Drvo segmenata

Drvo segmenata je struktura podataka koja nam dozvoljava da za $O(\log n)$ uradimo sljedeće operacije: nalaženje zbira-minimuma elemenata niza u segmentu $a[l..r]$, gdje su l i r upiti koji dolaze na ulaz; mogući su i upiti tipa izmjene jednog elementa ili izmjene svih elemenata na nekom segmentu (tj. dodjeljivanje vrijednost svim elementima na segmentu ili povećavanje svih vrijednosti na segmentu za datu vrijednost). Primjenom ove strukture rješava se veliki broj zadataka koji se pojavljuju na takmičenja. Pored navedenih operacija, moguće su i mnogo složenije operacije, kao i uopštenje strukture na više dimenzija (npr. podmatrica date matrice, ali za vrijeme $O(\log^2 n)$).

Postoji i struktura drvo intervala i drvo segmenata koji se koriste u računarskoj geometriji. U ovom poglavlju posmatramo samo gore opisani zadatak. Za ovu strukturu je potrebna linearna količina memorije: standardnom drvetu treba oko $4n$ memorijskih elemenata za rad nad nizom od n elemenata.

Osnovna varijanta drveta segmenata

Ramotrićemo najprostiji slučaj drveta segmenata: drvo za zbirove. Formalno, dat je niz $a[0..n-1]$ i naše drvo treba da odgovara na upite tipa „za date l i r naći sumu elemenata u segmentu $a[l..r]$“ i „dodijeliti vrijednost x elementu $a[i]$“.

Struktura drveta

Izračunamo i zapamtimo zbir svih elemenata niza a. Takođe, izračunamo i zapamtimo zbir prve i druge polovine niza, tj. segmenata $a[0..n/2]$ i $a[n/2+1..n-1]$. Sada svaku od polovina opet razbijemo na polovine i zapamtimo te zbirove. Nastavljamo ovaj postupak, dok ne dobijemo samo jedan element. Drugim riječima, dijelimo segment $[0..n-1]$ na polovine sve dok ne dobijemo jedinični segment; za svaki segment čuvamo zbir elemenata u njemu. Može se reći da ovi segmenti čine drvo: korijen drveta je segment $[0..n-1]$ a svaki čvor drveta ima tačno dva sina (osim listova, tj. segmenata dužine 1). Otuda potiče i naziv strukture, iako se nikakvo drvo ne kreira eksplicitno.

Primjetite da drvo ima manje od $2n$ čvorova. To se lako vidi iz načina kreiranja drveta: na prvom nivou imamo jedan čvor, na drugom najviše 2, na trećem najviše 4, itd, sve dok ne dostignemo n. Dakle, ukupna broj čvorova u drvetu je $n+n/2+n/4+...+1<2n$. Takođe, ako n nije stepen broja 2, tada svi nivoi neće biti potpuno popunjeni. Na primjer, za $n=3$, lijevi sin korijena je segment $[0..1]$ a desni je $[2..2]$ koji je list. Visina (dubina) drveta je $O(\log n)$, jer pri prelasku na niži nivo dužina segmenta je dva puta manja.

Kreiranje drveta

Koristimo metod od dna prema vrhu (engl. bottom-up): na početku upišemo vrijednosti niza $a[i]$ u listove drveta a zatim na osnovu njih odredimo zbirove parova koji su na sljedećem nivou, ponovimo za sljedeći nivo, itd. Ovo možemo uraditi rekurzivno: pozovimo funkciju na korijenu drveta a funkcija ako nije pozvana iz lista poziva samu sebe za lijevog i desnog sina ii izračunava zbirove, a ako je pozvana na listu, samo upisuje u sebe vrijednost elementa niza. Složenost kreiranja drveta je dakle $O(n)$

Upit tipa “zbir na segmentu”

Na ulazu su data 2 broja `l` i `r`. Za vrijeme `O(logn)` naći zbir brojeva iz segmenta `a[l..r]`.

Zadatak rješavamo tako što krećemo iz korijena drveta i koristimo prethodno izračunate zbirove u čvorovima. Razmotrimo u koji od sinova upada segment tipa `[l,r]`. Moguće su dvije varijante: segment `[l..r]` u cijelosti pripada jednom od sinova ili segment se siječe sa oba sina. Prvi slučaj je prost: samo pređemo u sina i primjenimo opet ovaj algoritam. U drugom slučaju, prelazimo u lijevog sina i nađemo odgovor u njemu a zatim pređemo u desnog sina, izračunamo odgovor i dodamo ga na prethodni odgovor iz lijevog sina. Drugim riječima, ako je lijevi sin predstavljao segment `[l1..r1]` a desni segment `[l2..r2]` (ne zaboravite da je `l2=r1+1`) tada pređemo u lijevog sina sa upitom `[l..r1]` a u desnog sina sa upitom `[l2..r]`. Dakle, upit tipa „zbir na segmentu“ je rekurzivna funkcija koja poziva sebe ili na lijevom sinu ili na desnom sinu ne mijenjajući granice upita ili poziva sebe za oba sina na dva odgovarajuća podupita. Rekurzivni poziv ne izvršavamo uvijek: ako se tekući upit poklapa sa granicama segmenta u nekom čvoru, samo vratimo zbir koji je pridružen tom čvoru.

Da bi dokazali da je složenost ovog algoritma `O(logn)`, odredimo koliko je segemenata mogla posjetiti naša rekurzivna funkcija. Na nultom nivou rekurzije, samo je jedan čvor. Na prvom nivou rekurzije u najgorem slučaju imamo 2 rekurzivna poziva, al će oni sadejstvovati tj. broj upita u drugom rekurzivnom pozivu biće za jedan veći od broja upita u prvom pozivu. Odatle slijedi da na sljedećem nivou svaki od tih poziva generisati još po 2 rekurzivna poziva, ali će polovina od njih biti obrađena nerekurzivno (zbog poklapanja granica upita sa granicom čvora). Dakle, u svakom trenutku biće najviše dvije grane rekurzije (možemo reći da se jedna grana približava lijevoj granici upita a druga desnoj granici). A broj čvorova drveta ne može biti veći od 4. Visina drveta je proporcionalna sa `logn`, to znači da je složenost zaista `O(logn)`.

Na kraju, dajmo još jedno tumačenje rada algortma: ulazni segment `[l..r]` razbijamo na nekoliko segmenata za koje je odgovor već izračunat i smješten u drvo. Ako to razbijanje uradimo kako treba, broj segemenata biće `O(logn)`.

