

Logaritamska struktura i tournament stablo

Logaritamska struktura i tournament stablo su strukture podataka. Za razliku od npr. heapa, binary search stabala ili hash tablice, ove strukture služe da nam efikasno odgovaraju na razne upite o svojstvima nekog **niza** X. Npr. Kolika je suma elemenata $X_3..X_8$? Ili koji je najveći element među $X_1..X_{12345}$?
U čemu je razlika između te dvije strukture?

Logaritamska struktura

Prvo nešto malo o izvedbi logaritamske strukture:

Elementi su numerirani $1..N$, a ne $0..N-1$

Niz X uopće ne postoji u memoriji već samo niz A, sa svojstvom:

$$A_i = \text{suma od } X_{i-\text{lobit}(i)+1} \text{ do } X_i,$$

Gdje je lobit(i) jednak najvećoj potenciji broja 2 s kojim je broj i dijeljiv.

Tako je npr.

$$A_{13} = X_{13}$$

$$A_{12} = X_9 + X_{10} + X_{11} + X_{12}$$

$$A_8 = X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + X_8$$

Na taj način možemo sumu X_1 do X_{13} naći kao zbroj $A_8 + A_{12} + A_{13}$, a općenito sumu X_a do X_b kao sumu X_1 do X_b umanjenu za sumu X_1 do X_{a-1} .

Jedan mali trik... lobit(i) se može u konstantnom vremenu izračunati kao $i \& -i$

Evo source-a:

```
struct logaritamska {
    int n;
    vector<int> a;

    logaritamska( int N ) {
        n = N;
        a.resize( n+1 );
        for( int i = 0; i <= n; ++i ) a[i] = 0;
    }

    int get( int lo, int hi ) {
        int ret = 0;
        for(      ; hi > 0; hi -= hi&-hi ) ret += a[hi];
        for( --lo; lo > 0; lo -= lo&-lo ) ret -= a[lo];
        return ret;
    }

    void set( int i, int value ) {
        for( value -= get( i, i ); i <= n; i += i&-i ) a[i] += value;
    }
};
```

Primjetite da kod funkcije set, vrijednost na koju želimo postaviti X_i prvo umanjujemo za trenutnu vrijednost X_i jer odgovarajuća polja u nizu A treba povećati samo za tu razliku. Nažalost, kako smo rekli da ta struktura ne pamti sam niz X, X_i moramo tražiti funkcijom

get koja je složenosti $O(\log_2 N)$. U praksi to se rijetko tako radi. Ili se dodatno pamti i niz X ili znamo da za neki indeks u nizu nećemo više puta pozivati funkciju set, pa se ne moramo brinuti o tome.

Logaritamska struktura isto tako može pamtit minimum ili maksimum u intervalu $[1, i]$ međutim tu postoje određena ograničenja.

Kao prvo, ne možemo općenito naći minimum(a, b), jer ne postoji nikakva relacija koja povezuje minimum(a, b) s minimum($1, a$) i minimum($1, b$).

Kao drugo, ako pamtimo minimum onda ne smijemo niti u jednom trenutku povećati vrijenost nekog elementa u nizu X . Zašto?

Pogledajmo npr. A_{24} . On pamti minimum elemenata $X_{17} .. X_{24}$. Neka je to X_{20} . I sada... Povećavanjem elementa X_{20} , više ne znamo je li on i dalje najmanji među elementima $X_{17} .. X_{24}$, tako da bi trebalo proći sve elemente i ponovo naći najmanjeg među njima. No time ta operacija više ne radi u logaritamskom vremenu. Pa ni strukturu više ne bismo mogli nazvati logaritamskom \square

Dosta o logaritamskoj...

Tournament stablo

Puno moćnija struktura, može sve što može i logaritamska, a i puno više. Konstrukcija tournament stabla je dosta jednostavna za razliku od logaritamske.

Članovi niza X nalaze se kao listovi u potpunom binarnom stablu. Stoga ako je N veličina niza X . N prvo povećamo na prvu potenciju broja 2 veću ili jednaku od N .

Korijen tog stabla označavamo brojem 1. A djeca svakog čvora t , označena su brojevima $2*t$ i $2*t+1$.

