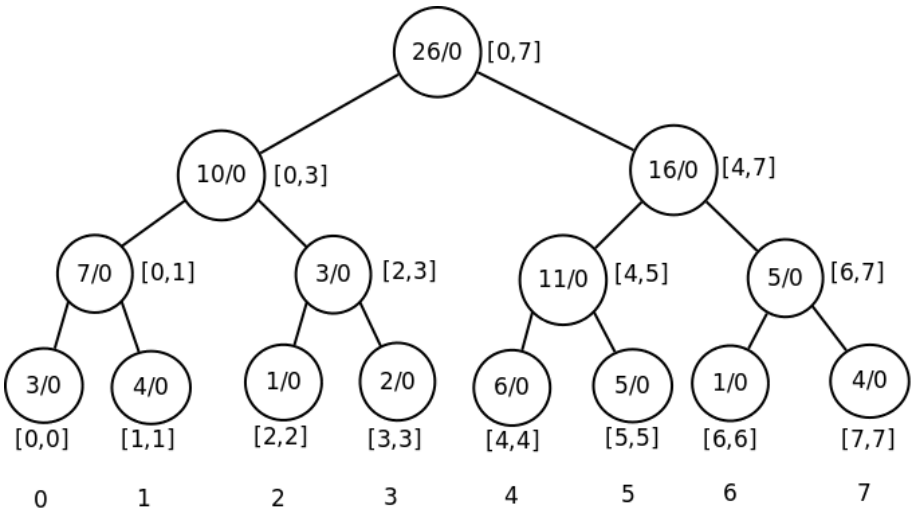
Lenjo segmentno drvo čuvaćemo korišćenjem dva niza: `drvo` i `lenjo`: u prvom ćemo čuvati elemente segmentnog drveta, a u drugom koeficijente lenje propagacije svakog od čvorova.

```
// 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 $4 \log n$.

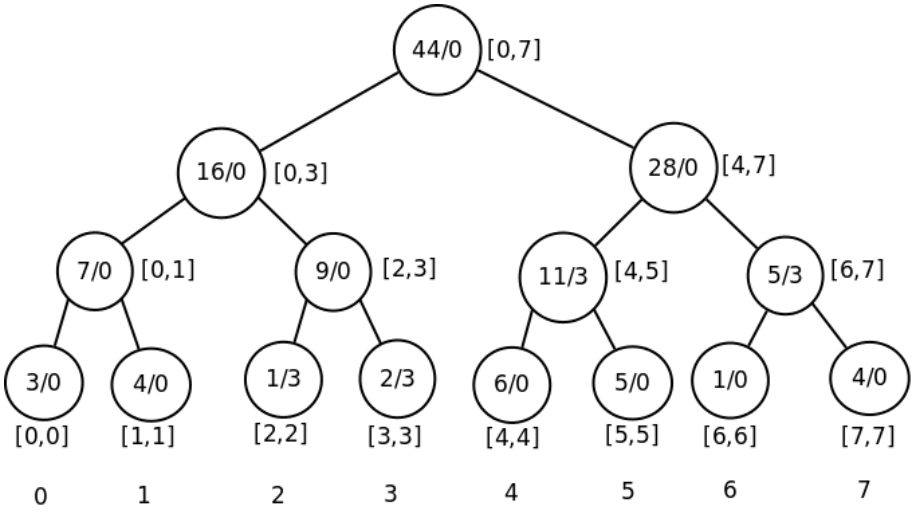
Prikažimo rad ove funkcije na primeru lenjog segmentnog drveta sa slike 8.



Slika 8: Lenjo segmentno drvo: u svakom čvoru čuvamo vrednost zbira odgovarajućeg segmenta i koeficijenta lenje propagacije. Inicijalno su vrednosti svih koeficijenata lenje propagacije jednaki nula.