Upiti modifikacije

Obradimo sada upite tipa „za dato `i` i `x`, postaviti `a[i]=x`“. Ovaj tip upita je prostiji od prethodnog tipa. Broj `a[i]` se pojavljuje u relativno malom broju čvorova drveta: na svakom nivou u po jednom čvoru tj. u ukupno `O(n)` čvorova. Napišimo rekurzivnu funkciju kojoj predajemo tekući čvor drveta; ona poziva rekurzivno onog sina u čijem segmentu pripada indeks i i poslije toga preračunava vrijednost zbira u tekućem čvoru na isti način kako smo to radili kod kreiranja drveta.

Implementacija

Glavni momenat u realizaciji je: kako čuvati u memoriji drvo segmenata? Jedan od načina je da se kreira eksplicitno drvo koje se čuva u nizu, tako da ako čvor ima indeks `i`, tada je lijevi sin na poziciji `2i` a desni sin na `2i+1`. Sada još samo moramo uvesti neki niz za čuvanje zbirova na svakom segmentu. Jedino treba obratiti pažnju da dimenzija takvog iza treba da je `4n` a ne `2n`. Stvar je u tome što ovakva numeracija ne radi najbolje ako n nije stepen broja 2 – tada se pojavljuju propušteni brojevi kojima ne odgovara nijedan čvor drveta (faktički, numeracija se ponaša kao da smo n zaokružili na prvi stepen broja 2 veći od n), što ne

dovodi do problema pri realizaciji ali dimenziju niza treba uvećati na `4n`. U našoj implementaciji, niz `t[]` je predviđen za drvo segmenata:

```
int n, t[4*MAXN];
```

Funkcija za kreiranje drveta po datom nizu `a[]` ima sljedeći oblik: rekurzivna je i ima argumente sam niz `a[]`, broj `v` tekućeg čvora i granice `tl` i `tr` segmenta koji odgovara tom čvoru. Iz glavnog programa je pozivamo sa argumentima `v=1, tl=0, tr=n-1`.

```
void build (int a[], int v, int tl, int tr) {
	if (tl == tr)
		t[v] = a[tl];
	else {
		int tm = (tl + tr) / 2;
		build (a, v*2, tl, tm);
		build (a, v*2+1, tm+1, tr);
		t[v] = t[v*2] + t[v*2+1];
	}
}
```

Funkcija za upite tipa zbira, je takođe rekurzivna funkcija, koja ima iste argumente kao i prethodna, i još argumente `l` i `r` koji su granice tekućeg upita. Radi jednostavnosti, funkcija uvijek ima dva rekurzivna poziva, iako je u stvari potreban samo jedan: suvišnom pozivu se predaju `l` i `r` takve da je `l>r`.

```
int sum (int v, int tl, int tr, int l, int r) {
	if (l > r)
		return 0;
	if (l == tl && r == tr)
		return t[v];
	int tm = (tl + tr) / 2;
	return sum (v*2, tl, tm, l, min(r,tm))
		+ sum (v*2+1, tm+1, tr, max(l,tm+1), r);
}
```

Na kraju, upis modifikacije. I njemu se predaje informacija o tekućem čvoru drveta, indeks elementa koji se mijenja i nova vrijednost.

```
void update (int v, int tl, int tr, int pos, int new_val) {
	if (tl == tr)
		t[v] = new_val;
	else {
		int tm = (tl + tr) / 2;
		if (pos <= tm)
			update (v*2, tl, tm, pos, new_val);
		else
			update (v*2+1, tm+1, tr, pos, new_val);
		t[v] = t[v*2] + t[v*2+1];
	}
}
```

Primjetite da je lako napraviti nerekurzivnu verziju funkcije `update`, pošto je u pitanju tzv. „repna“ rekurzija. Međutim, nerekurzivna verzija može raditi nekoliko puta brže od rekurzivne verzije.

Kao moguće optimizacije, množenje i dijeljenje brojem 2 mogu se zamijeniti „bit po bit operacijama“ (bitwise operacijama), što takođe ubrzava kod.

Složenije verzije drveta segmenata

Drvo segmenata je veoma prilagodljiva struktura i dozvoljava nam da ga uopštimo na različite načine. U ovom poglavlju pokušaćemo da klasifikujemo neke od tih načina.

Složenije funkcije u upitima

Poboljšanja drveta u ovom smjeru mogu biti očigledna (kao u slučaju upita tipa minimuma ili maksimuma umjesto upita tipa zbir) ali i veoma komplikovana.

Traženje minimuma/maksimuma

U originalnom zadatku, umjesto zbira tražićemo minimum ili maksimum na segmentu. Tada se drvo praktično ni u čemu ne razlikuje od onog opisanog ranije. Samo treba izmijeniti način izračunavanja `t[v]` u funkcijama `build` i `update` a takođe i izračunavanje rezultata u funkciji `sum` (zamijeniti zbir minimumom ili maksimumom)

Traženje minimuma/maksimuma i broja njegovog pojavljivanja

Zadatak je isti kao prethodni, ali još pored npr. maksimuma treba vratiti i koliko se puta ona pojavljuje na segmentu. Ovaj zadatak se prirodno pojavljuje ako npr. rješavamo sljedeći zadatak pomoću drveta segemenata: naći broj najdužih rastućih podnizova uzastopnih elemenata u datom nizu.

Da bi riješili ovaj zadatak, u svakom čvoru drveta čuvaćemo par brojeva: pored maksimuma i broj pojavljivanja maksimuma u datom segmentu. Tada pri kreiranju drveta, moramo po dva takva para dobijena od sinova da kreiramo par za tekući čvor. Spajanje dva takva para u jedan treba raditi i u upitima modifikacije i upitima traženja maksimuma, pa je ta operacija odvojena u posebnu funkciju.

```
pair<int,int> t[4*MAXN];

pair<int,int> combine (pair<int,int> a, pair<int,int> b) {
	if (a.first > b.first)
		return a;
	if (b.first > a.first)
		return b;
	return make_pair (a.first, a.second + b.second);
}

void build (int a[], int v, int tl, int tr) {
	if (tl == tr)
		t[v] = make_pair (a[tl], 1);
	else {
		int tm = (tl + tr) / 2;
		build (a, v*2, tl, tm);
		build (a, v*2+1, tm+1, tr);
```

```
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}

pair<int,int> get_max (int v, int tl, int tr, int l, int r) {
        if (l > r)
                return make_pair (-INF, 0);
        if (l == tl && r == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        return combine (
                get_max (v*2, tl, tm, l, min(r,tm)),
                get_max (v*2+1, tm+1, tr, max(l,tm+1), r)
        );
}

void update (int v, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                t[v] = make_pair (new_val, 1);
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}
```

Traženje NZD / NZS

Zadatak je naći NZD/NZS brojeva u datom segmentu. Ovo interesantno uopštenje drveta segmenata dobija se na isti način kao i kod minimuma: dovoljno je u svakom čvoru čuvati NZD /NZS u odgovarajućem segmentu niza.