Dakle, iz svega toga slijedi da su elementi niza X označeni su brojevima N do $2*N-1$ unutar stabla.

I sad... ako stablo koristimo za traženje sume, onda svaki čvor u stablu pamti sumu elemenata niza unutar njegovog podstabla. Ako stablo koristimo za traženje minimuma ili maksimuma, onda svaki čvor u stablu pamti minimum odnosno maksimum elemenata niza unutar njegova podstabla.

Implementacija:

```

const int inf = 1000000000;

class tournament {
private:
    struct data {
        int sum;
        int minimum;
        int maximum;
        data() { sum = 0; minimum = inf; maximum = -inf; }
    };
    int offset;
    vector<data> a;

    int from, to;
    int sum( int i, int lo, int hi ) {
        if( from >= hi || to <= lo ) return 0;
        if( lo >= from && hi <= to ) return a[i].sum;
        return sum( 2*i, lo, (lo+hi)/2 ) + sum( 2*i+1, (lo+hi)/2, hi );
    }
    int min( int i, int lo, int hi ) {
        if( from >= hi || to <= lo ) return inf;
        if( lo >= from && hi <= to ) return a[i].minimum;
        return min( 2*i, lo, (lo+hi)/2 ) <? min( 2*i+1, (lo+hi)/2, hi );
    }
    int max( int i, int lo, int hi ) {
        if( from >= hi || to <= lo ) return -inf;
        if( lo >= from && hi <= to ) return a[i].maximum;
        return max( 2*i, lo, (lo+hi)/2 ) >? max( 2*i+1, (lo+hi)/2, hi );
    }
}

public:
    tournament( int N ) {
        for( offset = 1; offset < N; offset *= 2 );
        a.resize( 2*offset );
    }

    int operator () ( int i ) { return a[offset+i].sum; } // daje X[i]

    void set( int i, const int value ) { // postavlja X[i] na value
        i += offset;
        a[i].sum = a[i].minimum = a[i].maximum = value;

        for( i /= 2; i > 0; i /= 2 ) {
            a[i].sum = a[2*i].sum + a[2*i+1].sum;
            a[i].minimum = a[2*i].minimum <? a[2*i+1].minimum;
            a[i].maximum = a[2*i].maximum >? a[2*i+1].maximum;
        }
    }
    // trazi sumu elemenata [lo, hi>
    int sum( int lo, int hi ) { from = lo; to = hi; return sum( 1, 0, offset ); }
    int min( int lo, int hi ) { from = lo; to = hi; return min( 1, 0, offset ); }
    int max( int lo, int hi ) { from = lo; to = hi; return max( 1, 0, offset ); }
};

```

Na prvi pogled može izgledati jako komplicirano, ali nije. Ova implementacija pamti i sumu i minimum i maksimum.

Usredotočimo se na funkciju set. Koja postavlja X_i na neku vrijednost. Prvo treba broj i povećati za offset (odnosno prvu potenciju broja 2 veću od N). I nakon što postavimo X_i treba osvježiti njegove roditelje s novim vrijednostima.

Funkcija get (odnosno sum, min, ili max) je nešto kompliciranija. Tražimo sumu u intervalu $[from, to>$. Prvo pozivamo $sum(i=1, lo=0, hi=offset)$.

Varijable i , lo i hi znače da se trenutno nalazimo u čvoru i koji pamti sumu elemenata iz intervala $interval [lo, hi>$.

Prvi if: `if(from >= hi || to <= lo) return 0;`

gleda jesu li intervali $[from, to>$ i $[lo, hi>$ disjunktni, ako jesu onda ne moramo dalje tražiti i vraćamo nulu.

Drugi if: `if(lo >= from && hi <= to) return a[i].sum;`

gleda je li interval $[lo, hi>$ potpuno unutar intervala $[from, to>$, te ako je onda vratimo sumu cijelog tog intervala.

Ako nijedan if nije zadovoljen, znači da se intervali djelomično preklapaju, pa dijelimo interval $[lo, hi>$ popola, i za svaki zovemo rekurziju.

Osim za traženje minimuma, maksimuma i sume niza. Ovo stablo može se koristiti za odgovoriti na još puno raznih upita. Npr. koji je najdesniji element u intervalu koji nije 0.

Luka Kalinovčić