Let us code

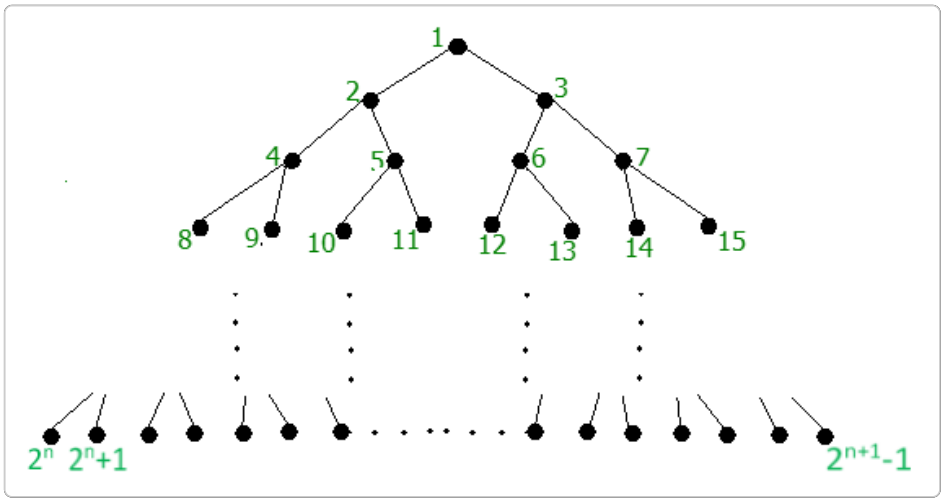
Segment Trees

Motivational Problems:
http://www.spoj.pl/problems/BRCKTS/
http://www.spoj.com/problems/GSS3/
http://www.spoj.com/problems/HORRIBLE
http://www.spoj.pl/problems/IOPC1207/
https://www.spoj.com/problems/GSS2/
http://www.spoj.com/problems/SEGSQRSS/
http://www.spoj.com/problems/ORDERSET/
http://www.spoj.com/problems/HELPR2D2/
http://www.spoj.com/problems/TEMPLEQ

Segment trees (shortened as segtrees), as some of you might have heard of, is a cool data structure, primarily used for range queries. It is a height balanced binary tree with a static structure. The nodes of segment tree correspond to various intervals, and can be augmented with appropriate information pertaining to those intervals. It is somewhat less powerful than balanced binary trees because of its static structure, but due to the recursive nature of operations on the segtree, it is incredibly easy to think about and code.

Structure

In a segtree, the basic structure (an array in almost all cases) on which you want to answer queries are stored at leaves of the tree, in the sequential order. All internal nodes is formed by "merging" its left and right children. The overall tree is a complete binary tree of height n.



A segtree on 2^n elements. The children of node labelled i are (2*i) and (2*i+1).

The diagram shows how the binary tree is built up and how the nodes are indexed. Given a node with label *i*, its left and right children are *2i* and *2i+1* respectively, and nodes $i*2^k$ to $(i+1)*2^k$ -1 are its kth level descendants. However, segtrees are most effective when you think of them as just a recursive structure.

As an example, consider a segment tree with n=3. As in the above figure, this would mean that the segment tree has 15 nodes. Such a segment tree could be used to store ranges corresponding to an array of size 8 (indices 0 through 7). The leaf nodes (8 to 15) all correspond to intervals containing one element only : That is, node 8 would correspond to the interval [0,0] in the array and node 9 would be [1,1] and so on. As we go up the segtree, the interval corresponding to each node in the segtree is found by merging the intervals of its two children. That way, node 4 will correspond to interval [0,1] and node 3 to interval [4,7] and so on.

Now assume that you need to make a query on some arbitrary interval in the array. The most straightforward way would be to look at the lowermost level in the segment tree. But that would require as many operations as there are elements in the interval, and is hence costly. For example, If one desires to query the interval [2,7], this would mean having to look at the nodes 10, 11, 12, 13, 14 and 15 in the segtree. But we can be smart and choose only nodes 3 and 5 : the former takes care of the interval [4,7] while the latter takes care of [2,3]. When we have longer intervals and thus deeper segtrees, the advantage gained by choosing conjoined intervals is huge. The basic idea behind segment trees is to recursively choose the best intervals for updation and querying such that the total number of operations needed is reduced.

The exact implementation is detailed in the rest of the article. In this blog, I go on to show that most of the times, a segtree can be described by these fundamental operations: **merge**, **split** and **update_single_subtree**. Add to it **binary_search**, and almost every variant of segtree that I have come across can be broken down in terms of these operations. At the end of this blog, I will give a **stl-like packaged version of segtree**.

Basic Idea:

A segtree can be seen of as doing two things at the same time: a) The internal nodes **summarize the information stored at descendant leaves**, so that the information about all the leaves can be extracted quickly. The **merge** operation does this task of summarizing the information stored at two nodes into a single node. b) The internal nodes **store the operations that have to be applied to its descendants**. This gets propagated down in a lazy manner. The **update_single_subtree** puts information about how to update the leaves into a single node, and **split** operation propagates this down the tree in a lazy manner. For example, if you have to add a constant C to all descendant leaves, the Update_single_subtree operation will store the value of C at the node, and split operation will pass down the value of C to both its children.

Merge Operation:

Example 1

Lets try to solve the following problem using segtrees:
Given an array A[1 ... m], you want to perform the following operations:
a) report minimum element in a range A[i ... j]
b) set A[i] = x.