Izračunavanje broja nula, traženje k-te nule

U ovom zadatku želimo odgovarati na upite o broju nula na datom segmentu i na upit „naći k-tu nulu“.

Promijenimo neznatno podatke koje čuvamo u drvetu: u nizu `t[]` čuvaćemo broj nula u odgovarajućem segmentu. Jasno je da treba promijeniti funkcije `build`, `sum` i `update`, i time je riješen upit o broju nula. Da bi odredili poziciju k-te nule u nizu, spuštamo se po drvetu od korijena i prelazimo u lijevog ili desnog sina u zavisnosti u kom se segmentu nalazi tražena k-ta nula. Dovoljno je pogledati vrijednost u lijevom sinu: ako je ona veća ili jednaka od k, treba preći u lijevog sina, inače preći u desnog sina.

Možemo otkinuti slučaj kada k-ta nula ne postoji, ako kao rezultat funkcija vrati npr. broj -1.

```
int find_kth (int v, int tl, int tr, int k) {
        if (k > t[v])
                return -1;
        if (tl == tr)
```

```
                return tl;
        int tm = (tl + tr) / 2;
        if (t[v*2] >= k)
                return find_kth (v*2, tl, tm, k);
        else
                return find_kth (v*2+1, tm+1, tr, k - t[v*2]);
}
```

Traženje prefiksa niza sa zadatim zbirom

Zadatak je sljedeći: za dato `x` treba naći indeks `i` tako da je zbir prvih i elemenata niza a veći ili jednak od `x`, pri čemu su svi elementi niza nenegativni.

Ovaj zadatak se može riješiti primjenom binarnog traženja. Za tekuću vrijednost i izračunavamo zbir do `i`-tog elementa. Ako je manja, povećavamo i, inače ga smanjujemo. Složenost će biti $O(\log^2 n)$.

Umjesto toga, možemo iskoristii ideju iz prethodnog zadatka i tražiti poziciju jednim traženjem kroz drvo: prelazimo iz čvora lijevo ili desno u zavisnosti od veličine u lijevom sinu. Odgovor će biti jedno traženje po drvetu, što je složenosti $O(\log n)$.

Traženje segmenta sa maksimalnim zbirom

Kao i ranije, na ulazu je dat niz `a[0..n-1]` i upiti tipa `(l,r)` koji imaju sljedeće značenje: naći takav podsegment `a[l1..r1]` takav da je $l \le l1$, $r1 \le r$ i da je zbir na tom podsegmentu maksimalan. Dopušteni su i upiti modifikacije pojedinih elemenata niza. Elementi niza mogu biti i negativni (npr. ako su svi elementi negativni, maksimalni podsegment biće prazan i zbir će biti nula).

Ovo je primjer netrivijalnog uopštenja drveta segmenta. U svakom čvoru čuvamo četiri vrijednosti: zbir na tom segmentu, maksimalni zbir među svim prefiksima tog segmenta, maksimalni zbir među svim sufiksima segmenta i maksimalni zbir na podsegmentu. Drugim riječima, za svaki segment je izračunat odgovor, a dopunski i za sve segmente odgovor za sve segmente koji kreću od lijeve ivice segmenta, kao i za sve segmente koji imaju fiksiranu desnu granicu.

Opet ćemo primjeniti rekurzivni pristup: neka su u tekućem čvoru sve četiri vrijednosti izračunate u lijevom i desnom sinu. Izračunajmo ove vrijednosti za tekući čvor. Za taj čvor je odgovor jednak:

- ili odgovoru iz lijevog sina, što znači da najbolji podsegment u tekućem čvoru u cjelosti pripada segmentu u lijevom sinu
- ili odgovoru u desnom sinu, što znači da najbolji podsegment u tekućem čvoru u cjelosti pripada segmentu u desnom sinu
- ili zbiru maksimalnog sufiksa u lijevom sinu i maksimalnog prefiksa u desnom sinu, što znači da lijevi kraj najboljeg segmenta pripada lijevom sinu a desni kraj najboljeg podsegmenta pripada desnom sinu.

Dakle, odgovor za tekući čvor je maksimum od ove tri veličine. Preračunavanje maksimalne sume na prefiksima i sufiksima je jednostavno. Funkcija `combine` realizuje ova izračunavanja, kojoj se predaju dvije strukture data, koji predstavljaju podatke o lijevom i desnom sinu i koja vraća podatke u tekući čvor.

```
struct data {
	int sum, pref, suff, ans;
};

data combine (data l, data r) {
	data res;
	res.sum = l.sum + r.sum;
	res.pref = max (l.pref, l.sum + r.pref);
	res.suff = max (r.suff, r.sum + l.suff);
	res.ans = max (max (l.ans, r.ans), l.suff + r.pref);
	return res;
}
```

Na taj način smo naučili kako da kreiramo drvo segmenata. Odatle je lako dobiti i implementaciju za upite tipa modifikacije: kao i kod standardnog drveta, preračunavamo vrijednosti u svim čvorovima drveta koji se mijenaju promjenom nekog elementa `a[i]`. Opet koristimo funkciju `combine`, Takođe, za izračunavanje vrijednosti u listovima koristimo pomoćnu funkciju `make_data`, koja vraća strukturu `data` izračunatu iz jednog broja `val`.

```
data make_data (int val) {
	data res;
	res.sum = val;
	res.pref = res.suff = res.ans = max (0, val);
	return res;
}
void build (int a[], int v, int tl, int tr) {
	if (tl == tr)
		t[v] = make_data (a[tl]);
	else {
		int tm = (tl + tr) / 2;
		build (a, v*2, tl, tm);
		build (a, v*2+1, tm+1, tr);
		t[v] = combine (t[v*2], t[v*2+1]);
	}
}
void update (int v, int tl, int tr, int pos, int new_val) {
	if (tl == tr)
		t[v] = make_data (new_val);
	else {
		int tm = (tl + tr) / 2;
		if (pos <= tm)
			update (v*2, tl, tm, pos, new_val);
		else
			update (v*2+1, tm+1, tr, pos, new_val);
		t[v] = combine (t[v*2], t[v*2+1]);
	}
}
```

