7th September 2016

SEGMENT TREE (Range Update,Range queries,Lazy Propagation)

Where to use?

A list of random integers is given and following operations are to be done.

- Update any value at index i (list[i]=new_value).
- Update a Range (from index=L to R ,increment every element by a given value).
- Find max(or sum,gcd,lcm..etc) of a given range.

We can do all above operations in O(N) time simply using a loop and yes first operation in constant time but what if there are a lot of queries to perform.(Like if we have q queries of above types than the above approach will take O(N*q) time(we need something faster). Using Segment Tree we can perform all above operation in O(Log(N)).Wow that's impressive, isn't it?Let's see how does it work.

Implementation

It takes O(N log N) time to build segment tree.Once tree is built we can do queries in O(log N) time.Segment tree is a complete binary tree so we can store it in an array efficiently.

And yes it takes O(N) extra space.

extra space required = (2*x-1).

where

x=2^(ceil(log2 (N))).

Don't worry you will get it soon.

Let there is a list of 8 numbers-

list[]={ 5, 2, 4, 6, 11, 8, 3, 2 }.

and we have to perform several operations of following two types-

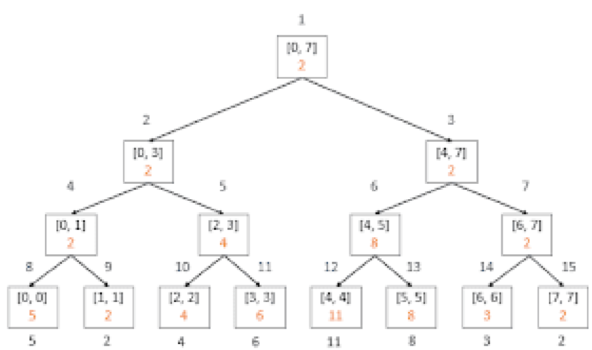
1. update element at index i.
2. find minimum of a given interval [L....R].

See the image below it represents a segment tree built to perform above two types of operations efficiently (O(Log N)) for the above list.Seeing the image,you can clearly notice some points as-

- There are N leaves in the tree each having an element of the list.
- There are (N-1) internal nodes each holding minimum element of some interval.
- Root represents the minimum element of the entire list.

Now see root represents minimum of the entire list or we can say minimum of range (0-7), now we will break this interval into two equal intervals and store the answer to the left half interval in the left child of root and answer to the other half interval in the right child of the list.In this way we keep breaking the intervals until the interval represents only one element so these nodes become the leaf nodes of segment tree and holds list elements (because min of range (1,1) will be list[1] as there is only one element in the list).

Firstly we initialize the elements of tree array to zero and recursively update the answers of intervals in internal nodes of the tree.We do it in bottom up manner because to efficiently calculate the answer for a node(range) we need to compare the answers of its children nodes.Thus to construct segment tree it requires (O(N Log N)) time (Log N is the height of tree).



[https://2.bp.blogspot.com/-iR8ShymJLu8/V9LzLT_ULQI/AAAAAAAAARM/AFHNIu--JQcqvpif3VxYT06VkLktQSD6ACPcB/s1600/segment.png]

Once segment tree is built we can answer any range query in (O (Log N)) time as we will only have to traverse as much as height of the tree.

Example Query: Find minimum of range [2-5]

Firstly go to root see query interval is fully inside the segment [0-7] so go down now at node index 2 of tree represents segment[0-3] here it covers [2-3] part of the query interval so go down,now see tree node at index=4 represents segment [0-1] that is completely out of the query segment so return(inf-a big integer) from there as it can't be the answer. Tree index=5 holds the answer for segment [2-3] that is fully inside the query so return node value(=4). In this way in the right subtree of root node at index=6 will return 8 and node at index=7 will return inf. So minimum of these two will be the answer.

Thus minimum element in range [2-5] = 4 (min(4,8)).

You will understand the above explanation better after seeing the recursive query function in my code.