Choose a value of n = depth of your tree = ceil (log m), so that all the array elements can fit at the lowest level of your tree. The merge function for this problem does exactly what you think ! Yes, it will just take the two values and return the minimum. So the first operation that is required will be taken care of by merge itself. The second operation requires us to modify one of the leaves of our tree. There are two possible approaches: a) Update the leaf, and notify all of its parents in the tree, to update the information they store.

```
struct node{
        int val;
        void split(node& l, node& r){}
        void merge(node& a, node& b)
        {
                val = min( a.val, b.val );
        }
}tree[1<<(n+1)];
node range_query(int root, int left_most_leaf, int right_most_leaf, int u, int v)
{
        //query the interval [u,v), ie, {x:u<=x<v}
        //the interval [left_most_leaf,right_most_leaf) is
        //the set of all leaves descending from "root"
        if(u<=left_most_leaf && right_most_leaf<=v)
                return tree[root];
        int mid = (left_most_leaf+right_most_leaf)/2,
                left_child = root*2,
                right_child = left_child+1;
        tree[root].split(tree[left_child], tree[right_child]);
        node l=identity, r=identity;
        //identity is an element such that merge(x,identity) = merge(identity,x) = x for all x
        if(u < mid) l = range_query(left_child, left_most_leaf, mid, u, v);
        if(v > mid) r = range_query(right_child, mid, right_most_leaf, u, v);
        tree[root].merge(tree[left_child],tree[right_child]);
        node n;
        n.merge(l,r);
        return n;
}
void mergeup(int postn)
{
        postn >>=1;
        while(postn>0)
        {
                tree[postn].merge(tree[postn*2],tree[postn*2+1]);
                postn >>=1;
        }
}
void update(int pos, node new_val)
{
        pos+=(1<<n);
        tree[pos]=new_val;
        mergeup(pos);
```

```
43 }
```

min_segtree_query.cpp hosted with ❤ by GitHub view raw

b) Implement the update_single_subtree function which will update the value of node. You will not not need to implement a split function because information is not propagated downwards.

```
1  void update_single_node(node& n, int new_val){
2          n.val = new_val;
3  }
4  void range_update(int root, int left_most_leaf, int right_most_leaf, int u, int v, int new_val)
5  {
6          if(u<=left_most_leaf && right_most_leaf<=v)
7                  return update_single_node(tree[root], new_val);
8          int mid = (left_most_leaf+right_most_leaf)/2,
9                  left_child = root*2,
10                 right_child = left_child+1;
11         tree[root].split(tree[left_child], tree[right_child]);
12         if(u < mid) range_update(left_child, left_most_leaf, mid, u, v, new_val);
13         if(v > mid) range_update(right_child, mid, right_most_leaf, u, v, new_val);
14         tree[root].merge(tree[left_child], tree[right_child]);
15 }
16 void update(int pos, int new_val){
17         return range_update(1,1<<n,1<<(n+1),pos+(1<<n),pos+1+(1<<n),new_val)
18 }
```

range_min_segtree.cpp hosted with ❤ by GitHub view raw

Example 2

problem: http://www.spoj.com/problems/GSS3/

The data stored at node and merge function are given below. Rest of the code remains same as Example 1 above. Just to make things clearer, I have also given how to create a single leaf from a value.

```
1  struct node
2  {
3          int segmentSum,bestPrefix,bestSuffix,bestSum;
4          node split(node& l, node& r){}
5          node merge(node& l, node& r)
6          {
7                  segmentSum = l.segmentSum + r.segmentSum;
8                  bestPrefix = max( l.segmentSum + r.bestPrefix , l.bestPrefix );
9                  bestSuffix = max( r.segmentSum + l.bestSuffix , r.bestSuffix );
10                 bestSum    = max( max( l.bestSum , r.bestSum) , l.bestSuffix + r.bestPrefix );
11         }
12 };
13 node createLeaf(int val)//the leaf nodes, which represent a single element of the array, will be created in this mar
14 {
15         node n;
16         n.segmentSum = n.bestPrefix = n.bestSuffix = n.bestSum = val;
17         return n;
18 }
19 //range_query and update function remain same as that for last problem
```

gss3.cpp hosted with ❤ by GitHub view raw

Exercise:

Using only the above concept of merge function, try solving the following problem:
http://www.spoj.pl/problems/BRCKTS/

Split Operation:

Example 3:

Let us add another operation to example 1:
c) add a constant C to all elements in the range A[i..j]

Now you will have to add another parameter to your node structure, which is the value you have to add to all the descendants of that node.
I show the modified merge and the split functions. Also, the update_single_node functions for operation c) is shown.

```
struct node
{
        int min, add;
        split(const node& a, const node& b)
        {
                a.add+=add,  a.min+=add,
                b.add+=add,  b.min+=add;
                add=0;
        }
        void merge(node a, node b)
        {
                min = min( a.min, b.min );
                add = 0;
        }
};
void update_single_node(node& n, int add)
{
  n.add+=add,
  n.min+=add;
}
//range_query, update and range_update remain same.
```

Example 3.cpp hosted with ❤ by **GitHub** **view raw**

An observant coder might notice that the update function for operation b) will need to be changed slightly, as shown below.

```
void splitdown(int postn)
{
        if(postn>1) splitdown(postn>>1);
        tree[postn].split(tree[2*postn],tree[2*postn+1]);
}
void update(int postn, node nd)
{
        postn += 1<<n;
        splitdown(postn>>1);
        tree[postn] = nd;
        mergeup(postn);
}
```

general_merge_up.cpp hosted with ❤ by **GitHub** **view raw**

The incredible thing that I feel about segtrees is that it is completely defined by the structure of the node, and the merge and split operations. Therefore, all the remaining code does not change.

