

# Segment Tree

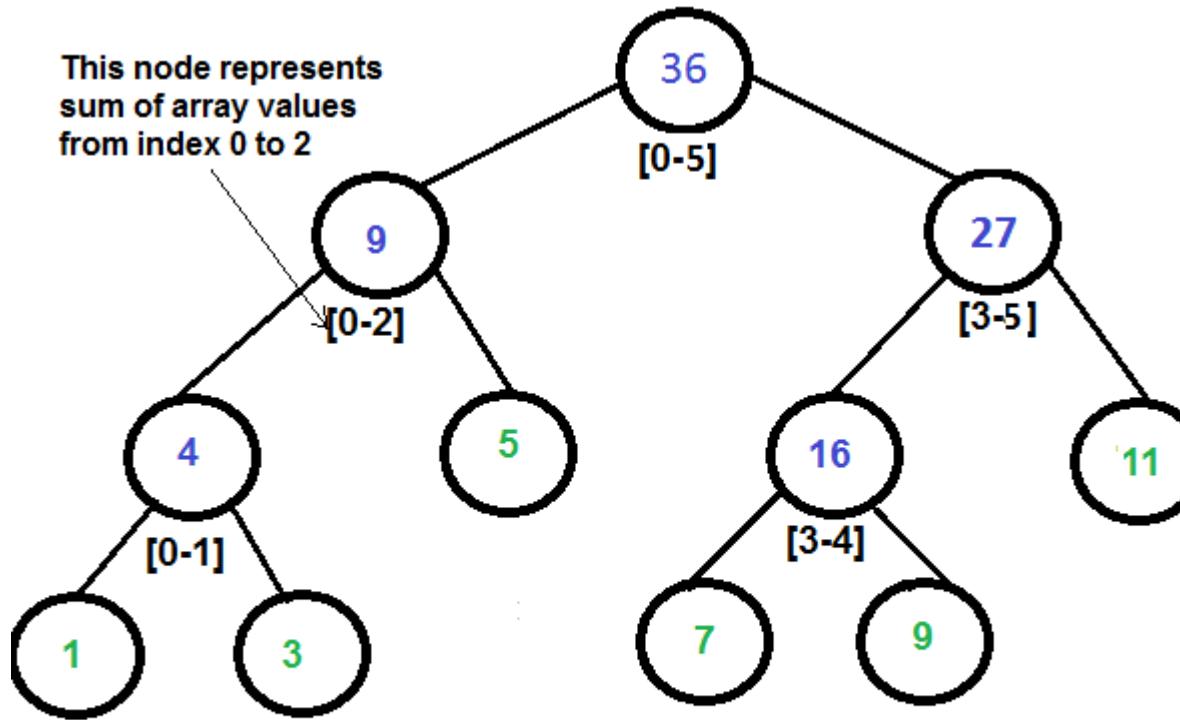
Suppose we have an array  $arr[0 \dots n - 1]$ .

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 to a new value  $x$ . We need to do  $arr[i] = x$  where  $0 \leq i \leq n - 1$ .

## **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.
3. An array representation of tree is used to represent Segment Trees.



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

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  $\lfloor \frac{i-1}{2} \rfloor$

## Construction of Segment Tree from given array:

1. We start with a segment  $arr[0 \dots n - 1]$ .
2. If it has not yet become a segment of length 1, divide the current segment into two halves and then call the same procedure on both halves.
3. For each such segment, we store the sum in the corresponding node.
4. All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree.
5. Since the constructed tree is always a full binary tree with  $n$  leaves, there will be  $n - 1$  internal nodes. Total number of nodes will be  $2 * n - 1$ .

## Query for Sum of given range:

```
int getSum(node, l, r)
{
    if the range of the node is within l and r
        return value in the node
    else if the range of the node is completely outside l and r
        return 0
    else
        return getSum(node's left child, l, r) +
               getSum(node's right child, l, r)
}
```

## Update a value:

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

```

// C program to show segment tree operations like construction, query
// and update
#include <stdio.h>
#include <math.h>

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

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

st    --> Pointer to segment tree
si    --> Index of current node in the segment tree. Initially
        0 is passed as root is always at index 0
ss & se  --> Starting and ending indexes of the segment represented
            by current node, i.e., st[si]
qs &qe   --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    // If segment of this node is a part of given range, then return
    // the sum of the segment
    if (qs <= ss && qe >= se)
        return st[si];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    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);
}

```

```

// Return sum of elements in range from index qs (query start)
// to 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 to update the nodes which have the given
index in their range. The following are parameters
st, si, ss and se are same as getSumUtil()
i    --> index of the element to be updated. This index is
        in the input array.
diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int si)
{
    // Base Case: If the input index lies outside the range of
    // this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update
    // the value of the node and its children
    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);
    }
}
```

```
// 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);
}
```

```
/* Function to construct segment tree from given array. This function  
allocates memory for segment tree and calls constructSTUtil() to  
fill the allocated memory */  
int *constructST(int arr[], int n)  
{  
    // Allocate memory for the segment tree  
  
    // Height of segment tree  
    int x = (int)(ceil(log2(n)));  
  
    // Maximum size of segment tree  
    int max_size = 2*(int)pow(2, x) - 1;  
  
    // Allocate memory  
    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;  
}
```

```
// A recursive function that constructs Segment Tree for array[ss..se].  
// 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 node of  
    // 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];  
}
```

```
// 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 = %dn",
           getSum(st, n, 1, 3));

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

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %dn",
           getSum(st, n, 1, 3));
    return 0;
}
```

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

## Time Complexity:

1. Time Complexity for tree construction is  $O(n)$ .
2. There are total  $2n - 1$  nodes, and value of every node is calculated only once in tree construction.
3. Time complexity to query is  $O(\log n)$ .
4. The time complexity of update is also  $O(\log n)$ .
5. To update a leaf value, we process one node at every level and number of levels is  $O(\log n)$ .