The University of Hong Kong

COMP2123:
PROGRAMMING TECHNOLOGIES AND TOOLS

SELF LEARNING REPORT

Topic: Explaining and Implementing Segment Tree Data Structure in C++

Group Members	Student UID
Sudhakar Shah	3035345320
Devang Maskara	3035344948
Trisha Gupta	3035342419

GitHub Link of Repository: Click Here
Full Version controlled with additional features

Objective:

1. Reasons to use segment trees.
2. Learn the implementation of a segment tree.

We consider the following problem that will help us achieve the above objectives
Problem:
We have an array arr [0 . . . n-1].

1. We should be able to efficiently find the minimum value from index *qs* (query start) to *qe* (query end) where *0* <= *qs* <= *qe* <= *n-1*. We have p such queries for this.
2. We should be able to update an index in the array when asked to. To ease understanding of time complexity, we have p such queries for this as well.

Both n & p being relatively large ~ both are roughly equal to 10^5

Motivation:

A ***simple solution*** is to run a loop from *qs* to *qe* and find minimum element in given range. This solution takes O(n) time in worst case per query. In total to find the minimum element, it will take O(pn). For updating an index in the array, it will take O(1) time, per update. Total updates will take O(p) time.
Thus, the total *time complexity is O(pn + p) = O(pn) with no extra memory.*

Another solution is to create a 2D array of n^2 where an entry [i, j] stores the minimum value in range arr[i..j]. The pre-processing to make the array is $O(n^2)$. Minimum of a given range can now be calculated in O(1) time per query. In total, to find the minimum of a given range takes O(p) time.
For updating an index in the array, will take $O(n^2)$ time per update, as all positions where the index lies between i and j have to be updated. Total updates will take $O(pn^2)$ time.
Thus, the total *time complexity is* $O(n^2 + p + pn^2) = O(pn^2)$*. Also, this approach needs* $O(n^2)$ *extra space* which may become huge for large input arrays.

Segment Tree can be used to do pre-processing and query in moderate time. With segment tree, pre-processing time to build the segment tree is O(n) and time for range minimum query is O(log n), per query. In total, to find the minimum of a given range takes O(p*(log n)) time.
For updating an index in the array, it will take O(log n) time, per update. Total updates will take O(p*(log n)) time.
Thus, the *total time complexity is O(n + ((p+p)*(log n))) ~ O(p*(log n)). The extra space required is O(n)* to store the segment tree.

This is the motivation to use a segment tree for range queries.

A segment tree is a tree data structure which is mostly used when there are multiple range queries and updates on the same array. Once built, a segment tree can't be modified. Segment trees are used in sensor networks and a higher version called Fenwick tree is used in adaptive arithmetic coding.

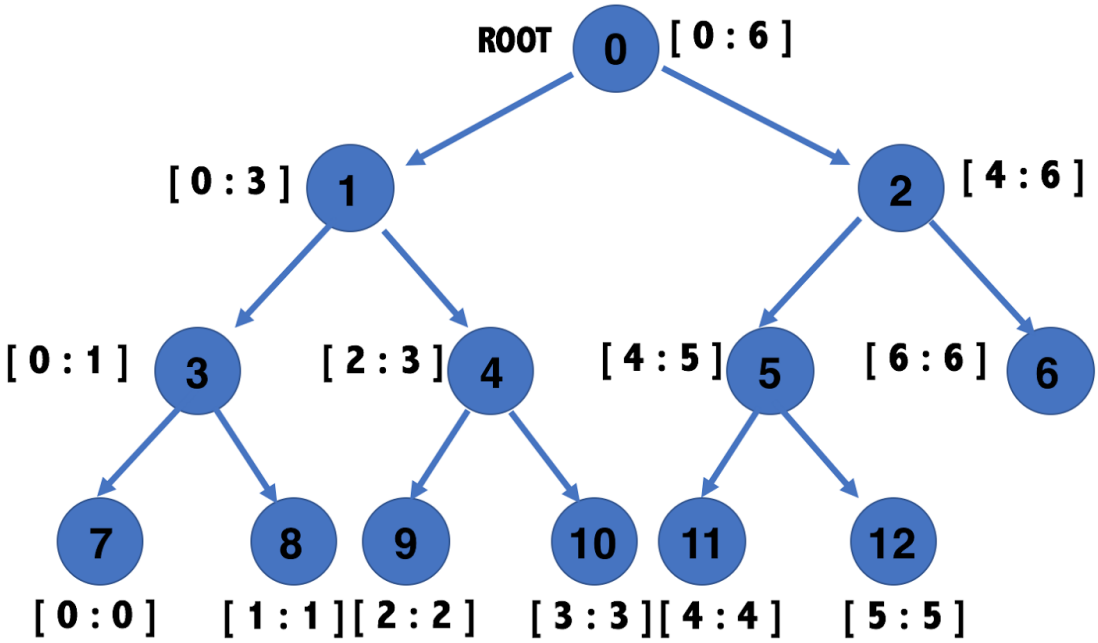
We will primarily concentrate on 3 functionalities of the segment tree:

1. Creating the segment tree - createTree.cpp
2. Getting the minimum value in a specified range - rangeQuery.cpp
3. Update a value in the input array and reflect the same in the segment tree - singleUpdate.cpp

Building a segment tree

A segment tree can be thought as a binary tree (a modified version of which we have learnt in lab 6.4). Each node stores the minimum value of an interval. The root of the tree represents the interval [0, n-1]. Each leaf node represents a single element at that position [i, i], 0 <= i <= n-1. The other internal nodes represent intervals [i, j] where
0 <= i < j <= n-1. Thus, each node stores the minimum value in the interval the node represents.