Example 4:
problem http://www.spoj.com/problems/HORRIBLE/
The relevant functions are given below:

```
struct node
{
        int numleaves, add, sum;
        void split(node& l, node& r)
        {
                l.add += add;
                l.sum += add * l.numleaves;
                r.add += add;
                r.sum += add * r.numleaves;
                add=0;
        }
        void merge(node& l, node& r)
        {
                numleaves = l.numleaves + r.numleaves;
                add = 0;
                sum = l.sum + r.sum;
```

```
        }
};
void update_single_subtree(node& n, int inc){
        n.add += inc;
        n.sum += inc * n.numleaves;
}
//range_query and range_update remain same as that for previous problems
```

horrible.cpp hosted with ♥ by GitHub view raw

Exercise:
Try solving
http://www.spoj.pl/problems/IOPC1207/
(Hint: you will need to maintain 3 separate trees, one each for X, Y and Z axis).
http://www.spoj.com/problems/SEGSQRSS/
http://www.spoj.com/problems/DQUERY/
(Hint: you will need to store all the queries and process them offline)
https://www.spoj.com/problems/GSS2/
(Hint: you will need to store all the queries and process them offline)

Binary Search:

Often, you are required to locate the first leaf which satisfies some property. In order to do this you have to search in the tree, where you traverse down a path in the tree by deciding at each point whether to go to the left left subtree, or to the right subtree. For example, consider the following problem: http://www.spoj.com/problems/ORDERSET/

Here, you have to first apply co-ordinate compression to map the numbers from range [1 .. 10^9] to [1 .. Q]. You can use stl maps for doing that, so assuming that that is done, you would again need a simple segtree as shown. Note that INSERT and DELETE operation can be defined by just implementing the update_single_subtree function separately for each case. Also, COUNT(i) is nothing but query(0,i). However, to implement k-TH, you will need to do a binary search as shown below:

```
struct node{
        int num_active_leaves;
        void split(node& l, node& r){}
        void merge(node& l, node& r){
                num_active_leaves = l.num_active_leaves + r.num_active_leaves;
        }
        bool operator<(const node& n)const{
                return num_active_leaves < n.num_active_leaves;
        }
};
int binary_search(node k)
//search the last place i, such that merge( everyting to the left of i(including i) ) compares less than k
{
        int root = 1;
        node n=identity; //identity satisfies merge(identity,y) = merge(y,identity) = y for all y.
        assert(!(k<identity));
        while(!isleaf(root)){
                int left_child = root<<1, right_child = left_child|1;
                tree[root].split(tree[left_child],tree[right_child]);
                node m;
                m.merge(n,tree[left_child]);
                if(m<k){//go to right side
                        n=m;
                        root=right_child;
                }else root=left_child;
        }
        node m;
        m.merge(n,tree[root]);
        mergeup(root);
        if(m<k)return root-leftmost_leaf;
        else return root-1-leftmost_leaf;
}
```

orderset.cpp hosted with ♥ by GitHub view raw

Exercise:

solve http://www.spoj.com/problems/HELPR2D2/ by implementing merge, binary search and update_single_subtree.
solve http://www.spoj.com/problems/TEMPLEQ

Finally to conclude this blog, here is link to **stl-type packaged version of segment trees** as promised earlier. Example of using the segtree class by solving spoj problems:
http://www.spoj.pl/problems/BRCKTS/, sample code here
http://www.spoj.com/problems/HORRIBLE, sample code here
http://www.spoj.com/problems/ORDERSET/, sample code here

Acknowledgments:
Raziman TV for teaching me segment trees.
Meeting Mata Amritanandamayi at Amritapuri inspired me to do something to improve level of programming in India.

Posted by utkarsh lath at Tuesday, January 01, 2013

Labels: data structure, segment tree

78 comments:

Ankesh Kumar Singh January 1, 2013 at 8:33 PM

Awesome work!

Reply

bloops January 1, 2013 at 8:55 PM

Great article! Looking forward even more articles from you. :)

Reply

sai16vicky January 1, 2013 at 9:30 PM

Very good work !!!

Reply

Risky January 1, 2013 at 11:36 PM

Super work. Looks like this place is going to be the go-to place for a guy aspiring to become a serious contender in programming contests.
Thanks a lot!

Reply

vikas pandey January 2, 2013 at 1:10 PM

nice one....Thanks for this...It will be good if we have a tutorial like this for binary indexed trees also....Thanks..:)

Reply

Replies

Crazy pk@mit April 8, 2013 at 8:18 AM

Absolutely true @! :)

Reply

shubham agarwal January 25, 2013 at 12:02 PM

grt work..
can anyone help me to solve this problem http://www.spoj.com/problems/CTRICK/ using segment tree.
I have an solution in O(n^2) which gives TLE.

Reply

Adrian Carballo January 30, 2013 at 9:03 AM

The function mergeup in example 1 will run forever because the value postn should be updated as postn >>= 1 in the while block, but it's not.

Reply

Replies

utkarsh lath January 30, 2013 at 9:02 PM

Thanks for pointing. Corrected.

Reply

Viki Sharma February 4, 2013 at 6:40 PM

Can anyone help me out with http://www.spoj.pl/problems/IOPC1207/
How exactly will 3 segment trees help? An update on one of the coordinates should also trigger an update on the other 2. How to manage all this? Will appreciate some help....

Reply

Replies

utkarsh lath February 4, 2013 at 7:59 PM

The three ranges(x,y,z) can be handled independently.

Viki Sharma February 5, 2013 at 12:19 AM

Thanks for the reply... In one of the booklets, I had seen the following formula: r1r2r3 + r1g2g3 + g1r2g3 + g1g2r3. What is its derivation?

Reply

vikash kumar March 20, 2013 at 1:43 AM

can u please explain in solution GSS3 that
what represent the bestprefix and bestsuffix ???

Reply

Replies

utkarsh lath March 20, 2013 at 1:52 AM

As you already understand, a node of the segtree corresponds to a subarray of the original array. Now, bestSuffix of a node corresponds to suffix of that interval whose sum is largest. Therefore, bestSuffix stores the sum of the 'best' suffix. Similarly, bestPrefix of a node corresponds to prefix of that interval whose sum is largest.

Reply

Naman Kalkhuria May 19, 2013 at 6:44 PM

i am struggling with your brckts spoj code,,,where i dont understand what min is representing in each node.

