Experiment No. 3

Title: Implementation of Segment Tree

Batch: B1 **Roll No:16010420133** **Experiment No.:3**

Aim: To study Segment Tree for implementation of problem statement that is based on multiple range queries and verify given test cases.

Resources needed: Text Editor, C/C++ IDE

<u>Theory</u>:

Segment Tree is used in cases where there are multiple range queries on array and modifications of elements of the same array. For example, finding the sum of all the elements in an array from indices L to R, or finding the minimum (famously known as Range Minimum Query problem) of all the elements in an array from indices L to R. These problems can be easily solved with one of the most versatile data structures, **Segment Tree**.

What is Segment Tree?

A Segment Tree is a data structure that allows answering range queries over an array effectively, while still being flexible enough to allow modifying the array. This includes finding the sum of consecutive array elements a[l…r], or finding the minimum element in a such a range in O(log*n*) time. Between answering such queries, the Segment Tree allows modifying the array by replacing one element, or even changing the elements of a whole sub-segment (e.g. assigning all elements a[l…r] to any value, or adding a value to all element in the sub-segment).

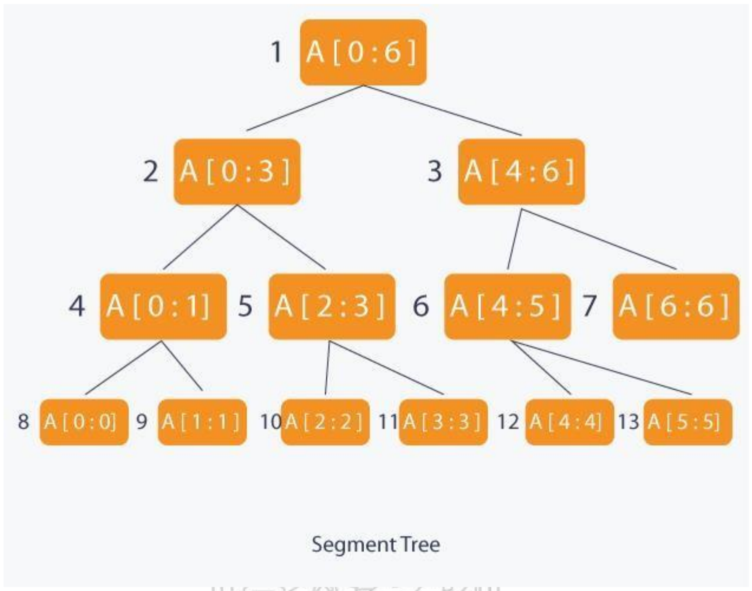
Segment Tree is a basically a binary tree used for storing the intervals or segments. Each node in the Segment Tree represents an interval. Consider an array A of size N and a corresponding Segment Tree T:

1. The root of T will represent the whole array A[0:N−1].
2. Each leaf in the Segment Tree T will represent a single element A[i] such that 0 ≤ I < N.
3. The internal nodes in the Segment Tree T represents the union of elementary intervals A[i:j] where 0 ≤ i < j < N.

The root of the Segment Tree represents the whole array A[0:N−1]. Then it is broken down into two half intervals or segments and the two children of the root in turn represent the A[0:(N−1)/2] and A[(N−1)/2+1:(N−1)]. So in each step, the segment is divided into half and the two children represent those two halves. So the height of the segment tree will be log2N.

There are N leaves representing the N elements of the array. The number of internal nodes is N−1. So, a total number of nodes are 2×N−1.

The Segment Tree of array A of size 7 will look like :



Segment Tree Construction -

Before constructing the segment tree, we need to decide:

1. The value that gets stored at each node of the segment tree. For example, in a sum segment tree, a node would store the sum of the elements in its range [l,r].
2. The merge operation that merges two siblings in a segment tree. For example, in a sum segment tree, the two nodes corresponding to the ranges a[l1…r1] and a[l2…r2] would be merged into a node corresponding to the range a[l1…r2] by adding the values of the two nodes.

A vertex is a "leaf vertex", if its corresponding segment covers only one value in the original array. It is present at the lowermost level of a segment tree. Its value would be equal to the (corresponding) element a[i].

Now, for construction of the segment tree, we start at the bottom level (the leaf vertices) and assign them their respective values. On the basis of these values, we can compute the values of the previous level, using the merge function. And on the basis of those, we can compute the values of the previous, and repeat the procedure until we reach the root vertex.

It is convenient to describe this operation recursively in the other direction, i.e., from the root vertex to the leaf vertices. The construction procedure, if called on a non-leaf vertex, does the following:

1. recursively construct the values of the two child vertices
2. merge the computed values of these children.

We start the construction at the root vertex, and hence, we are able to compute the entire segment tree.