Main Array: input [], having the elements for which we need to find the minimum range
Length of input [] = n
For the simplicity of understanding, let's consider an input array of length n = 7.



We can build the tree using the bottom up approach of recursion, where we start with the leaves, since the minimum value of one element is the element itself and keep on moving upwards.

How to represent segment trees?
We will represent the tree as an Array! In the above figure, the number inside the nodes, represents the position of that node in the segment tree array as illustrated later, while the range written beside it shows that the node in the segment tree is holding the minimum of the shown range of the input array.

How can we draw the tree structure if we are given the segTree [] array?
Index 0 is the parent node of the tree
Left child of i: 2i+1 Right Child of i: 2i+2
For any given index, the parent of the node is floor[(i-1)/2]

Length of the array to store the segment tree (length of segTree [] array)?
Height of a segment tree which is a binary tree 'h' = ceil[$\log_2(n)$]
Thus, the length of the array storing the segment tree is $2*(2^h)-1$. Memory complexity of array storing segment tree ~ O(n)

Time complexity to create the array representation of the segment tree: O(n)

1. **Let's look at how to create the tree first - createTree.cpp:**

As mentioned earlier, we can use an array to represent the segment tree.

```
2  #include <iostream>
3  using namespace std;
4  void createTree(int input[],int segTree[],int start, int end, int pos)
5  {
```

The above method is used to create the required segment tree. The parameters are:

1. input []: the given array to us
2. segTree []: the array which stores our tree
3. pos: the current node in the segment tree which we are looking at
4. start and end: the starting and ending index of the range of pos, i.e. the current node we are looking at

Information for minimum of range [start,end] is present in segTree[pos].
Initially, while calling the method, start = 0 and end = n-1, i.e. the start and end index of the input array, and pos = 0 as the minimum of the range [0, n-1] is stored in the root, at position 0 of the segTree array.

Step 1: We check if (start == end)

```
6      // if only one element is left
7      if (start==end)
8      {
9          segTree[pos]=input[start];
10         return;
11     }
```

If starting and ending index of given range is the same means that we are only looking at one value in the input array. In which case, we simply store that one element in current node, i.e. one of the leaf nodes of the segment tree and return.

Step 2:

```
12     int mid=(start+end)/2;
13     //creating the left child
14     createTree(input,segTree,start,mid,2*pos+1);
15     //creating the right child
16     createTree(input,segTree,mid+1,end,2*pos+2);
17     segTree[pos]= min(segTree[2*pos+1],segTree[2*pos+2]);
18 }
```

- Else it means that we still have a range in that node, so we split the range into 2 halves of a binary tree and recur in the left and right half.
- Left tree - start to mid is left half, for which we send the pos as (2*pos)+1 as we store the minimum of start to mid index in left child, i.e. at segTree[2*pos+1].
- Right tree - Mid+1 to end is right half for which we send the pos as (2*pos)+2 as we store the minimum of mid+1 to end in right child, i.e. at segTree[2*pos+2].

- In current node (pos) we store the minimum of the left child and right child i.e. minimum of segTree[2*pos+1] (left child) and segTree[2*pos+2] (right child), which we get after the recursion.
- Now, this process continues until we reach our base condition, i.e. there is only one element in the given array, which represents the leaf nodes of the segment tree. In such a case, we simply store it as shown in step 1.

In the graph figure, the number inside the nodes, represents the position of that node in the segment tree array, while the range written beside it shows that that node in the segment tree is holding the minimum of the shown range of the input array. This illustrates how we use the bottom up approach of recursion to make the segment tree, as we go down till the leaf nodes and then use their values to find the minimum value of the ranges stored in the node above it.