Reply

Replies

Sai Teja Pratap August 25, 2013 at 11:09 AM

Min represents the minimum prefix sum of the corresponding node
In other words if node n represents the range [x,y] min is minimum of a_x+ a_(x+1)...a_(j) such that j<=y

Reply

Sai Teja Pratap August 22, 2013 at 2:06 AM

Hi,
I did not understand the purpose of line 24 in 1st snippet.
tree[root].merge(tree[left_child],tree[right_child]);

Why is this merge required?

Reply

Replies

utkarsh lath August 22, 2013 at 10:30 PM

Hi,
In the first snippet, doing merge is not necessary, because your split operation does not do anything. However, in general, your split operation might rely on an impending merge when tracing path back up the tree.

Reply

Mario M August 23, 2013 at 4:15 AM

1.What do the arguments "left_most_leaf" ,"right_most_leaf",u,v mean in range_query and range_update functions mean?
2.Why is merge not necessary in Line 24 of Snippet 1? If its not done , then how is value at tree[root] updated?

Reply

Replies

utkarsh lath August 23, 2013 at 7:41 AM

1. As expected, left_most_leaf is the array index of leaf which is reached when you keep going going towards left child starting from root. Conceptually, every node of segment tree represents a segment of the array. This segment is [left_most_leaf, right_most_leaf). That means, the segment starts at left_most_leaf'th element(inclusive) of segtree and ends at right_most_leaf'th element(exclusive). Note that first leaf of the segtree has index 2^n.
2. Merge function is necessary in update function which takes care of updating value of all relevant nodes. It may or may not be necessary for query function depending on whether the child could possibly get updated when query is made.

Mario M August 23, 2013 at 11:56 AM

What is the time complexity of range_update function? Can you please justify it ?

utkarsh lath August 25, 2013 at 8:54 PM

Time complexity at of course O(log n). Consider an update call for an interval [u, v). A node of the segtree representing the subinterval [l, r) is visited iff
1. [l, r) contains u or v-1.
2. [l, r) is a subset of [u, v) and parent of [l, r) is not.

Justification 1(short version).
Consider the leaves representing left and right end points of your interval. You could eventually visit these two nodes. Therefore, you can visit any ancestor of these two nodes as well, because only these nodes can potentially satisfy property 1. Also, for any other node that satisfies property 2, its sibling must satisfy property 1 (if its sibling also covers the interval [u,v), then so will its parent). Therefore (4 log n) nodes will be visited in worst case.

Justification 2(longer version of above).
Nodes at level(height) i represent intervals [0, 2^i), [2^i, 2*2^i), [2*2^i, 3*2^i), ...
There will be at most two nodes at any level that 'partially' cover the interval [u,v). These nodes will be those whose intervals that contain u or v-1. Rest of the nodes will either cover the interval [u, v) completely, or not cover at all. Let the xth node at level i

contain u in its interval and yth node at level i contain v in its interval. The it is easy to see that at most 4 nodes at level i will be visited by our algorithm. These four nodes are node #x, node #x+1(if it is right sibling of node#x), node #y-1(if it is left sibling of node#y), node #y. For any other node between x and y, the property 2 mentioned on top will not be satisfied.

Reply

HARSHHH August 28, 2013 at 7:55 AM

it is awesome !!..
can u tell me why are we using split function in example 1 although we haven't written any code inside it ??

Reply

Replies

utkarsh lath August 30, 2013 at 6:14 AM

Split is not necessary for example 1, however, I have included because
a) It is needed in general.
b) I have not yet introduced split function other than saying that it should be there. But, I want the reader to see how it naturally fits into the scheme of things. Everytime you want to work with children of a node, you split it first, and when work is over, you merge the two children.

HARSHHH October 14, 2013 at 8:26 AM

I am trying to implement the framework given by you for example 1 but am unable to do it successfully..
Can u tell me were am going wrong .
here is the link..http://ideone.com/9WPfoF
thnx..:)

Reply

Lee Wei August 30, 2013 at 5:57 AM

Hi,

I refer to your STL segtree class "segment_tree.cpp" @ https://gist.github.com/utkarshl/4417204

For this line in the code:
static bool const has_split = (sizeof(test(0))==sizeof(char));

Could you please explain what this line of code is trying to accomplish, especially this expression `sizeof(test(0)`? I can't seem to grasp the meaning of this. Intuitively I was thinking that this should be related to the 'lazy propagation' feature of your segtree implementation, where you only pushdown/split whenever the need arises. What has this got to do with sizeof(), esp. sizeof char?, and how does test(0) execute, as in what other functions/methods get invoked once this is called?

Great tutorial btw. Thanks!

Reply

Replies

Lee Wei August 30, 2013 at 6:14 AM

Is it merely there as a placeholder for `false`?

utkarsh lath August 30, 2013 at 6:17 AM

I dont want to force the user to write a split function, however, if split function is there, it should be called. And all this should be decided at compile time. After lot of googling and trial and error, I found that what I did does the trick. "has_split" is true iff the split function is implemented..

Lee Wei August 30, 2013 at 7:06 AM

Oh I see, could you please point me to a link/tutorial that explains in further detail how this line of code achieves that please? I'm still rather confused how this happens, I'm afraid.

Lee Wei August 30, 2013 at 7:58 AM

I sort of understand how your code block works now. Your STL class got me AC on this problem on my first try @ http://codeforces.com/contest/339/problem/D. Thanks again!

