Segment Tree ( Problem 1 )

We will see how to implement segment tree with the following example.

**Problem**

You are given an array **A** of **N** elements. Following operations are needed to be applied on the given array :

* Update an element.
* Get maximum element of a range.
* Get minimum element of a range.
* Get sum of all elements of a range.

**Structure**

It is clear that at each node we need to store maximum value, minimum value and sum of all the elements in the range of current node. Hence following will be the structure :

struct node

{

int sum, minimum, maximum;

} segTree[4\*N]; // size of segment tree is 4 times size of given input

As we know that segment tree can be stored in the form of array, therefore if current non leaf node is at index **i** than its left child's index will be **2\*i** and right child's index will be **2\*i+1**.

**Build Tree**

As we know building segment tree is bottom-up approach, so value(s) of a node depends upon the value(s) of its children.   
If we are at leaf node, that means only single element is present in that range. Hence minimum value, maximum value and sum of that range will be equal to that single element.   
For non leaf node :

* Sum of node = Sum of left child + Sum of right child
* Minimum of node = minimum ( Minimum of left child, Minimum of right child )
* Maximum of node = maximum ( Maximum of left child, Maximum of right child )

Following is code snippet to build segment tree :

// rangeLeft will be left most index of current range

// rangeRight will be right most index of current range

// index will be location of current node in array of segTree

void build\_tree (int rangeLeft, int rangeRight, int index)

{

if (rangeLeft == rangeRight) {

// we are at leaf node

segTree[index].sum = A[rangeLeft];

segTree[index].minimum = A[rangeLeft];

segTree[index].maximum = A[rangeLeft];

return;

}

int mid = (rangeLeft + rangeRight) / 2;

build\_tree(rangeLeft, mid, 2\*index);

build\_tree(mid+1, rangeRight, 2\*index+1);

segTree[index].sum = segTree[2\*index].sum + segTree[2\*index+1].sum;

segTree[index].minimum = min(segTree[2\*index].minimum, segTree[2\*index+1].minimum);

segTree[index].maximum = max(segTree[2\*index].maximum, segTree[2\*index+1].maximum);

}

**Update Tree**

We will start from root node and try to reach the leaf node of the index at which we need to update the value. After reaching leaf node we will update value of all the nodes which were visited during traversal from root to leaf.  
Following is code snippet to update segment tree :

// updateIndex will be the index whose value is needed to be updated in original array.

// newValue will be new value which is needed to be updated at updateIndex

// rangeLeft will be left most index of current range

// rangeRight will be right most index of current range

// index will be location of current node in array of segTree

void update\_tree (int updateIndex, int newValue, int rangeLeft, int rangeRight, int index)

{

if (rangeLeft <= updateIndex && updateIndex <= rangeRight) {

if (rangeLeft == rangeRight) {

// we are at leaf node

segTree[index].sum = newValue;

segTree[index].minimum = newValue;

segTree[index].maximum = newValue;

return;

}

int mid = (rangeLeft + rangeRight) / 2;

update\_tree(updateIndex, newValue, rangeLeft, mid, 2\*index);

update\_tree(updateIndex, newValue, mid+1, rangeRight, 2\*index+1);

segTree[index].sum = segTree[2\*index].sum + segTree[2\*index+1].sum;

segTree[index].minimum = min(segTree[2\*index].minimum, segTree[2\*index+1].minimum);

segTree[index].maximum = max(segTree[2\*index].maximum, segTree[2\*index+1].maximum);

}

}

**Query Tree**

* Start from root and check if current range completely lies within query range than return current node's required value.
* If current node completely lies outside the query range than return following values :
  + If query was to get sum of a range than return 0 as there is no element common in current node's range and query range.
  + If query was to get minimum of a range than return **∞** as there is no element common in current node's range and query range.
  + If query was to get maximum of a range than return **-∞** as there is no element common in current node's range and query range.
* Else query on current node's children, merge their answers and return.

Following is code snippet to query segment tree :

// queryLeft will be left most index of current query's range

// queryRight will be right most index of current query's range

// rangeLeft will be left most index of current range

// rangeRight will be right most index of current range

// index will be location of current node in array of segTree

int get\_sum(int queryLeft, int queryRight, int rangeLeft, int rangeRight, int index)

{

if (queryRight < rangeLeft || rangeRight < queryLeft || queryRight < queryLeft)

return 0;

if (queryLeft <= rangeLeft && rangeRight <= queryRight)

return segTree[index].sum;

int mid = (rangeLeft + rangeRight) / 2;

return get\_sum(queryLeft, queryRight, rangeLeft, mid, 2\*index) + get\_sum(queryLeft, queryRight, mid+1, rangeRight, 2\*index+1);

}

int get\_minimum(int queryLeft, int queryRight, int rangeLeft, int rangeRight, int index)

{

if (queryRight < rangeLeft || rangeRight < queryLeft || queryRight < queryLeft)

return INT\_MAX;

if (queryLeft <= rangeLeft && rangeRight <= queryRight)

return segTree[index].minimum;

int mid = (rangeLeft + rangeRight) / 2;

return min(get\_minimum(queryLeft, queryRight, rangeLeft, mid, 2\*index), get\_minimum(queryLeft, queryRight, mid+1, rangeRight, 2\*index+1));

}

int get\_maximum(int queryLeft, int queryRight, int rangeLeft, int rangeRight, int index)

{

if (queryRight < rangeLeft || rangeRight < queryLeft || queryRight < queryLeft)

return INT\_MIN;

if (queryLeft <= rangeLeft && rangeRight <= queryRight)

return segTree[index].maximum;

int mid = (rangeLeft + rangeRight) / 2;

return max(get\_maximum(queryLeft, queryRight, rangeLeft, mid, 2\*index), get\_maximum(queryLeft, queryRight, mid+1, rangeRight, 2\*index+1));

}