Ostalo je još realizovati funkciju za upite. Kao i ranije, spuštamo se po drvetu od korijena, razbijajući upit `[l..r]` na više podupita koji se poklapaju sa segmentima iz drveta segmenata. Spajanjem odgovora za sve podsegmente, dobijamo konačni odgovor. Ova funkcija je ista kao za standardno drvo, samo umjesto zbira/maksimuma/minimuma treba

koristiti funkciju `combine`. Postoji još jedna razlika u odnosu na funkciju `sum`: ne dopuštamo slučajeve kada je lijeva granica `l` veća od desne granice `r`.

```
data query (int v, int tl, int tr, int l, int r) {
	if (l == tl && tr == r)
		return t[v];
	int tm = (tl + tr) / 2;
	if (r <= tm)
		return query (v*2, tl, tm, l, r);
	if (l > tm)
		return query (v*2+1, tm+1, tr, l, r);
	return combine (
		query (v*2, tl, tm, l, tm),
		query (v*2+1, tm+1, tr, tm+1, r)
	);
}
```

Čuvanje cijelog podniza u čvoru drveta

Ovo je odvojena sekcija jer u čvoru ne čuvamo neku sažetu informaciju o segmentu (kao što je suma, maksimum, minimum...) već sve elemente niza. U korijenu drveta biće cio niz, u lijevom sinu prva a u desnom druga polovina niza, itd. Najprostija varijanta primjene te tehnike je da se u svakom čvoru drveta čuva sortirana lista svih elemenata u tom čvoru. Kod složenijih varijanti u čvoru umjesto liste možemo koristiti neku složeniju strukturu kao što su set, map, itd. Svi ti metodi imaju jednu zajedničku osobinu: u svakom čvoru drveta segmenata čuvamo neku strukturu koja u sebi sadrži sve elemente iz tog segmenta, a veličina te strukture je proporcionalna dužini segmenta.

Prvo pitanje za ovakvu klasu problema – kolika je količina memorije potrebna? Ako u svakom čvoru drveta čuvamo listu svih brojeva u tom segmentu, tada će drvo zauzimati $O(n \log n)$ memorije, jer svaki `a[i]` ulazi u $O(\log n)$ segmenata (po jedan na svakom nivou drveta). Dakle, uprkos naizgled ekstravagantnosti ove strukture, ona treba samo malo više memorije od običnog drveta segmenata.

U daljem tekstu je opisano nekoliko tipičnih primjena ove strukture. Primjetite direktnu analogiju sa dvodimenzionalnim strukturama podataka (zaista, ovo jeste u nekom smislu dvodimenzionalna struktura podataka sa ograničenim mogućnostima).

Traženje najmanjeg broja većeg ili jednakog zadatom broju na segmentu, bez upita modifikacije

Odgovaramo na upite tipa `(l,r,x)` – naći najmanji broj `x` u segmentu `a[i..r]` koji je veći lii jednak od `x`.

Kreirajmo drvo segmenata u kojem u svakom čvoru čuvamo sortiranu listu svih brojeva iz tog segmenta. Ponovo koristimo rekurziju: neka su za sinove datog čvora napravljene liste; sada samo spojimo te dvije liste u jednu, što možemo uraditi jednim prolaskom kroz liste koristeći dva pokazivača. Ako koristimo C++ biće još prostije, jer je algoritam spajanja (engl. `merge`) uključen u STL:

```
vector<int> t[4*MAXN];
```

```
void build (int a[], int v, int tl, int tr) {
	if (tl == tr)
		t[v] = vector<int> (1, a[tl]);
	else {
	   int tm = (tl + tr) / 2;
	   build (a, v*2, tl, tm);
	  build (a, v*2+1, tm+1, tr);
	   merge (t[v*2].begin(),t[v*2].end(),t[v*2+1].begin(),
t[v*2+1].end(), back_inserter (t[v]));
	}
}
```

Već znamo da će drvo segmenta kreirano na ovaj način zauzimati `O(nlogn)` memorije. Vrijeme kreiranja je takođe `O(nlogn)` jer se svaka lista kreira za vrijeme prorcionalno dužini liste. Može se povući paralela sa mergesort-om, samo što ovdje čuvamo informaciju na svakom nivou spajanja, a ne samo krajnji rezultat.

Sada razmotrimo odgovor na upit. Spuštamo se po drvetu, kao kod standardnog drveta, razbijajući naš segment `a[l..r]` na nekoliko podsegmenata (oko `O(logn)` segmenata). Jasno je da je odgovor minimum od svih odgovora u podsegmentima. Ostalo je još da vidimo kako odgovoriti na upit na jednom podsegmentu koji se poklapa sa čvorom drveta. Kako su liste u čvoru sortirane, najlakše je koristiti binarno traženje po listi u čvoru drveta i vratiti prvi broj iz liste koji je veći ili jednak sa `x`. Dakle, vrijeme odgovora na jedan upit u podsegmnetu je `O(logn)`, pa se čitav upit obradi za $O(\log^2 n)$.

```
int query (int v, int tl, int tr, int l, int r, int x) {
	if (l > r)
		return INF;
	if (l == tl && tr == r) {
		vector<int>::iterator pos = lower_bound (t[v].begin(),
t[v].end(), x);
		if (pos != t[v].end())
			return *pos;
		return INF;
	}
	int tm = (tl + tr) / 2;
	return min (
		query (v*2, tl, tm, l, min(r,tm), x),
		query (v*2+1, tm+1, tr, max(l,tm+1), r, x)
	);
}
```

Konstanta `INF` je jednaka nekom broju koji je veći od svih elemenata u nizu i njen smisao je „ne postoji odgovor u datom segmentu”.

