Segmentno stablo

Jelena Hadzi-Puric

Februar 2021

1 Uvod - upit raspona, range query

U ovoj temi, predstavljamo strukture podataka koje dozvoljavaju efikasno obradu upita nad rasponima (intervalima) nizova. Tipični range query, tj. upiti raspona su vezani za izracunavanje vrednostu unutar podnizova. Na primer:

- $\texttt{sum}_q(a, b)$: izračunaj sumu vrednosti intervala $[a, b]$
- $\texttt{min}_q(a, b)$: nadji minimum intervala $[a, b]$
- $\texttt{max}_q(a, b)$: nadji maximum intervala $[a, b]$

Na primer, razmotrimo interval $[3, 6]$ unutar niza:

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

U ovom primeru, $\texttt{sum}_q(3, 6) = 14$, $\texttt{min}_q(3, 6) = 1$ and $\texttt{max}_q(3, 6) = 6$.

Jednostavan način obrade upita raspona je upotreba petlje koja prolazi kroz sve vrednosti niza u opsegu. Na primer, može se sledeća funkcija koristiti za obradu upita za zbir u nizu:

```
int sum(int a, int b) {
    int s = 0;
    for (int i = a; i <= b; i++) {
        s += array[i];
    }
    return s;
}
```

Ova funkcija radi u $O(n)$ vremenskoj složenosti, gde n je dimenzija niza. Dakle, možemo obraditi q upita u $O(nq)$ vremenskoj slozenosti upotrebom ove funkcije. Medjutim, ako su i n i q veliki, ovaj pristup je spor. Srećom, ispostavilo se da ima načine za mnogo efikasniju obradu upita raspona.

2 Upiti nad statičkim nizovima

Razmotrimo slučaj *statičkog* niza (čije vrednosti se ne menjaju izmedju dva upita). Tada je dovoljno konstruisati statičku strukturu podataka koja se unapred obradjuje i na taj način efikasno odgovara na upite.

2.0.1 Obrada upita sume

Možemo jednostavno obraditi upite sume za statičke nizove korišćenjem prefix sum array. Svaka vrednost u nizu prefix suma je jednaka sumi vrednosti originalnog niza do odgovarajuće pozicije, npr., vrednost na poziciji k niza prefiksnih suma je $\texttt{sum}_q(0, k)$. Niz prefix suma se konstruiše u vremenskoj složenosti $O(n)$.

Na primer, razmotrimo niz:

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

Odgovarajući niz prefix suma je:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Kako niz prefix sum sadrži sve vrednosti of $\texttt{sum}_q(0, k)$, možemo izračunati ma koju vrednost $\texttt{sum}_q(a, b)$ u $O(1)$ vremenskoj složenosti:

$$\texttt{sum}_q(a, b) = \texttt{sum}_q(0, b) - \texttt{sum}_q(0, a - 1)$$

Ako definisemo $\texttt{sum}_q(0, -1) = 0$, onda gornja formula vazi i kada $a = 0$.

Na primer, razmotrimo interval $[3, 6]$:

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

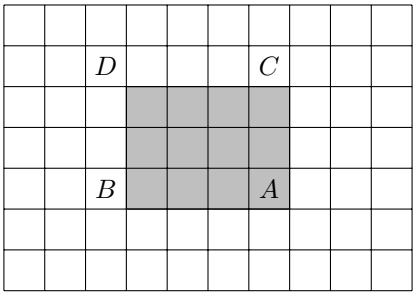
U ovom slucaju $\texttt{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$. Ova suma se moze izracunati na osnovu dve vrednosti iz niza prefix sum:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Thus, $\texttt{sum}_q(3, 6) = \texttt{sum}_q(0, 6) - \texttt{sum}_q(0, 2) = 27 - 8 = 19$.

Ova ideja moze da se uopsti i prenese na visedimnzioni domen. Na primer, mzoemo konstruisati dvo-dimenzionalni niz prefix sum koji se moze koristiti za izracunavanje sume bilo koje podmatrice u $O(1)$ vremenskoj slozenosti. Svaka suma u ovom nizu odgovara podnizu koji pocinje u gornjem levom uglu niza.

Pogledajmi ideju koju ilustruje skica:



Suma sivog podniza se moze izracunati koriscenjem formule

$$S(A) - S(B) - S(C) + S(D),$$

gde $S(X)$ oznacava sumu vrednosti podmatrice od levog gornjeg ugla do pozicije X.

2.0.2 Upiti minimuma raspona - algoritam retke tabele, sparse table

Upiti minimuma su kompleksniji za obradu negoli upiti sume. Ipak, postoji veoma jednostavan metod predobrade u vremenskoj slozenosti $O(n \log n)$ koji pomaze da se razresi upit minimuma u vremenskoj slozenosti $O(1)$ time[1]. Imajte na umu da budući da mogu upiti minimuma i maksimuma biti obradjeni na sličan način, možemo se u ovoj lekciji baviti samo upitima minimuma.

