

# Segment Tree | Set 1 (Sum of given range)

Let us consider the following problem to understand Segment Trees.

We have an array  $arr[0 \dots n-1]$ . We should be able to

- 1 Find the sum of elements from index  $l$  to  $r$  where  $0 \leq l \leq r \leq n-1$
- 2 Change value of a specified element of the array  $arr[i] = x$  where  $0 \leq i \leq n-1$ .

A **simple solution** is to run a loop from  $l$  to  $r$  and calculate sum of elements in given range. To update a value, simply do  $arr[i] = x$ . The first operation takes  $O(n)$  time and second operation takes  $O(1)$  time.

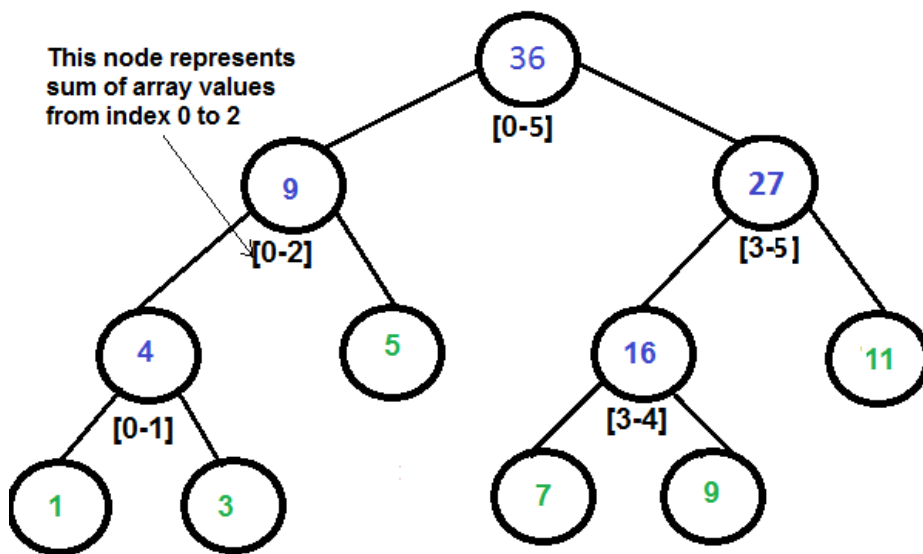
**Another solution** is to create another array and store sum from start to  $i$  at the  $i$ th index in this array. Sum of a given range can now be calculated in  $O(1)$  time, but update operation takes  $O(n)$  time now. This works well if the number of query operations are large and very few updates.

What if the number of query and updates are equal? **Can we perform both the operations in  $O(\log n)$  time once given the array?** We can use a Segment Tree to do both operations in  $O(\log n)$  time.

## Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index  $i$ , the left child is at index  $2*i+1$ , right child at  $2*i+2$  and the parent is at  $\lfloor (i-1)/2 \rfloor$ .



Segment Tree for input array {1, 3, 5, 7, 9, 11}

## Construction of Segment Tree from given array

We start with a segment  $arr[0 \dots n-1]$ . and every time we divide the current segment into two halves (if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment we store the sum in corresponding node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a **Full Binary Tree** because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with  $n$  leaves, there will be  $n-1$  internal nodes. So total number of nodes will be  $2*n - 1$ .

Height of the segment tree will be  $\lceil \log_2 n \rceil$ . Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be  $2 * 2^{\lceil \log_2 n \rceil} - 1$ .

## Query for Sum of given range

Once the tree is constructed, how to get the sum using the constructed segment tree. Following is algorithm to get the sum of elements.

```

int getSum(node, l, r)
{
    if range of node is within l and r
        return value in node
    else if range of node is completely outside l and r

```

}

## Update a value

Like tree construction and query operations, update can also be done recursively. We are given an index which needs to be updated. Let *diff* be the value to be added. We start from root of the segment tree, and add *diff* to all nodes which have given index in their range. If a node doesn't have given index in its range, we don't make any changes to that node.

## Implementation:

Following is implementation of segment tree. The program implements construction of segment tree for any given array. It also implements query and update operations.

```
#include <stdio.h>
#include <math.h>
```

```
// A utility function to get the middle index from corner
int getMid(int s, int e) { return s + (e -s)/2; }
```

```
/* A recursive function to get the sum of values in given array.
The following are parameters for this function.
```

```

st    --> Pointer to segment tree
index --> Index of current node in the segment tree.
        passed as root is always at index 0
ss & se --> Starting and ending indexes of the segment
        current node, i.e., st[index]
qs & qe --> Starting and ending indexes of query range

int getSumUtil(int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range,
    // sum of the segment
    if (qs <= ss && qe >= se)
        return st[index];
}

```

```

/* A recursive function to update the nodes which have the
their range. The following are parameters
    st, index, ss and se are same as getSumUtil()
    i    --> index of the element to be updated. This index is
    diff --> Value to be added to all nodes which have i in their
void updateValueUtil(int *st, int ss, int se, int i, int diff)
{
    // Base Case: If the input index lies outside the range
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then
    // of the node and its children
    st[index] = st[index] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*index + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*index + 2);
    }
}

// The function to update a value in input array and segment tree
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];
}

```

```

// Update the value in array
arr[i] = new_val;

// Update the values of nodes in segment tree
updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return sum of elements in range from index qs (query start)
// qe (query end). It mainly uses getSumUtil()
int getSum(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for
    // left and right subtrees and store the sum of values in this
    // node
    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];
}

/* Function to construct segment tree from given array. It
   allocates memory for segment tree and calls constructSTUtil
   to fill the allocated memory */
int *constructST(int arr[], int n)
{

```

```

// Allocate memory for segment tree
int x = (int)(ceil(log2(n))); //Height of segment tree
int max_size = 2*(int)pow(2, x) - 1; //Maximum size of array
int *st = new int[max_size];

// Fill the allocated memory st
constructSTUtil(arr, 0, n-1, st, 0);

// Return the constructed segment tree
return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %d\n", getSum(arr, st, n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding nodes
    updateValue(arr, st, n, 1, 10);

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %d\n", getSum(arr, st, n, 1, 3));

    return 0;
}

```

Output:

```

Sum of values in given range = 15
Updated sum of values in given range = 22

```

### Time Complexity:

Time Complexity for tree construction is  $O(n)$ . There are total  $2n-1$  nodes, and

value of every node is calculated only once in tree construction.

Time complexity to query is  $O(\log n)$ . To query a sum, we process at most four nodes at every level and number of levels is  $O(\log n)$ .

The time complexity of update is also  $O(\log n)$ . To update a leaf value, we process one node at every level and number of levels is  $O(\log n)$ .

## Segment Tree | Set 2 (Range Minimum Query)

### References:

<http://www.cse.iitk.ac.in/users/aca/lop12/slides/06.pdf>