Traženje najmanjeg broja većeg ili jednakog zadatom broju na segmentu, uz upite modifikacije

Postavka je ista kao u prethodnoj sekciji, samo se još dopuštaju upiti dodjeljivanja `a[i]=y`. Ideja rješenja je ista kao u prethodnoj sekciji, ali se umjesto liste u čvoru čuva balansirani spisak koji dozvoljava brzo traženje elementa `a[i]`, njegovo uklanjanje kao i dodavanje

novog broja. Pošto se brojevi u nizu `a` mogu ponavljati, optimalna struktura je `multiset`. Kreiranje drveta je približno isto kao i u prethodnom zadataku, ali zbog spajanja dva multiset-a, složenost raste do $O(n\log^2 n)$. (iako, u teoriji, crveno-crno drvo dozvoljava operaciju spajanja dva drveta za linearnom vrijeme, bilblioteka STL to ne garantuje). Odgovor na upit traženja je praktično isti kao u prethodnom zadatku, samo `lower_bound` treba pozivati od `t[v]`. Da bi odradili upit modifikacije, moramo se spuštati po drvetu, i unijeti izmjene u svih $O(\log n)$ skupova koji sadrže taj element. Prosto udaljimo staru vrijednost elementa (ne zaboravite da nije potrebno udaljiti sva ponavljanja tog elementa).

```
void update (int v, int tl, int tr, int pos, int new_val) {
	t[v].erase (t[v].find (a[pos]));
	t[v].insert (new_val);
	if (tl != tr) {
		int tm = (tl + tr) / 2;
		if (pos <= tm)
			update (v*2, tl, tm, pos, new_val);
		else
			update (v*2+1, tm+1, tr, pos, new_val);
	}
	else
		a[pos] = new_val;
}
```

Složenost obrade ovog upita je $O(\log^2 n)$.

Traženje najmanjeg broja većeg ili jednakog zadatom broju na segmentu, Ubrzanje pomoću tehnike „djelimičnog kaskadnog spusta“

Ubrzajmo vrijeme odgovora na upit traženja na $O(\log n)$ primjenom tehnike „djelimičnog kaskadnog spusta“ (engl. fractional cascading). Ova tehnika je način kako da poboljšamo vrijeme rada nekoliko binarnih traženja koji traže jednu te istu vrijednost. Naš algoritam je razbijao upit na nekoliko podazadataka i na svakom od njih binarnim traženjem tražio x. Tehnika „djelimičnog kaskadnog spusta“ nam dozvoljava da sva ta traženja objedinimo u jedno. Najprostiji i najočigledniji primjer parcijalnog kaskadnog spusta je sljedeći zadatak: dato je nekoliko sortiranih listi i u svakoj od njih tražimo prvi broj koji je veći ili jednak od datog broja x.

Ako bi direkno rješavali zadatak, tada bi za svaku listu pokrenuli binarno traženje. Ako je tih spiskova mnogo (recimo k), tada je složenost $O(k\log(n/k))$, gdje je n ukupna dužina svih listi, a najgori slučaj je kada su liste približno iste dužine tj. n/k.

Primjer 1: date su 4 liste realnih brojeva, pokrećemo binarno traženje za svaku listu

L_1 = 2.4, 6.4, 6.5, 8.0, 9.3
L_2 = 2.3, 2.5, 2.6
L_3 = 1.3, 4.4, 6.2, 6.6
L_4 = 1.1, 3.5, 4.6, 7.9, 8.1

Umjesto toga, možemo objediniti sve liste u jednu sortiranu listu, u kojoj za svaki broj n_i čuvamo spisak pozicija: poziciju u prvoj listi gdje se nalazi broj veći ili jednak od n_i, poziciju u drugoj listi gdje se nalazi broj veći ili jednak od n_i, itd. Drugim riječima, za svaki broj koji se pojavljuje, čuvamo rezulate binarnih traženja po njemu u svakoj od listi. U takvom slučaju,

složenost obrade upita traženja biće `O(logn+k)`, što je znatno bolje nego ranije, iako ćemo imati znatno veći utrošak memorije, jer nam je potrebno `O(nk)` memorije.

Primjer 2: objedinjavanjem listi iz primjera 1, dobijamo listu:

L = `1.1[0,0,0,0], 1.3[0,0,0,1], 2.3[0,0,1,1], 2.4[0,1,1,1], 2.5[1,1,1,1], 2.6[1,2,1,1], 3.5[1,3,1,1], 4.4[1,3,1,2], 4.6[1,3,2,2], 6.2[1,3,2,3], 6.4[1,3,3,3], 6.5[2,3,3,3], 6.6[3,3,3,3], 7.9[3,3,4,3], 8.0[3,3,4,4], 8.1[4,3,4,4], 9.3[4,3,4,5]`

Tehnika „djelimičnog kaskadnog spusta“ ide još dalje, jer omogućava da potrošimo samo `O(n)` memorije uz istu vremensku složenost `O(logn+k)`. Sada opet imamo k listi, ali zajedno sa svakom listom čuvamo svaki drugi element iz sljedeće liste. Sa svakim brojem čuvamo njegovu poziciju u obje liste (tekućoj i sljedećoj). Efikasno možemo odgovoriti na upit: pokrenemo binarno traženje po prvoj listi, a zatim idemo po sljedećim listama prelazeći u svaku sljedeći listu pomoću ranije izračunatih pokazivača i praveći jedan korak ulijevo, i tim samim uzimajući u obzir što smo već odbacili polovinu od sljedeće liste.

Primjer 3: Ilustracija „djelimičnog kaskadnog spusta“ za `O(n)` memorije i `O(logn+k)` vrijeme. Napravimo nove liste M_i. Posljednja lista M_k, jednaka je L_k; svaka prethodna lista M_i se formira spajanjem liste L_i sa svakim drugim elementom iz liste M_{i+1}. Sa svakim brojem u listi čuvamo i par brojeva: poziciju koja je rezultat traženja `x` u L_i i poziciju koja je rezultat traženja `x` u listi M_{i+1}.

M_1 = `2.4[0,1],2.5[1,1],3.5[1,3],6.4[1,5],6.5[2,5],7.9[3,5],8[3,6],9.3[4,6]`
M_2 = `2.3[0, 1], 2.5[1, 1], 2.6[2, 1], 3.5[3, 1], 6.2[3, 3], 7.9[3, 5]`
M_3 = `1.3[0, 1], 3.5[1, 1], 4.4[1, 2], 6.2[2, 3], 6.6[3, 3], 7.9[4, 3]`
M_4 = `1.1[0, 0], 3.5[1, 0], 4.6[2, 0], 7.9[3, 0], 8.1[4, 0]`

U našem zadatku nije neophodna najmoćnija verzija ove tehnike. Zaista, čvor sadrži sve brojeve koji se pojavljuju u lijevim i desnom sinu. Zbog toga, da bi izbjegli binarno traženje po listi sina, dovoljno za svaku listu u drvetu segmenata izračunati za svaki broj u listi njegovu poziciju u listama lijevog i desnog sina (tačnije, poziciju prvog broja manjeg ili jednakog od tekućeg broja). Umjesto obične liste brojeva, imaćemo listu trojki: broj, pozicija u listi lijevog sina, pozicija u listi desnog sina. To nam dozvoljava za `O(1)` odrediti poziciju u listi lijevog ili desnog sina, umjesto binarnog traženja po listi.