2. Let's look at how to get the minimum value in a specified range - rangeQuery.cpp:

```
1  #include <iostream>
2  #include <limits.h>
3  using namespace std;
4  // creating a variable with large value, the maximum an integer can hold
5  int MAX=INT_MAX;
6  //Minimum for range [start,end] is present in segTree[pos], computed while creating the tree
7  int minimumQuery(int segTree[], int queryStart, int queryEnd , int start , int end, int pos)
8  {
```

int MAX: The max value an integer can store - include limits header file to get the value

The above method is used in getting the minimum value in a specified range of the input array. The parameters are:

1. segTree: The segment tree that we created using createTree.cpp.
2. queryStart and queryEnd: The starting index and ending index of the range in which query to find the minimum is performed.
3. start and end: the starting and ending index of the range of pos, i.e. the current node we are looking at
4. pos: the current node in the segment tree which we are looking at

Step 1:

```
9      //if there is a full overlap
10     if (queryStart <= start && queryEnd >= end)
11     {
12         return segTree[pos];
13     }
```

If there is full overlap, i.e. the range of which the current node has information about lies completely within the queried range, then we return the value of the current node. This value is the minimum of a subsequence of the queried range. Or we can say that the information of the node is useful for this particular query and is returned.

Step 2:

```
14      //if there is no overlap
15      else if (queryEnd < start || queryStart > end)
16      {
17          // returning a very large value
18          return MAX;
19      }
```

There is no overlap at all, i.e the range whose information the current node carries is completely outside the query range. In this case the information in this node as well as the subsequent children node are of no use to us. Hence, we return MAX - a very large value so that the search of minimum value is not affected by this return.

Step 3:

```
20      //if there is partial overlap
21      int mid= (start+end)/2;
22      return min(minimumQuery(segTree,queryStart,queryEnd,start,mid,2*pos+1),
23          minimumQuery(segTree,queryStart,queryEnd,mid+1,end,2*pos+2));
24  }
```

- There is a partial overlap between the range of the current node and the queried node. The partial overlap indicates that if we go deeper into the tree from this node, we can surely find a node whose range is completely inside the queried range.
- Left tree - start to mid is left half, whose minimum is there in segTree[2*pos+1].
- Right tree - Mid+1 to end is right half, whose minimum is there in segTree[2*pos+2].
- Our aim is find the nodes which have information that are completely useful to us (Step 1) i.e these nodes have a complete overlap with queried range.
- We find the minimum of the values returned by such node and return.

So as the recursion works, we find all the nodes which have information exactly about a particular subsequence of the queried range. Let's call such nodes as **important nodes** for this query.
We can also see that the union of range of important nodes is equal to the queried range. Hence minimum of the values of important nodes is indeed the minimum of the queried range.

3. Finally, let's look at how to update a value in the input array - reflecting the change in the segment tree - SingleUpdate.cpp

```
4   //Information for range [start,end] is present in segTree[pos]
5   void updateSegmentTree(int segTree[],int updatePos, int start , int end, int updateValue, int pos)
6   {
```

The above method is used to update a value in the input array and reflect the same in the segment tree. The parameters are:

1. segTree : The segment tree that we created using createTree.cpp
2. updatePos: The input array position that needs to be updated
3. start and end: the starting and ending index of the range of pos, i.e. the current node we are looking at.
4. updateValue: The new value of input [updatePos], the position to be updated
5. pos: index of the current node in the segment tree which we are looking at. This node has the minimum value out of input[i] where i is from [start, end]

Step 1:

```
7    //out of bounds
8    if (start > end)
9        return;
```

All the nodes in the tree have a range in which start<=end. So, if the start index is greater than the end index, then there is no node corresponding to this range and hence we return without doing anything.

Step 2:

```
10   // if the position to be updated is outside the range present in the node
11   if (updatePos < start || updatePos > end)
12       return;
```

If the position to be updated is outside the range of the node, then we are not concerned about this branch of the tree as well as its children, hence we return without doing anything else.

Step 3:

```
13   // if only one element i.e the leaf node then update it
14   if (start == end)
15   {
16       segTree[pos] = updateValue;
17       return;
18   }
```

Our aim is to find leaf node i.e the node that has the value of the position we want to update. Once we reach there, we update the value. Now, as we have used recurrence to reach till this leaf node, we move back upwards as the parent nodes which has this position in their range may be affected and make the required updates.

Step 4:

```
19      //carrying the same process of the children to get to the leaf node holding the posiiton to be updated
20      int mid= (start+end)/2;
21      updateSegmentTree(segTree,updatePos,start,mid,updateValue,2*pos+1);
22      updateSegmentTree(segTree,updatePos,mid+1,end,updateValue,2*pos+2);
23
24      // using the updated children to determine the value for the parent
25      segTree[pos]= min(segTree[2*pos+1],segTree[2*pos+2]);
26  }
```

If the position to be updated is inside the range of the current node, then the value of the node may be affected by the update. We recur the same function for its left and right children and continue our search for the lead node.

Conclusion

Our team hopes we have been able to give an insight on the implementation of a segment tree. These segment trees can be used in sensor networks (here) in real life. We can use this segment tree to learn more about Fenwick Trees which can be used to maintain cumulative sums which are using in adaptive arithmetic coding. Our team learnt a lot in this process and we hope this material will be beneficial to everyone in the course. For more information on range updates in a segment tree and lazy propagation to make these range updates faster, you can visit the version controlled code available on github – link. Happy Learning!

References:

https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/tutorial/

http://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/

http://codeforces.com/blog/entry/5954