Lee Wei August 30, 2013 at 9:30 AM

Btw, have you ever considered extending this tutorial with the same depth of explanation and details to illustrate the implementation of 2D Segment Trees? This data structure would be useful for computational geometry/interval/points-in-space kind of competitive programming problems.

utkarsh lath August 30, 2013 at 10:13 AM

Hi,
I have not seen many applications of 2-D segment trees in programming contests, thats why I never thought of putting it up. If you would be kind enough to point to some problems for which 2-D segtrees are the best way to go, it would be motivating enough for a sequal :)

Lee Wei August 30, 2013 at 10:22 AM

Alright, I'll source for the exact problems as examples. Another possible extension to look into is also "Persistent Segment Trees" (meaning you keep multiple versions of a static variant of a segtree) eg. @ http://discuss.codechef.com/questions/6548/query-editorial

utkarsh lath August 30, 2013 at 10:32 AM

I am clear about how to go about persistence when information is propagated upwards only(no split operation). However, when split operation comes into picture, things become messier, and I currently dont know how to make persistent segtree for such a case :(

Lee Wei August 31, 2013 at 8:00 AM

Alright, I see. I don't have any idea either, of how to fit the persistent segtree into your nice framework right now.

On another note, I still don't quite see how lazy propagation is included in your segtree framework? Has it got to do with defining some extra functions and node attributes? Maybe you can illustrate quickly using this problem here @ http://www.codechef.com/problems/FLIPCOIN, which supposedly will TLE if you don't use lazy propagation (refer to discussion thread @ http://stackoverflow.com/questions/10715450/lazy-propagation-in-segment-tree)

Lee Wei August 31, 2013 at 8:23 AM

Also, could you please clarify why the methods split_down(int leaf_idx), merge_up(int leaf_idx) and is_leaf(int tree_idx) are declared public in your segtree class? and demonstrate when+how to use them please?

It only seems clear to me to advertise point/range query/update, and binary_search (and init of course).

utkarsh lath August 31, 2013 at 8:46 AM

solution of flipcoin.
http://www.codechef.com/viewsolution/2592784

Split operation does the lazy propagation :)

These functions(along with all the class fields) are there mainly so that in case you decide to do some stuff on your own, you can use these as helper functions.

Lee Wei August 31, 2013 at 9:38 AM

Oh alright, thanks for the quick response!

I'm trying to understand the solution to FLIPCOIN using your framework @ http://www.codechef.com/viewsolution/2592784, but I'm confused on this one line in method flip_me() within struct node's definition:

flip ^= 1;

I understood flip to represent whether the interval for which the current node is responsible for needs to be updated. For example, for the line of code `s.update<&flip_node>(a,b);`, I look for the node X that summarizes the information for the interval [a,b], and then make its flip flag true by setting `X.flip = true;` to say that I want to flip all the array elements, where the flipping operation is taken to mean as the question specifies. This would be my reasoning for writing `flip=true` within flip_me() instead of `flip^=1` as in the AC solution (btw flip^=1 is equivalent to saying 'if flip is true then make it false, otherwise if flip is false then make it true', ie. toggle the boolean value of flip, right?)

Maybe it might help to teach me how this is supposed to work `segtree.update(a,b)`, so that I know what I should be coding in each of the following pluggable bits:
- FUNCTION: this is called on the minimal set of nodes responsible for the interval [a,b]. Is this understanding correct?
- I understand how merge() works, but not quite sure how split() works, meaning: is it called after FUNCTION? how is lazy propagation ensured - in this case, I think it means this line of code `if(not flip)return;` within split(), because without it, it will just propagate down both the children of whichever current node it is examining
- In short, I'd like to know exactly what should be my objective/thoughts when trying to write the split() function, as well as FUNCTION?

Thanks for your help in advance. Much appreciated

utkarsh lath August 31, 2013 at 10:07 AM

Okay, firstly, you need to understand why toggle flip instead of setting it to true. Suppose you perform two update operations, one on interval (1, 5) and other on interval (3, 9). Then the interval (3,5) has been flipped twice, which is equivalent to not flipping it all. That is the logic behind making toggling the value of flip. If already flipped, 'unflip' it, else flip it.

The split/merge operations work based on the fact that any single interval can be represented as a union of O(log n) nodes of our segtree.

When you decide to use lazy propagation, you must decide what information needs to be passed onto children of a node, and how would you store it at a single node to 'lazily' propagate it down the tree at some point of time in future. In this problem, I store a flag 'flip' to indicate whether all children of the current node should be flipped. This flipping process of course takes place lazily. That means for now, I just store this flag, and update the current node so that all parents of the current node no longer need to bother about it. And if at some point of time in future, a query/update is made on an interval which intersects the interval of current node, then **at that point of time** I will update my children(split function does exactly this). In this way, whenever you visit a node of a segtree, it is guarenteed to be fully updated, inspite of the fact that all nodes are not updated as soon as an update operation is requested.

Lee Wei August 31, 2013 at 10:24 AM

Ok, that is a good explanation, and again, a quick response.

1. I really needed the part "...flipped twice, which is equivalent to not flipping it all..." that now clears up why flip is being toggled

2. so split/merge will only be invoked onto the minimal set of nodes necessary to cover the interval that is being queried, is this understanding correct? For example, as in your tutorial diagram, for the nodes 10-15, split/merge will only be called up to the point of nodes 3 and 5, since 3 covers [12,15] and 5 covers [10,11], yes?

3. if understanding in #2 is right, then your 3rd paragraph makes sense. So if I were to rephrase what you've mentioned, when you first encounter an update statement, all you do is set the flags as necessary, but don't actually propagate the changes down the tree. Later on, if a new query/update over an interval that overlaps but does not completely cover this interval, for example as per your tutorial diagram, say the interval [12,14] is requested, then the split/merge operation would return the set of nodes [6 and 7]. However, these being children of node #3, now contain 'stale'/'outdated' information, and so they need to be updated. This updating will be done by propagating the latest information from your previously 'marked' node #3 downwards (and this is via the split() as you've implemented) to nodes #6 and #7, but no further than that, because the set of nodes [6 and 7] cover the requested interval entirely. Thereafter, to return the results of the query, merge will be called in reverse to return the updated (and thus correct) results.

utkarsh lath August 31, 2013 at 10:29 AM

You got it right :)

Lee Wei August 31, 2013 at 10:30 AM

Sorry, for my point 3, i meant nodes 6 and 7 covering interval of nodes 12 to 15 inclusive, as per your tutorial diagram, not 12 to 14.