Najjednostavnije je ovu tehniku primjenjivati kada nema upita modifikacije. Tada su te pozicije samo brojevi, koje je lako izračunati prilikom kreiranja drveta u toku spajanja dva sortirana niza. Ako su dopušteni upiti tipa modifikacije, algoritam postaje složeniji: pozicije sad treba čuvati u obliku iterarora unutar strukture multiset, a pri upitu modifikacije treba ih pravilno uvećavati ili umanjivati. Zadatak se sada svodi na detalje implementacije, a osnovna ideja zamjene `O(logn)` binarnih traženja jednim traženjem u korijenu drveta, u potpunosti je opisana,

Moguće varijante drveta segmenata

Čuvanje svih elemenata u čvoru podrazumijeva cijelu klasu zadataka koji se mogu rješavati na taj način: sve zavisi od strukture koja odabrana za čuvanje elemenat unutra čvora. U prehodnim sekcijama smo koristili `vector` i `multiset`, ali se može koristiti bilo koja druga struktura: drugo drvo segmenata (kasnije o tekstu biće više riječi o tome, poglavlje „Višedimenzionalno drvo segmenata“), Fenvikovo drvo (posebno poglavlje), Dekartovo drvo (posebno poglavlje), itd.

Promjene (modifikcije) na segmentu

U prethodnim poglavljima smo razmatrali samo upite modifikacije na samo jednom elementu niza. Na drvetu segmenata moguće je imati i upite koji mijenjaju vrijednost svih elemenata na nekom segmentu niza za vrijeme `O(logn)`.

Sabiranje na segmentu

Najprostiji slučaj modifikacije na segmentu je dodavanje broja `x` svim elementima na segmentu `a[l..r]`. Upit čitanja je kao i ranije, traženje vrijednosti nekog broja `a[i]`.

Da bi upit modifikacije na segmentu bio efikasan, u svakom čvoru čuvaćemo koliko treba dodati svakom broju segmenta. Na primjer, ako je potrebno cijelom nizu `a[0..n-1]` dodati 2^n, tada u korijen drveta postavljamo broj 2. Na taj način, možemo upit modifikacije segmenta obraditi efikasno, umjesto da mijenjamo svih `O(n)` vrijednosti. Upit čitanja realizujemo spuštanjem po drvetu, sabirajući sve vrijednosti po putu koje su zapisane u čvorovima.

```
void build (int a[], int v, int tl, int tr) {
	if (tl == tr)
		t[v] = a[tl];
	else {
		int tm = (tl + tr) / 2;
		build (a, v*2, tl, tm);
		build (a, v*2+1, tm+1, tr);
	}
}
void update (int v, int tl, int tr, int l, int r, int add) {
	if (l > r)
		return;
	if (l == tl && tr == r)
		t[v] += add;
	else {
		int tm = (tl + tr) / 2;
		update (v*2, tl, tm, l, min(r,tm), add);
		update (v*2+1, tm+1, tr, max(l,tm+1), r, add);
	}
}
int get (int v, int tl, int tr, int pos) {
	if (tl == tr)
		return t[v];
	int tm = (tl + tr) / 2;
	if (pos <= tm)
		return t[v] + get (v*2, tl, tm, pos);
	else
		return t[v] + get (v*2+1, tm+1, tr, pos);
}
```

Dodjeljivanje na segmentu

Neka je sada upit modifikacije dodjeljivanje vrijednosti `p` nekom segmentu `a[l..r]`. Drugi upit je čitanje vrijednosti `a[i]`.

Da bi dodijelili vrijednost cijelo segmentu, u svakom čvoru drveta čuvamo da li je taj segment u cjelosti obojen nekom bojom p ili ne, i ako jeste, koji je broj u pitanju. Na taj način možemo raditi tzv. „odložene promjene“ na drvetu: pri upitu modifikacije, umjesto promjene vrijednosti u većem broju čvorova drveta, mijenjamo samo neke od njih, postavljajući oznaku „obojen“ za druge segmente što znači da taj segment zajedno sa svim svojim podsegmentima treba da bude obojen u tu boju. Poslije upita modifikacije, drvo postaje, uopšteno govoreći, zastarjelim – u njemu su nepopunjene informacije. Na primjer, ako je upit „cijelom nizu `a[0..n-1]` dodijeliti vrijednost p“, tada u drvetu izvodimo samo jednu promjenu – označimo da je korijen „obojan“ bojom p. Ostali čvorovi drveta su nedirnuti, iako u stvari cijelo drvo treba biti obojano u boju p.

Pretpostavimo da imamo drugi upit modifikacije „segmentu `a[0..n/2]` dodijeliti broj q“. Sada treba cijelog lijevog sina obojiti u boju q, ali prije toga moramo obraditi korijen. Začkoljica je u tome što treba sačuvati činjenicu da je desna polovina drveta i dalje obojana u boju p, a u tom trenutku za desno drvo nikakva informacija nije sačuvana. Zbog toga, moramo progurati informaciju iz korijena tj. moramo istom bojom obojati lijevog i desnog sina a iz korijena ukloniti oznaku o boji. Poslije toga možemo slobodno obojati lijevog sina, ne gubeći nikavu informaciju. Uopštavanjem dobijamo da za svaki upit bilo koje vrste u vrijeme spuštanja po drvetu moramo progurati informacije iz tekućeg čvora ka oba sina. U stvari, mi koristimo odložene modifikacije ali samo u mjeri u kojoj nam je potrebno, da ne bi pogoršali složenost `O(logn)`. Zbog toga je potrebno napsiati funkciju `push`, kojoj se kao argument predaje čvor drveta, i koja progura informaciju od čvora ka sinovima. Funkciju treba pozvati na samom početku funkcije koja obrađuje upite, ali je ne treba pozivati na listovima drveta.

```
void push (int v) {
	if (t[v] != -1) {
		t[v*2] = t[v*2+1] = t[v];
		t[v] = -1;
	}
}

void update (int v, int tl, int tr, int l, int r, int color) {
	if (l > r)
		return;
	if (l == tl && tr == r)
		t[v] = color;
	else {
		push (v);
		int tm = (tl + tr) / 2;
		update (v*2, tl, tm, l, min(r,tm), color);
		update (v*2+1, tm+1, tr, max(l,tm+1), r, color);
	}
}

int get (int v, int tl, int tr, int pos) {
	if (tl == tr)
		return t[v];
	push (v);
	int tm = (tl + tr) / 2;
	if (pos <= tm)
		return get (v*2, tl, tm, pos);
```

```
        else
                return get (v*2+1, tm+1, tr, pos);
}
```