Ideja je da se preračunaju sve vrednosti $\min_q(a, b)$ gde $b - a + 1$ (duzina raspona) je stepen dvojke. Na primer, za niz

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

izračunavaju se sledeće vrednosti:

a	b	$\texttt{min}_q(a, b)$
0	0	1
1	1	3
2	2	4
3	3	8
4	4	6
5	5	1
6	6	4
7	7	2

a	b	$\texttt{min}_q(a, b)$
0	1	1
1	2	3
2	3	4
3	4	6
4	5	1
5	6	1
6	7	2

a	b	$\texttt{min}_q(a, b)$
0	3	1
1	4	3
2	5	1
3	6	1
4	7	1
0	7	1

[1]Ova tehnika je uvedena 2000. i ponekad se zove algoritam retkih matrica(tabela). Postoje i sofisticiranije tehnike gde vreme predobrade je samo $O(n)$, ali takvi algoritmi nisu potrebni u takmičarskom programiranju.

Broj preračunatih vrednosti je $O(n \log n)$, jer postoji $O(\log n)$ raspona cije su dužine stepen dvojke.

Vrednosti se efikasno izračunavaju korišćenjem sledeće rekurzivne formule

$$\texttt{min}_q(a, b) = \min(\texttt{min}_q(a, a + w - 1), \texttt{min}_q(a + w, b)),$$

gde $b - a + 1$ je stepen dvojke i važi da $w = (b - a + 1)/2$. Izračunavanje svih ovih vrednostri zahteva $O(n \log n)$ vremensku složenost.

After this, any value of $\texttt{min}_q(a, b)$ can be calculated in $O(1)$ time as a minimum of two precalculated values. Let k be the largest power of two that does not exceed $b - a + 1$. We can calculate the value of $\texttt{min}_q(a, b)$ using the formula

$$\texttt{min}_q(a, b) = \min(\texttt{min}_q(a, a + k - 1), \texttt{min}_q(b - k + 1, b)).$$

In the above formula, the range $[a, b]$ is represented as the union of the ranges $[a, a + k - 1]$ and $[b - k + 1, b]$, both of length k.

As an example, consider the range $[1, 6]$:

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

The length of the range is 6, and the largest power of two that does not exceed 6 is 4. Thus the range $[1, 6]$ is the union of the ranges $[1, 4]$ and $[3, 6]$:

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

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

Since $\texttt{min}_q(1, 4) = 3$ and $\texttt{min}_q(3, 6) = 1$, we conclude that $\texttt{min}_q(1, 6) = 1$.

3 Segment tree, stablo segmenata

Segment tree[2] je struktura podataka koja efikasno podrzava i obezbedjuje rad sa bar dve operacije: obrada upita raspona i azuriranje(izmena) vrednosti niza. Segmentni stablo moze da obezbedi rad sa upitima sume, maksimuma, minimuma,...tako da obe operacije se obave u $O(\log n)$ vremenskoj slozenosti.

U poredjenju sa slicnim strukturama (binary indexed tree, Fenwick stablo, turnirsko, wavelet,...) prednost segmentnog stabla je sto je uopstenija struktura ako se pogleda primena na upite raspona. Zapravo, binary indexed trees efikasno

[2]Implementacija bottom-up koju predstavljamo je po ugledu na resenje TOP coder izazova iz 2006. Inace, slicna struktura je koriscena kasnih 70-ih da se napadnu problemi (racunarske) geometrij.

podrzava upite sume [3], ali segment stablo se koristi i za mnoge druge upite. Ali, segmentno stablo zahteva vise memorije i pazljivo se mora razmotriti efikasna implementacija u konkretnom programskom jeziku.

3.0.1 Struktura

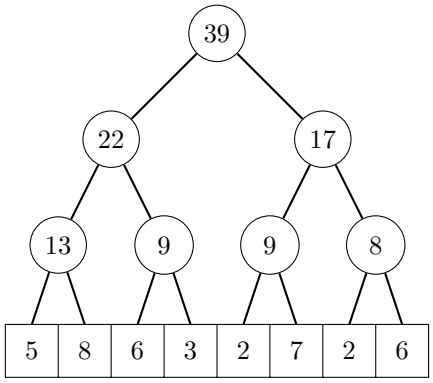
Segmentno stablo je binarno stablo takvo da cvorovi nivoa na dnu stabla odgovaraju elementima polaznog niza, dok ostali cvorovi sadrze informacije neophodne za obradu upita raspona (range queries).

U uvodnoj predstavu, pretpostavljamo de je velicina polaznog niza stepen dvojke o koristimo indeksiranje niza od 0. Ako velicina niza nije stepen dvojke, uvek u njega mozemo da ubacimo dodatne elemente, neutralne u odnosu na operaciju koju koristimo.