Lee Wei August 31, 2013 at 10:31 AM

excellent, thank you very much again!

Lee Wei August 31, 2013 at 11:02 AM

Hi again,

I also need to seek clarification concerning the identity node, and the invariant that merge(identity,y)=merge(y,identity)=y.

How would you construct such an identity node, and just briefly indicate how the invariant holds, for a problem like http://codeforces.com/contest/339/problem/D, where the merge function (whether to employ XOR or OR) depends on the parity of the depth/level of the node?

How does the correctness of the identity node affect the answer? Is it always safe to just initialise all node attributes that are integers to 0 for merge functions like sum,min,max etc.? and 1 for merge functions that involve multiplication (since 1*x=x*1=x) - is that an example of what you mean by identity node?

Thanks again

utkarsh lath August 31, 2013 at 11:28 AM

Hi,

As you can see, the only place where identity can affect the result is in binary search function. If you do not plan to use this function, just pass any valid node as identity and it will work. It is true that identity might not exist for arbitrary operations, however, when binary search makes sense, identity is usually defined. I had two options to implement identity:
a) Take it as a parameter in binary_search function every time it is called. Since identity is not going to change for every call of binary search, it is not a good idea.
b) Take it in the constructor itself. Might cause a bit inconvenience, but if you do not plan to use binary search, value of identity is inconsequential, so pass anything convenient.

Lee Wei September 2, 2013 at 4:58 AM

Yes, I'm aware that it's only used in the binary_search function, as you've said, and it's also clearly evident from the framework code.

Alright, I'll take your word for it, that when it's necessary, it'll be well-defined. Thanks

Reply

Lee Wei September 11, 2013 at 11:20 AM

Hi,

I've recently come across a problem in an ongoing contest which you might be interested to validate your segtree framework with @http://www.codechef.com/SEPT13/problems/MLCHEF

Lee Wei

Reply

Replies

utkarsh lath September 12, 2013 at 7:18 AM

We will discuss about this once the contest is over :)

Lee Wei September 14, 2013 at 1:20 AM

Found this SO post @ http://stackoverflow.com/questions/18687589/after-subtracting-a-number-from-a-sequence-how-many-of-remaining-numbers-are-po/18688191#18688191, but my implementation from it is not yet correct.

Lee Wei September 16, 2013 at 2:57 AM

haha I didn't know/realise you were the editorialist for that question!

Lee Wei September 16, 2013 at 7:47 AM

I didn't manage to use your STL segtree framework template to get AC for that problem MLCHEF. Now that the contest is over, would you please demonstrate how I could do that? I'm unable to find your submission for that problem on CodeChef, using your handle 'utkarsh_lath'. Thanks

utkarsh lath September 17, 2013 at 1:32 AM

http://www.codechef.com/viewsolution/2684822

Solution of the above problem Using my framework :)

Lee Wei September 23, 2013 at 8:59 AM

hi again, thank you very much for illustrating how to use your code to solve MLCHEF. i'm trying to learn from that example, but am currently stuck at understanding your `remove_dead_chef()` function. would you please enlighten me?

i noticed you call `remove_dead_chef()` right after updating the segtree, which i can understand, because it is certainly necessary to "kill" the chefs (ie. leaf nodes of your segtree), by "resetting" the nodes representing them in some way. i would like to firstly ask:
- could you not have reset them to the identity node instead of creating another "zero" node? since it does mention in the problem spec that dead chefs, ie. those with hp<=0, become "immortal", ie. hp=infinity.

my next point of clarification lies with the implementation of this function. i shall reproduce it below for your convenience:

```
void remove_dead_chef()
{
int idx;
while((idx=tree.binary_search(ZERO)+1)<=N) {
tree.update(idx);
}
}
```

- i understand that after update, if at least one chef dies, the segtree will have internal nodes whose min <= 0 (not necessarily min=0), and also at least one leaf node (ie. representing a single chef) whose min must be <=0 (also not necessarily 0), alive will still be 1 because at this point you've never propagated any alive values downwards (ie. split()), and poison = 0. so, in English, what is your `remove_dead_chef()` function trying to achieve at the end of it? it seems to me like you're (binary-)searching through the segtree for a node whose values - min,alive,poison - are all 0, which according to my understanding as described in the previous statement, has a possibility that there is no such node in the segtree - in the case when the chef has been poisoned more than his original health. And even if you found such a node, it's already ZERO - all its fields are 0 - so the `n.init(0)` code in `remove_chef()` is redundant?

i hope my confusion comes across. please explain so that I can understand and learn.

much appreciated.

utkarsh lath September 23, 2013 at 9:40 AM

Hi again,

If you look at the less operator

```
bool operator<(const node& n) const {
return n.min<min;
}
```

It is clear that If node n is not 'dead' (or dead, but handled previously), then n<ZERO.
However, if even a single leaf under it is dead, n<ZERO will return false.

Now, look at the definition of binary search from my code

//search the last place i, such that merge(everyting to the left of i(including i)) compares less than k

Now, when you think about it, if the binary search function does exactly what it states, then it should give you the index of first dead chef.

Indeed, I have reset them to identity. node n; n.init(0); makes n identity. I have not reset it to ZERO, because its health is set to infinity, even though alive count is 0.

Yours doubt about "when does information of dying of a chef get propagated down to the corresponding leaf?" It happens during binary search. Binary search always calls split function when going down a path.

I hope things are clear now :)

Lee Wei September 25, 2013 at 9:51 AM

hmm, i'm having trouble thinking about the high level meaning of binary_search and your ordering.