Funkciju `get` možemo realizovati i na drugi način: umjesto odloženih modifikacija odmah vraćamo odgovor čim funkcija uđe u čvor koji je u cjelosti označen jednom bojom.

Sabiranje na segmentu, upit maksimuma

Neka je upit modifikacije opte sabiranje na segmentu a upit čitanja vraća maksimum na nekom segmentu. Jasno je da u čvoru moramo čuvati maksimum na segmentu. Suptilnost u zadataku je kako preračunavati te vrijednosti. Npr, ako imamo upit „prvoj polovini niza a dodati broj 2^n“, tada broj 2 upisujemo u lijevi sin korijena. Kako odrediti maksimum u lijevom sinu i korijenu? Važno je da ne dođe do zabune koji maksimum čuvamo u čvoru drveta: onaj sa već dodatim brojem ili onaj bez dodatog broja. Važno je izabrati jedan od ta dva i pridržavati ga se kroz cijelo drvo. Ako izaberemo prvi način, maksimum u korijenu je veći od maksimuma u lijevom i desnom sinu kome je dodat sabirak za korijen. Ako izaberemo drugi način, tada odredimo maksimum u lijevom sinu plus sabirak za lijevog sina i maksimum u desnom sinu plus sabirak iz desnog sina: veći od ta dva zbira biće maksimum u korijenu.

Moguće varijante

Razmotrili smo samo osnovne primjene drveta segmenata sa modifikacijama na segmentu. Ostali zadaci se dobijaju na isti način kao što je opisano ovdje. Samo moramo voditi računa da precizno sprovedemo odložene modifikacije. Čak i ako smo u tekućem čvoru „progurali“ odloženu modifikaciju, to skoro sigurno nije urađeno u lijevom i desnom sinu. Zbog toga je često neophodno pozvati funkciju `push` za lijevog i desnog sina tekućeg čvora ili precizno uzimati u obzir odložene modifikacije u njima.

Uopštenje na višedimenzionalni slučaj

Drvo segmenata se lako uopštava na dvodimenzionalni (i uopšte višedimenzionalni) slučaj. Za 2D slučaj, prvo razbijamo prvi indeks po segmentima, a za svaki takav segment kreiramo drvo segmenata po drugom indeksu. Dakle, osnovna ideja je ugnježdavanje drveta po drugom indeksu unutar drveta po prvom indeksu.

Pojasnimo ideju na primjeru konkretnog zadatka

2D drvo segmenata u najprostijoj verziji

Data je pravougaona matrica `a[0..n-1][0..m-1]`. Postoje upiti tipa „naći zbir (ili minimum ili maksimum) u pravougaoniku `a[x1..x2][y1..y2]`“ i upiti tipa „`a[x][y]=p`“.

Na trenutak zaboravimo da je niz a 2D i posmatrajmo samo prvu koordinatu (tj. prvi indeks). Kreiramo obično drvo segmenata po prvom indeksu. Vrijednost `i` svakom čvoru neće biti jedan broj već cijelo drvo segmenata, jer za fiksirani segment `[l..r]` prvog indeksa, faktički imamo podmatricu `a[l..r][0..m-1]` i za nju kreiramo drvo segmenata. Sama implementacija operacije kreiranja drveta sastoji se od dvije funkcije, po jedne za svaku koordinatu: `build_x` i `build_y`. Prva funkcija se ni po čemu ne razlikuje od 1D slučaja, ali zato `build_y` mora odvojeno razmatrati dva slučaja: kada tekući segment `[tlx..trx]` po prvom indeksu ima dužinu 1 i kada mu je dužina veća od 1. U prvom slučaju prosto uzmemo

traženu vrijednost iz matrice `a[][]`, a u drugom vršimo spajanje dva drveta segmenata iz lijevog sina i desnog sina po koordinati `x`.

```
void build_y (int vx, int lx, int rx, int vy, int ly, int ry) {
	if (ly == ry)
		if (lx == rx)
			t[vx][vy] = a[lx][ly];
		else
			t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
	else {
		int my = (ly + ry) / 2;
		build_y (vx, lx, rx, vy*2, ly, my);
		build_y (vx, lx, rx, vy*2+1, my+1, ry);
		t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
	}
}

void build_x (int vx, int lx, int rx) {
	if (lx != rx) {
		int mx = (lx + rx) / 2;
		build_x (vx*2, lx, mx);
		build_x (vx*2+1, mx+1, rx);
	}
	build_y (vx, lx, rx, 1, 0, m-1);
}
```

Takvo drvo zauzima linearnu količinu memorije ali sada sa većom konstanom – `16mn`. Kreira se opisanom funkcijom `build_x` za linearno vrijeme.

Odgovor na upit realizujemo po istom principu: razbijamo upit po prvoj koordinati a kada dođemo do nekog čvora drveta po prvoj koordinati pozivamo upit na odgovarajućem drvetu po drugoj koordinati.

```
int sum_y (int vx, int vy, int tly, int try_, int ly, int ry) {
	if (ly > ry)
		return 0;
	if (ly == tly && try_ == ry)
		return t[vx][vy];
	int tmy = (tly + try_) / 2;
	return sum_y (vx, vy*2, tly, tmy, ly, min(ry,tmy))
		+ sum_y (vx, vy*2+1, tmy+1, try_, max(ly,tmy+1), ry);
}

int sum_x (int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
	if (lx > rx)
		return 0;
	if (lx == tlx && trx == rx)
		return sum_y (vx, 1, 0, m-1, ly, ry);
	int tmx = (tlx + trx) / 2;
	return sum_x (vx*2, tlx, tmx, lx, min(rx,tmx), ly, ry)
		+ sum_x (vx*2+1, tmx+1, trx, max(lx,tmx+1), rx, ly, ry);
}
```

Vrijeme rada funkcije je $O(\log n \cdot \log m)$, jer se spušta po drvetu po prvoj koordinati a zatim isto to radi po drvetu za drugu koordinatu.