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 deci i nakon njihovog ažuriranja, pri povratku iz rekurzije vrednost korena određujemo kao zbir ažuriranih vrednosti njegove dece. Na levoj strani se segment $[0, 3]$ seče sa segmentom $[2, 7]$ pa i on prepušta ažuriranje naslednicama i ažurira se tek pri povratku iz rekurzije. Segment $[0, 1]$ je disjunktan u odnosu na segment $[2, 7]$ i tu onda nije potrebno ništa raditi. Segment $[2, 3]$ je ceo sadržan u segmentu $[2, 7]$, i za njega direktno znamo kako se zbir uvećava: pošto ovaj čvor pokriva dva elementa i svaki se uvećava za 3, zbir ovog segmenta 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

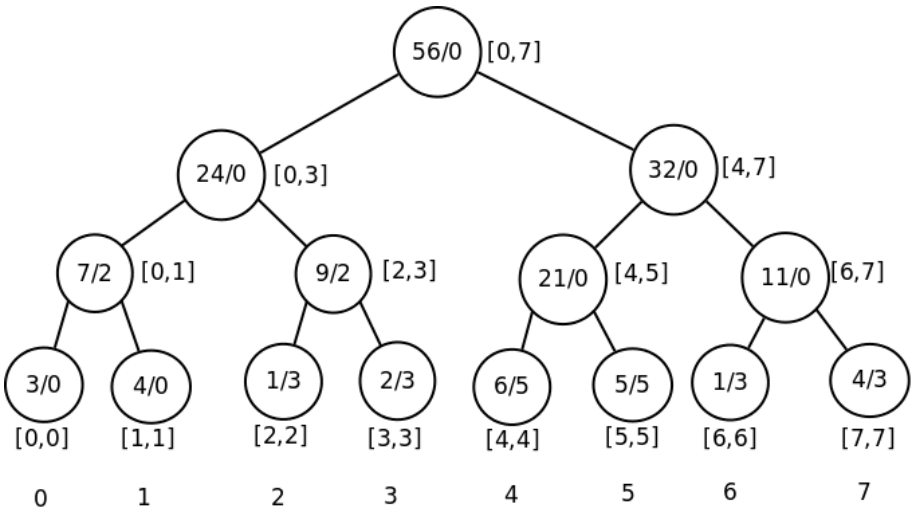
element koren, već samo naslednicima upisujemo da je potrebno propagirati uvećanje za 3, ali samu propagaciju odlažemo za trenutak kada ona postane neophodna. U povratku iz rekurzije, vrednost 10 ažuriramo i nova vrednost je jednaka $7 + 9 = 16$. Što se tiče desnog poddrveta, segment $[4, 7]$ je ceo sadržan u segmentu $[2, 7]$, pa možemo direktno izračunati novu vrednost zbira u ovom čvoru. Naime, 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 sa korenom u ovom čvoru 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 izvršavanja ove operacije dobija se drvo prikazano na slici 9. Ovo lenjo segmentno drvo odgovara nizu $3, 4, 4, 5, 9, 8, 4, 7$.



Slika 9: Lenjo segmentno drvo nakon ažuriranja svih elemenata iz segmenta $[2, 7]$ za vrednost 3.

Pretpostavimo da je u dobijenom segmentnom drvetu potrebno elemente iz segmenta $[0, 5]$ uvećavati za vrednost 2. Ponovo se kreće od korena lenjog segmentnog drveta i kada se ustanovi da se segment $[0, 7]$ seče sa segmentom $[0, 5]$ ažuriranje se prepušta deci i vrednost u korenu 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 jer je njegov koeficijent lenje propagacije različit od nula, međutim još nije, pa se najpre 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 segmentu $[0, 5]$, pa se 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, najpre se 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 čime se dobija drvo prikazano na slici 10. Ovo lenjo segmentno drvo odgovara nizu $5, 6, 6, 7, 11, 10, 4, 7$.



Slika 10: Lenjo segmentno drvo nakon ažuriranja svih elemenata iz segmenta $[0, 5]$ za 2.

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 $O(\log n)$.

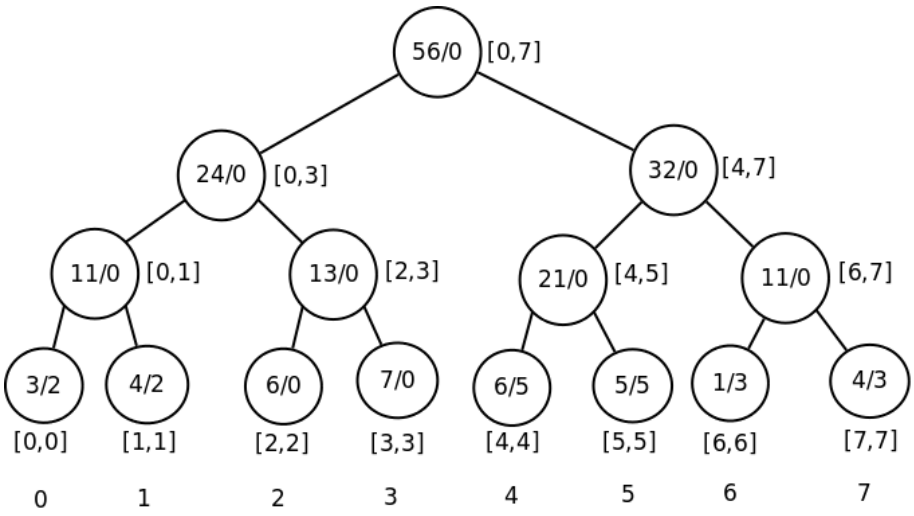
```
// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [x, y]
// izračunava se zbir elemenata polaznog niza
// sa pozicija iz segmenta [a, b]
int saberi(int drvo[], int lenjo[], int 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]
  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,
r  // 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 drvo prikazano na slici 10 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 11.



Slika 11: 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 drvetu, 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.