Once the Segment Tree is built, its structure cannot be changed. We can update the values of nodes but we cannot change its structure. Segment tree provides two operations:

1. Update: To update the element of the array A and reflect the corresponding change in the Segment tree.
2. Query: In this operation we can query on an interval or segment and return the answer to the problem (say minimum/maximum/summation in the particular segment).

Implementation of Building Process:

```
void build(int node, int start, int end)
{
        if(start == end)
        {
             // Leaf node will have a single element
            tree[node] = A[start];
        }
        else
        {
            int mid = (start + end) / 2;
            // Recurse on the left child
            build(2*node, start, mid);
            // Recurse on the right child
            build(2*node+1, mid+1, end);
            // Internal node will have the sum of both of its children
            tree[node] = tree[2*node] + tree[2*node+1];
        }
}
```

Complexity of build() operation is – O(N)

Implementation of Update Process:

```
void update(int node, int start, int end, int idx, int val)
{
        if(start == end)
        {
              // Leaf node
              A[idx] += val;
              tree[node] += val;
        }
        else
        {
              int mid = (start + end) / 2;
              if(start <= idx and idx <= mid)
             {
                // If idx is in the left child, recurse on the left child
                update(2*node, start, mid, idx, val);
             }
             else
             {
                 // if idx is in the right child, recurse on the right child
                update(2*node+1, mid+1, end, idx, val);
             }
             // Internal node will have the sum of both of its children
             tree[node] = tree[2*node] + tree[2*node+1];
         }
 }
```

Complexity of update() is O(logN).

Implementation of Query Process:

```
int query(int node, int start, int end, int l, int r)
{
    if(r < start or end < l)
    {
        // range represented by a node is completely outside the given range
        return 0;
    }
    if(l <= start and end <= r)
    {
        // range represented by a node is completely inside the given range
        return tree[node];
    }
      // range represented by a node is partially inside and partially outside the given range
    int mid = (start + end) / 2;
    int p1 = query(2*node, start, mid, l, r);
    int p2 = query(2*node+1, mid+1, end, l, r);
    return (p1 + p2);
}
```

Complexity of query() is O(logN).

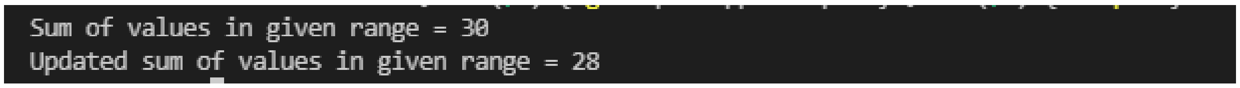
Activity:

Write a program to solve range-based query over an array for performing sum and update operation using segment tree.

Solution:

```
#include <bits/stdc++.h>
using namespace std;
int getMid(int s, int e)
{
     return s + (e -s)/2;
}
```

```
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    if (qs <= ss && qe >= se)
        return st[si];


    if (se < qs || ss > qe)
        return 0;


    int mid = getMid(ss, se);
    return getSumUtil(st, ss, mid, qs, qe, 2*si+1) +
        getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    if (i < ss || i > se)
        return;


    st[si] = st[si] + diff;


    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 2);
    }
}


void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    if (i < 0 || i > n-1)
    {
        cout<<"Invalid Input";
        return;
    }


    int diff = new_val - arr[i];
    arr[i] = new_val;

    updateValueUtil(st, 0, n-1, i, diff, 0);
}


int getSum(int *st, int n, int qs, int qe)
{


    if (qs < 0 || qe > n-1 || qs > qe)
    {
        cout<<"Invalid Input";
        return -1;
    }


    return getSumUtil(st, 0, n-1, qs, qe, 0);
}
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) +
            constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}



int *constructST(int arr[], int n)
{
    int x = (int)(ceil(log2(n)));
    int max_size = 2*(int)pow(2, x) - 1;
    int *st = new int[max_size];
    constructSTUtil(arr, 0, n-1, st, 0);
    return st;
}


int main()
{
    int arr[] = {0,2,4,6,8,10,12,14};
    int n = sizeof(arr)/sizeof(arr[0]);


    int *st = constructST(arr, n);
    cout<<"Sum of values in given range = "<<getSum(st, n, 1, 5)<<endl;
    updateValue(arr, st, n, 1, 10);
    cout<<"Updated sum of values in given range = "
            <<getSum(st, n, 1, 4)<<endl;
    return 0;
}
```



```
Sum of values in given range = 30
Updated sum of values in given range = 28
```

Outcomes:

Conclusion:

Thus, I have understood the concept of segment tree and its implementation.

References:

1. https://www.hackerearth.com/practice/data-structures/advanced-data-ructures/segment-trees/tutorial/
2. https://cp-algorithms.com/data_structures/segment_tree.html