C++ CODE

```
#include<bits/stdc++.h>

using namespace std;


#define N 8
#define MAX 16
#define inf 999999999

int arr[N];      //Unordered list of elements
int tree[MAX];   //Array to store segment tree

/**
 * Building the segment tree
 */
void build_tree(int index, int ss, int se) {
	if (ss > se){
```

```
        return;
      }
    if(ss == se) { // Leaf node
        tree[index] = arr[ss]; // Initialize value
      return;
    }
    int mid=(se+ss)/2;
    build_tree(index*2 , ss, mid); // Initialize left child
    build_tree(index*2+1, mid+1, se); //Initialize right child

    tree[index] = min(tree[index*2], tree[index*2+1]); // Initializing root value
}

/**
 * update element at index ii of the array,add diff into it
 */
void update_tree(int index, int ss, int se, int ii,int diff) {

    if(ss > se || ss > ii || se < ii) // Segment doesn't contain index ii
      return;

     if((ss == se)and(ss == ii)) {// Update the node
        tree[index] += diff;
        return;
    }
      int mid=(se+ss)/2;
    update_tree(index*2, ss, mid, ii, diff); // Updating left child
    update_tree(index*2+1, mid+1, se, ii, diff); // Updating right child

    tree[index] = min(tree[index*2], tree[index*2+1]); // Updating root with min

}

/**
 * Query to find minimum of given range [i, j]
 */
int query_tree(int index, int ss, int se, int i, int j) {
```

```
    if(ss > se || ss > j || se < i){
            return inf;
        } // Segment Out of range

    if(ss >= i and se <= j) {// Segment within Range [i, j]
        return tree[index];
        }
        int mid=(ss+se)/2;
    int p = query_tree(index*2, ss,mid, i, j); // Query left child
    int q = query_tree(1+index*2, 1+mid, se, i, j); // Query right child

    int answer = min(p, q); // Return result

    return answer;
}

int main() {
    arr[0]=5;
        arr[1]=2;
        arr[2]=4;
        arr[3]=6;
        arr[4]=11;
        arr[5]=8;
        arr[6]=3;
        arr[7]=2;
        //Building Segment Tree
    build_tree(1, 0, N-1);
        // Updating the tree
    update_tree(1, 0, N-1, 6, 5); // Increment element at index [6] by 5
    update_tree(1, 0, N-1, 7, 12); // Increment element at index [7] by 12
    update_tree(1, 0, N-1, 1, 100); // Increment element at index [1] by 100
        //Query to find minimum element of a Range
    cout << query_tree(1, 0, N-1, 0, N-1) << endl; // Get min of range [0, N-1]

        return 0;

}//End
```

If we try to update a Range using above code,it will take O(N) time because we will have to update all the segments containing the Range.So now segment tree will act like simple array (O(N) time to update). To solve this problem there is a trick but it takes O(N) more space. It is called lazy propagation.

In lazy propagation we only update the interval when that interval is needed to perform a query and to do so we store update values in a different array and use that array to update interval at the time of query.So updating the range operation can be done in O(LogN) time itself,but keep in mind we don't really update the range in segment tree we just store it in lazy array and give the updated result at the time of query.To understand it more see my code it uses a lazy array to store some values temporarily.

C++ CODE (With Lazy Propagation)