1. firstly, in your definition of operator<, `n.min== (other value)) according to their min health/hp attribute, right? at this point, shall i think of dead chefs as having negative hp or INF hp, and therefore the nodes they are subsumed under too? ie. conceptually, it is (alive chef with some positive hp)...(ZERO)...(dead chef negative hp), and not (dead chef INF hp)...(alive chef with some positive hp)...(ZERO), right?
therefore, if the dead chef has negative hp, itself, or some parent node it is subsumed under, has min health/hp < 0, and this node would be <ZERO.

...i'm trying to understand your two statements: "It is clear that If node n is not 'dead' (or dead, but handled previously), then n<ZERO. However, if even a single leaf under it is dead, n<ZERO will return false." in light of my reasoning described in the previous paragraph.

next, conceptually, to reason about your post-condition of your binary_search, you're saying that given the 'arrangement' (i know it may not actually be laid out this way in memory, but i'm trying to think about what your function is really achieving at the end of it all, and therefore learn how/when to use it)
(alive chef with some positive hp)...(ZERO)...(dead chef negative hp)
...returns i, such that everything to the left of i (including i) compares < k (and not <=k right?), does this mean it will return the first of the ZERO nodes, whereas what you really want (and also indicate in your previous comment) is the index of the first dead (hp<0) chef, ie. one after the index of the last of the ZEROs...please correct me if any of this is incorrect.

2. "I have not reset it to ZERO, because its health is set to infinity, even though alive count is 0." - yes I agree with you, and understand that ZERO != identity. but you set the health to infinity and not 0 because the question says so, right?

3. ah. I understand your last point. didn't occur to me that binary_search calls split() (and also merge() on the way back up as well, as it usually is the case, right?)

thanks again very much for your patience

Lee Wei September 25, 2013 at 9:53 AM

just a dummy reply here, because i forgot to ask it to notify me.

utkarsh lath September 25, 2013 at 6:55 PM

There are two aspects to a dead chef:
a) when a chef has just died, his health is <= 0.
b) when a dead chef has been 'handled', we reset its health to infinity.

From now on, dead chef by default means 'chef who has just died and has *not* been _handled_'

Conceptually, the ordering is :
Identity <= dead chefs who have been 'handled' < alive chefs < ZERO [ZERO is not actually present in my array(=segtree)] <= dead chef(=> unhandled)

If a parent node in segtree has a dead chef(=unhandled) under it, then its min value is <=0. So, such a node is **not less than ZERO** [refer to the ordering above]
It looks like things have become difficult for you to understand because if there are two nodes n, m then n < m <=> m.min < n.min. i.e. the ordering of nodes is reverse of the ordering by health. The most healthy node comes first, and least healthy comes last.

I hope now you can see what is wrong with
"therefore, if the dead chef has negative hp, itself, or some parent node it is subsumed under, has min health/hp < 0, and this node would be <ZERO
To make things more explicit, when I say '0', I mean the integer '0' and the ordering is that of integers. When I say 'ZERO', I mean a node and the ordering is that on nodes.

I hope the following statement is clear now, "If node n is not 'dead' (or dead, but handled previously), then n<ZERO. However, if even a single leaf under it is dead, n<ZERO will return false."

""...returns i, such that everything to the left of i (including i) compares < k (and not <=k right?), does this mean it will return the first of the ZERO nodes, whereas what you really want (and also indicate in your previous comment) is the index of the first dead (hp<0) chef, ie. one after the index of the last of the ZEROs...please correct me if any of this is incorrect.""
* < k, not <= k
* Why will there be any ZERO nodes ? I have not explicitly put any ZERO nodes in my segtree, therefore, if any node has 'min = 0', then it is because the chef has died just now. ZERO is not present in my array because dead chefs are made identity, and not ZERO.
* "it will return the first of the ZERO nodes .... " is not right, because the array has no ZERO nodes. Moreover, it returns the index of node just before the first node whose health is <= 0. Therefore, if node with index 3 is the first one whose health is <=0, then my binary search returns 2. This explains the '+1' in "idx=tree.binary_search(ZERO)+1". The health of idx-th chef can be 0, but it is not a ZERO node, because his alive count is still 1. ZERO is only a _conceptual_ node, used for comparisons.
* "whereas what you really want (and also indicate in your previous comment) is the index of the first dead (hp<0) chef" You are wrong, If someone's health becomes 0, they die. so i need (hp<= 0).

2. Right.

3. Good. It does call merge as you can see in the code.

Lee Wei September 26, 2013 at 4:58 AM

yes, thanks for the various confirmations of my understanding, and not to mention the many clarifications as well.

just a few more things: one that I can think of right now is about what you said "Moreover, it returns the index of node just before the first node whose health is <= 0" - so your binary_search is either a case of (if exact match found, return index of first match) otherwise (return greatest node, ie. furthest you can go, - it's quite confusing to name the term here because you are also reversing the natural ordering (as i've guessed) - which does not exceed the node being queried)
eg. according to the same ordering you described previously, ...alive chef{hp>0,alive=1,poison?

consider (using NATURAL, not reversed, order), with the following format for a node {hp=X,alive=Y,poison=Z}, with the same definition of the operator< as n.min<min, the following arrangement:
...{-1,0,0}, {0,1,0}(ie. dead unhandled chef), {1,1,0}...
what would binary_search(ZERO) return? is it the index of the node {0,1,0}? ie. the position before which you should be inserting the node in order not to violate any ordering constraints?

thanks

Lee Wei October 1, 2013 at 10:06 AM

just a hello to remind you of my request above.

utkarsh lath October 1, 2013 at 9:06 PM

Hi. I think you are misunderstanding the very simple statement:

"//search the last place i, such that merge(everything to the left of i(including i)) compares less than k.", which is equivalent to

max { i | min(health[0], health[1], ... health[i]) > 0 }

So, intuitively, it gives index of last guy before first dead chef if any, otherwise N-1. I cant imagine whats not clear in this.

for the array:

...{-1,0,0}, {0,1,0}(ie. dead unhandled chef), {1,1,0}...
binary_search(ZERO) returns the index just before {-1, 0, 0} if {-1,0,0,} is the first dead(unhandled) chef.

