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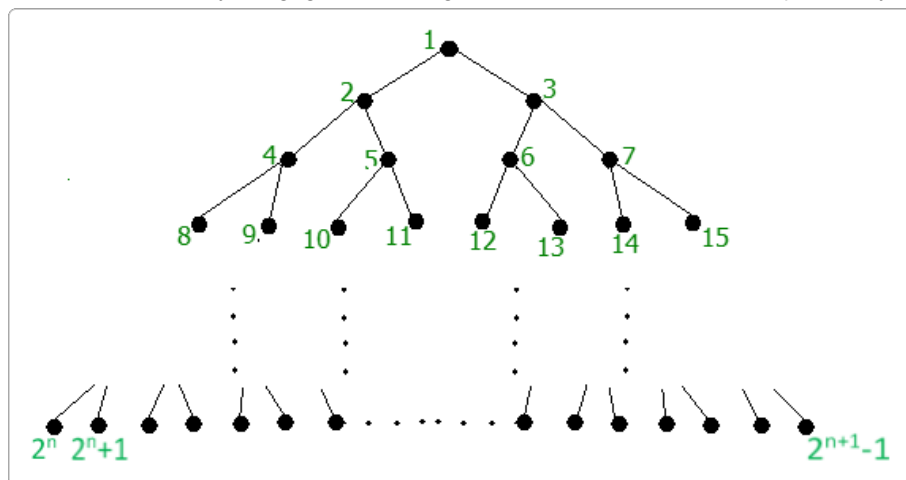
A segtree on 2^n elements. The children of node labelled i are (2*i) and (2*i+1).

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.

```cpp
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);
}
```

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.

```cpp
void update_single_node(node& n, int new_val){
        n.val = new_val;
}
void range_update(int root, int left_most_leaf, int right_most_leaf, int u, int v, int new_val)
{
        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 man
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.

```
 1  struct node
 2  {
 3       int min, add;
 4       split(const node& a, const node& b)
 5       {
 6            a.add+=add,  a.min+=add,
 7            b.add+=add,  b.min+=add;
 8            add=0;
 9       }
10       void merge(node a, node b)
11       {
12            min = min( a.min, b.min );
13            add = 0;
14       }
15  };
16  void update_single_node(node& n, int add)
17  {
18    n.add+=add,
```

```
19        n.min+=add;
20   }
21   //range_query, update and range_update remain same.
```

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

```
1    void splitdown(int postn)
2    {
3          if(postn>1) splitdown(postn>>1);
4          tree[postn].split(tree[2*postn],tree[2*postn+1]);
5    }
6    void update(int postn, node nd)
7    {
8          postn += 1<<n;
9          splitdown(postn>>1);
10         tree[postn] = nd;
11         mergeup(postn);
12   }
```

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:

```
1    struct node
2    {
3          int numleaves, add, sum;
4          void split(node& l, node& r)
5          {
6                l.add += add;
7                l.sum += add * l.numleaves;
8                r.add += add;
9                r.sum += add * r.numleaves;
10               add=0;
11         }
12         void merge(node& l, node& r)
13         {
14               numleaves = l.numleaves + r.numleaves;
15               add = 0;
16               sum = l.sum + r.sum;
17         }
18   };
19   void update_single_subtree(node& n, int inc){
20         n.add += inc;
21         n.sum += inc * n.numleaves;
22   }
23   //range_query and range_update remain same as that for previous problems
```

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:

```cpp
1   struct node{
2           int num_active_leaves;
3           void split(node& l, node& r){}
4           void merge(node& l, node& r){
5                   num_active_leaves = l.num_active_leaves + r.num_active_leaves;
6           }
7           bool operator<(const node& n)const{
8                   return num_active_leaves < n.num_active_leaves;
9           }
10  };
11  int binary_search(node k)
12  //search the last place i, such that merge( everyting to the left of i(including i) ) compares less than k
13  {
14          int root = 1;
15          node n=identity; //identity satisfies merge(identity,y) = merge(y,identity) = y for all y.
16          assert(!(k<identity));
17          while(!isleaf(root)){
18                  int left_child = root<<1, right_child = left_child|1;
19                  tree[root].split(tree[left_child],tree[right_child]);
20                  node m;
21                  m.merge(n,tree[left_child]);
22                  if(m<k){//go to right side
23                          n=m;
24                          root=right_child;
25                  }else root=left_child;
26          }
27          node m;
28          m.merge(n,tree[root]);
29          mergeup(root);
30          if(m<k)return root-leftmost_leaf;
31          else return root-1-leftmost_leaf;
32  }
```

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   g+1 +32 Recommend this on Google

Labels: data structure, segment tree

86 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 !!!