Ramotrimo upit modifikacije a[x][y]=p. Jasno je da do promjena dolazi za segmente koji sadrže koordinatu x (a ima ih $O(\log n)$) a za drva segmenata koji im odgovaraju mijenajće se ona koja sadrže koordinatu y (takvih ima $O(\log m)$). Nema značajne razlike u odnosu na 1D slučaj, osim što se prvo spuštamo po prvoj koordinati a zatim po drugoj.

```
void update_y (int vx, int lx, int rx, int vy, int ly, int ry, int x, int
y, int new_val) {
	if (ly == ry) {
		if (lx == rx)
			t[vx][vy] = new_val;
		else
			t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
	}
	else {
		int my = (ly + ry) / 2;
		if (y <= my)
			update_y (vx, lx, rx, vy*2, ly, my, x, y, new_val);
		else
			update_y (vx, lx, rx, vy*2+1, my+1, ry, x, y,
new_val);
		t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
	}
}

void update_x (int vx, int lx, int rx, int x, int y, int new_val) {
	if (lx != rx) {
		int mx = (lx + rx) / 2;
		if (x <= mx)
			update_x (vx*2, lx, mx, x, y, new_val);
		else
			update_x (vx*2+1, mx+1, rx, x, y, new_val);
	}
	update_y (vx, lx, rx, 1, 0, m-1, x, y, new_val);
}
```

Kompresija 2D drveta segmenata

Dat je sljedeći zadatak: dato je n tačaka u ravni, zadati koordinatama (x_i, y_i) i dobijamo upite tipa „koliko tačaka leži u pravougaoniku određenom tačkama (x_1, y_1), (x_2, y_2)“. Jasno je da u takvom slučaju nije potrebno kreirati 2D drvo segmenata sa $O(n^2)$ elemenata. Najveći dio te memorije biće protraćen, jer svaka tačka pojedinačno može upasti u $O(\log n)$ segmenata drveta po prvoj koordinati, a to znači da će zbirna veličina svih drveta po drugoj koordinati biti $O(n \log n)$. Zbog toga uradimo sljedeće: u svakom čvoru drveta po prvoj koordinati čuvaćemo drvo segmenata kreirano samo po onim drugim koordinatama koje se pojavljuju u segmentima prvih koordinata. Drugim riječima, pri kreiranju drveta segmenata u nekom čvoru vx i granicama tlx i trx, posmatraćemo samo one tačke koji ulaze u taj segment i kreirati drvo segmenata nad njima. Prednost takvog pristupa je što će svako drvo segmenata po drugoj koordinati zauzimati tačno onoliko memorije koliko treba. Na taj način

će ukupna količina memorije biti $O(n \log n)$. Odgovor na upite će biti $O(\log^2 n)$, jer pri pozivu upita po drugoj koordinati moramo pokrenuti binarno traženje po toj koordinati, ali to ne pogoršava složenost.

Na žalost, više ne možemo izvoditi proizvoljne upite modifikacije: ako se pojavi nova tačka, mi u nekom drvetu segmenata po drugoj koordinati moramo dodati element u sredinu, što nije moguće efikasno uraditi.

Za kraj, primjetimo da kompresovano drvo segmenata ekvivalentno sa modifikacijom 1D drveta segmenata opisanog u poglavlju „Čuvanje cijelog podniza u čvoru drveta“. Ovdje opisano drvo je samo poseban slučaj čuvanja podniza u svakom čvoru drveta, gdje se sam podniz čuva u obliku drveta segmenata. Ako moramo odustati od 2D drveta segmenata zbog toga što ne možemo odgovoriti ne neki tip upita, možemo zamijeniti ugnježdena drveta segmenata nekom „moćnijom“ strukturom kao što je npr. Dekartovo drvo.

Drvo segmenata sa istorijom vrijednosti

Perzistentna struktira podataka je ona koja pri svakoj modifikaciji zapamti svoje prethodno stanje. To nam dozvoljava da po potrebi pristupimo bilo kojoj verziji strukture i odradimo upit na njoj.

Drvo segmenata je struktura koja se može pretvoriti perzistentnu strukturu. Naravnom razmotrićemo efikasnu perzistentnu strukturu, a ne oni koju kopira samu sebe u potpunosti pri modifikaciji. Svaki upit modifikacije dovodi do promjena u $O(\log n)$ čvorova, od korijena naniže. Znači, ako su pokazivači na lijevog i desnog sina pravi pokazivači (pointeri) koje čuvamo u čvoru, tada pri upitu modifikacije umjesto čvorova koje mijenjamo kreiramo nove čvorove i pokazivače iz njih usmjeravamo u stare čvorove. Dakle, dobijamo $O(\log n)$ novih čvorova i među njima novi korijen. Čitava prethodna verzija drveta ostaje bez promjena prelomljena za stari korijen.

```
struct vertex {
	vertex * l, * r;
	int sum;

	vertex (int val)
		: l(NULL), r(NULL), sum(val)
	{ }

	vertex (vertex * l, vertex * r)
		: l(l), r(r), sum(0)
	{
		if (l)  sum += l->sum;
		if (r)  sum += r->sum;
	}
};

vertex * build (int a[], int tl, int tr) {
	if (tl == tr)
		return new vertex (a[tl]);
	int tm = (tl + tr) / 2;
	return new vertex (
		build (a, tl, tm),
```

```
                build (a, tm+1, tr)
        );
}

int get_sum (vertex * t, int tl, int tr, int l, int r) {
        if (l > r)
                return 0;
        if (l == tl && tr == r)
                return t->sum;
        int tm = (tl + tr) / 2;
        return get_sum (t->l, tl, tm, l, min(r,tm))
                + get_sum (t->r, tm+1, tr, max(l,tm+1), r);
}

vertex * update (vertex * t, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                return new vertex (new_val);
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return new vertex (
                                update (t->l, tl, tm, pos, new_val),
                                t->r
                        );
        else
                return new vertex (
                                t->l,
                                update (t->r, tm+1, tr, pos, new_val)
                        );
}
```

Na ovaj način možemo praktično svako drvo segmenata pretvoriti u perzistentnu strukturu podataka.