Reply

abhishek panigrahy September 16, 2013 at 8:23 AM

Hi Utkarsh ,
For the Problem: http://www.codechef.com/SEPT13/problems/MLCHEF
Can you tell me why this solution of mine yielded TLE (using segment tree and lazy propogation) http://www.codechef.com/viewplaintext/2649148.
I had made a comment on codechef reguarding this submission of mine. But still i am unable to find the error.Can you plz give me a test case where this solution fails or why I am getting TLE for this submission of mine

Reply

Replies

utkarsh lath September 17, 2013 at 12:09 AM

Replied on discuss.codechef.com

abhishek panigrahy September 17, 2013 at 1:04 AM

Thank U very much Utkarsh . I could never have known my fault if not for your help .

Reply

Akshay Sharma September 18, 2013 at 8:22 AM

TYPO:
In code for example 1a) lines 22,23 when recursively calling the function, the name should be range_query instead of query.

Reply

Replies

utkarsh lath September 19, 2013 at 6:57 AM

thanks, corrected

Reply

Akshay Sharma September 18, 2013 at 9:01 AM

- 1 b) looks like the recursive version of mergeUp method. can you justify?
- in example 3c) modified update method, rename 'node n' variable to proper name. It creates confusion with 'integer n' , the size of input set in the function body.

Reply

Replies

utkarsh lath September 19, 2013 at 7:08 AM

- Your question is not very clear, but I will try to answer anyways. mergeup is supposed to be called if you have _edited_ a leaf and the ancestors of leaf should update the information stored. range_update is supposed to edit minimal number of internal nodes to reflect update on a range. Therefore, range_update needs to merge nodes as well, because whenever a leaf gets updated, all ancestors of the leaf should be updated accordingly.

- example 3c modified as suggested. Thanks for pointing.

Akshay Sharma September 23, 2013 at 8:41 AM

What I deduce is range_update is useful only if-
1) there is update on range
2) we can club together individual updates on leaves and call range_update(for corresponding range) once

for the given example,I feel it is more or less doing the same job as mergeUp just recursively. we call update on range(pos,pos+1);

utkarsh lath September 23, 2013 at 9:55 AM

"What I deduce is range_update is useful only if-
1) there is update on range
2) we can club together individual updates on leaves and call range_update(for corresponding range) once
"

You are right. If you cannot club them together, you can use lazy propagation to propagate updates on leaf downwards later at a more convenient time.

If range_update is given a non trivial range, it will merge up, but instead of merging up from a single leaf, it will merge up from a small number of internal nodes - those whose union corresponds to the range given. The total number of node from which mergeup happens is guaranteed to be O(log(n)) for any range.

Reply

Dasthan September 21, 2013 at 1:47 AM

Should the destructor for segtree at https://gist.github.com/utkarshl/4417204#file-segment_tree-cpp-L67 be

delete []tree;

?

Reply

Replies

utkarsh lath September 23, 2013 at 9:46 AM

yep, corrected. thanks :)

Reply

Saksham Garg October 13, 2013 at 1:08 PM

This comment has been removed by the author.

Reply

Mgccl October 16, 2013 at 11:46 PM

Is this basically equivalent to finger tree for a monoid in functional languages? http://www.soi.city.ac.uk/~ross/papers/FingerTree.html

It does look very similar(in functionality) to me.

Reply

Replies

utkarsh lath October 17, 2013 at 12:51 AM

No, its unlikely. But you can read all of it and find out for yourself :)

Mgccl October 17, 2013 at 11:59 AM

From what I read, the only difference seems to be it's there is no obvious way to do things like "add a constant C to all elements in the range A[i..j]", because it's a entirely different monoid operation (+) from the summary structure (min). Still, it be interesting to compare them. I will notify you if there is something interesting going on.

Chao Xu October 17, 2013 at 1:16 PM

I did some thinking and realized I once did implement a summary structure on top of the finger tree and thought it is part of the finger tree structure. never mind that.

However a finger tree with a summary structure should be the operationally the same as segment tree.

Chao Xu October 17, 2013 at 1:48 PM

So it looks like this does almost everything.

http://dailyhaskellexercise.tumblr.com/post/57867162779/sum-over-a-consecutive-subsequence

The infixSum returns a monoid operation over A[i..i+m] in O(log m) time.
The split operation in the finger tree (unfortunately have the same name as the split in segment tree) split the tree to represent A[0..i] and A[i+1..n-1], by feeding an monotonic predicate p into it. So it find the first position i, where p(A[i]) is true). This is how I implemented the infixSum too. If the input monoid element is u. I represent it as (1,u). If the operation is *, then I make a new monoid operation x, so (a,u)x(b,v) = (a+b,u*v). This allow me to split by position. So I split to find a tree represent A[i..i+m], and then take the summary of that tree.

I did not implement the split operation in the segment tree, so naively update the leaves A[i..i+m] can takes O(m log n) time, where n is the number of elements. A more optimal solution would take O(m log (n/m)) if desired.

Note this allow binary search(since split itself is binary search), and have a advantage over segment tree: insert and delete.

In theory this is an amazing data structure! All I need to do is define an operation between the monoid, and a natural predicate for binary search, and I have almost optimal data structure for many problems.

In practice it is amazingly slow because it's way too abstract. and the finger tree implementation I use seem to have amortized instead of worst case bounds for many operations. The one with worst case bounds are so complex I don't think anyone has ever implemented them.

Reply

Vivek Sharma October 20, 2013 at 4:13 AM

Thanks a lot Utkarsh for this wonderful article and also for giving related problem.

Reply

Enter your comment...

Comment as:

Publish Preview

Notify me

Home

Subscribe to: Post Comments (Atom)

Total Pageviews

23,181

Contributors

pgm

Raziman T V

utkarsh lath

Blog Archive