Uvedimo segmentna stabla koji obradjuju upite sume. Razmotrimo niz:

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

Odgovarajuce segmentno stablo:



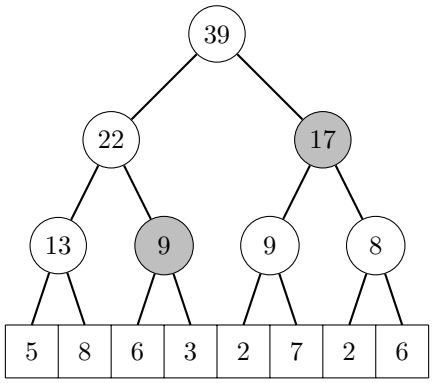
Svaki unutrasnji cvor stabla odgovara opsegu(rasponu, intervalu) niza cija je velicina jednaka stepenu dvojke. Kada stablo segmenata podrzava upit sume, vrednost svakog unutrasnjeg cvora predstavlja sumu odgovarajucih elemenata niza, i izaracunava se kao suma levog i desnog deteta cvora.

Ispostavlja se da svaki raspon $[a, b]$ spode moze podeliti na $O(\log n)$ podraspona cije vrednosti se cuvaju u cvorovima stabla. Na primer, analizirajmo raspon [2,7]:

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

[3]jer, upotrebom *dva* binary indexed trees je moguce podrzati rad sa upitima minimuma, ali implementacija moze biti kompleksnija u poredjenju sa upotrebom segment stabla.

Vidimo da $\texttt{sum}_q(2,7) = 6+3+2+7+2+6 = 26$. Dakle, sledeci cvorovi stabla odgovaraju rasponu:



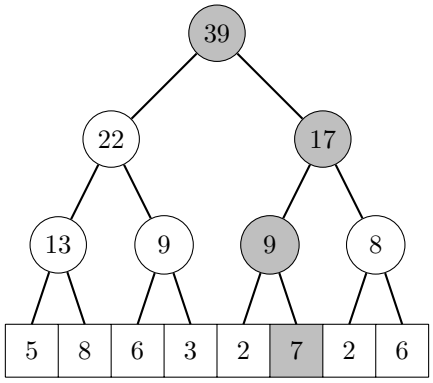
Dakle, drugi nacin da se izaracuna suma je $9 + 17 = 26$.

Kada se suma izracunava koriscenjem cvorova koji su na stablu pozicionarini NAJVISE moguce, na svakom nivou stabla NEOPHODNA SU NAJVISE DVA cvora. Otuda je ukupan broj cvorova koji se koristi $O(\log n)$.

Nakon azuriranja niza, moraju se azurirati i svi cvorovi cija vrednost zavisi od azurirane vrednosti.

After an array update, we should update all nodes whose value depends on the updated value. This can be done by traversing the path from the updated array element to the top node and updating the nodes along the path.

Primer: The following picture shows which tree nodes change if the array value 7 changes:



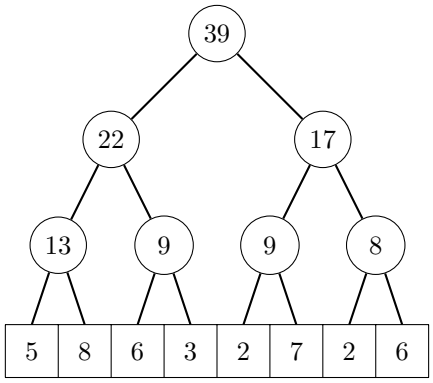
The path from bottom to top always consists of $O(\log n)$ nodes, so each update changes $O(\log n)$ nodes in the tree.

3.0.2 Implementacija

We store a segment tree as an array of $2n$ elements where n is the size of the original array and a power of two. The tree nodes are stored from top to bottom:

`tree`[1] is the top node, `tree`[2] and `tree`[3] are its children, and so on. Finally, the values from `tree`[n] to `tree`[$2n-1$] correspond to the values of the original array on the bottom level of the tree.

For example, the segment tree



is stored as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Using this representation, the parent of `tree`[k] is `tree`[$\lfloor k/2 \rfloor$], and its children are `tree`[$2k$] and `tree`[$2k+1$]. Note that this implies that the position of a node is even if it is a left child and odd if it is a right child.

The following function calculates the value of $\texttt{sum}_q(a,b)$:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

The function maintains a range that is initially $[a+n, b+n]$. Then, at each step, the range is moved one level higher in the tree, and before that, the values of the nodes that do not belong to the higher range are added to the sum.

The following function increases the array value at position k by x:

```
void add(int k, int x) {
    k += n;
    tree[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
```

```
        tree[k] = tree[2*k]+tree[2*k+1];
    }
}
```

First the function updates the value at the bottom level of the tree. After this, the function updates the values of all internal tree nodes, until it reaches the top node of the tree.

Both the above functions work in $O(\log n)$ time, because a segment tree of n elements consists of $O(\log n)$ levels, and the functions move one level higher in the tree at each step.