Let we have an array of 8 integers initially containing 50 at each index.
arr[]={50,50,50,50,50,50,50,50}
We will do several updates on array and then query to print minimum of a range.

```
#include<bits/stdc++.h>

using namespace std;


#define N 8
#define MAX 16
#define inf 999999999

int arr[N];      //Unordered list of elements
int tree[MAX];   //Array to store segment tree
int lazy[MAX]={0};

/**
 * Building the segment tree
 */
void build_tree(int index, int ss, int se) {
      if (ss > se){
          return;
```

```
        }
      if(ss == se) { // Leaf node
            tree[index] = arr[ss]; // Initialize value
        return;
    }
    int mid=(se+ss)/2;
    build_tree(index*2 , ss, mid); // Initialize left child
    build_tree(index*2+1, mid+1, se); //Initialize right child

    tree[index] = min(tree[index*2], tree[index*2+1]); // Initiaizing root value
}

/**
 * update elements in range [ii,jj] of the array,add diff
 */
void update_tree(int index, int ss, int se, int ii,int jj,int diff) {


       if(lazy[index] != 0) { // This node needs to be updated
           tree[index] += lazy[index]; // Update it

        if(ss != se) {
            lazy[index*2] += lazy[index]; // Mark child as lazy
                lazy[index*2+1] += lazy[index]; // Mark child as lazy
        }

           lazy[index] = 0; // Reset
          }
         if(ss > se || ss > jj || se < ii) // Segment doesn't contain Range [ii,jj]
        return;

      if((ss >= ii)and(se <= jj)) {// Segment fully in range
                tree[index]+=diff;
                if (ss != se){   //if not a leaf node
                     lazy[index*2] += diff;
                     lazy[index*2+1] += diff;
                }
            return;
    }
```

```
        int mid=(se+ss)/2;
    update_tree(index*2, ss, mid, ii, jj, diff); // Updating left child
    update_tree(index*2+1, mid+1, se, ii, jj, diff); // Updating right child

    tree[index] = min(tree[index*2], tree[index*2+1]); // Updating root with min
}

/**
 * Query to find minimum of given range [i, j]
 */
int query_tree(int index, int ss, int se, int i, int j) {



    if(ss > se || ss > j || se < i){
            return inf;
         } // Segment Out of range

         if(lazy[index] != 0) { // This node needs to be updated
        tree[index] += lazy[index]; // Update it

        if(ss != se) {
            lazy[index*2] += lazy[index]; // Mark child as lazy
            lazy[index*2+1] += lazy[index]; // Mark child as lazy
        }

        lazy[index] = 0; // Reset
    }

    if(ss >= i and se <= j) {// Segment within Range [i, j]
        return tree[index];
        }
        int mid=(ss+se)/2;
    int p = query_tree(index*2, ss,mid, i, j); // Query left child
    int q = query_tree(1+index*2, 1+mid, se, i, j); // Query right child

    int answer = min(p, q); // Return result

    return answer;
}
```

```
int main() {
	arr[0]=50;
	arr[1]=50;
	arr[2]=50;
	arr[3]=50;
	arr[4]=50;
	arr[5]=50;
	arr[6]=50;
	arr[7]=50;
	//Building Segment Tree
    build_tree(1, 0, N-1);
	// Updating the tree
    update_tree(1, 0, N-1, 6, 7, -10); // Decrease elements in range [6,7] by 10
    update_tree(1, 0, N-1, 2, 7, -1); // Decrease elements in range [2,7] by 1
    update_tree(1, 0, N-1, 1, 5, -20); // Decrease elements in range [1,5] by 20
	//Query to find minimum element of a Range
    cout << query_tree(1, 0, N-1, 0, N-1) << endl; // Get min in range [0, N-1]

	return 0;
}
```

Problems to try:

1. http://www.spoj.com/problems/KQUERY/
2. http://www.spoj.com/problems/SEGSQRSS/

Guys if you have any query or find any error in the above explanation please comment. #HappyCoding

Posted 7th September 2016 by Unknown

7 View comments

ranbir kapoor September 10, 2016 at 11:47 AM

Nice explanation dude but plz provide the code

Reply

Replies

Kartik Chaudhary September 10, 2016 at 2:55 PM

Code is there now.Thank YOu :)

Reply

Turxan Badalov September 13, 2016 at 12:33 PM

First of all thank you for this good material! I can't understand only one condition above in update_tree function:
if((ss <= ii)and(se >= jj)) {// Segment fully in range
tree[index]+=diff;
Is it correct? Maybe you wanted to write == instead of <= and >= ?
Because it will change all my vertices in [0; N-1] range, that is not actually what I want. Or I am missing something?

Reply

Replies

Kartik Chaudhary September 14, 2016 at 4:37 AM

Thank You for enlightening the mistake.I have updated it now.

Reply

Turxan Badalov September 13, 2016 at 4:55 PM

I have tested this and think it should be:
(ii <= ss)and(se <= jj)
Correct me if I am wrong.

Reply

Replies

Kartik Chaudhary September 14, 2016 at 4:38 AM

Yes you are absolutely correct.Updated now.

Reply

Anonymous November 18, 2016 at 3:09 AM

Thankyou for this simple explaination. Plz update fenwick tree also.

Reply

Enter your comment...

Comment as: frankie.y.liu@gr ▾ Sign out

Publish Preview Notify me