Segment Tree | Set 1 (Sum of given range)

```
// C++ program to show segment tree operations like construction, query
// and update
#include <bits/stdc++.h>
using namespace std;

// 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 the 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);
}

/* 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)
        {
                cout<<"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)
// 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)
        {
                cout<<"Invalid Input";
                return -1;
        }

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

// 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];
}

/* 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)
{
        //Height of segment tree
// 2^0,2^1…2^h where h is the height of the tree.
// n=2^(h-1) => h=ceil(log2(n))
        int x = (int)(ceil(log2(n)));
// It's a GP. Sz=2^(h+1)-1
        //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;
}

// 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
```

```
        cout<<"Sum of values in given range = "<<getSum(st, n, 1, 3)<<endl;

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

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

Segment tree | Efficient implementation

```
#include <bits/stdc++.h>
using namespace std;

// limit for array size
const int N = 100000;
int n; // array size
// Max size of tree
int tree[2 * N];

// function to build the tree
void build( int arr[])
{
        // insert leaf nodes in tree
        for (int i=0; i<n; i++)
                tree[n+i] = arr[i];

        // build the tree by calculating parents
        for (int i = n - 1; i > 0; --i)
                tree[i] = tree[i<<1] + tree[i<<1 | 1];
}
// function to update a tree node
void updateTreeNode(int p, int value)
{
        // set value at position p
        tree[p+n] = value;
        p = p+n;

        // move upward and update parents
        for (int i=p; i > 1; i >>= 1)
                tree[i>>1] = tree[i] + tree[i^1]; // for this logic to work the child of a node must be 2*i & 2*i+1 where i is the index of the parent. So the index of the root node must start from 1, unlike the previous case.
}
// function to get sum on interval [l, r)
int query(int l, int r)
{
        int res = 0;

        // loop to find the sum in the range
        for (l += n, r += n; l < r; l >>= 1, r >>= 1)
        {
                if (l&1)
                        res += tree[l++];

                if (r&1)
                        res += tree[--r];
        }

        return res;
}

// driver program to test the above function
int main()
{
        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

        // n is global
        n = sizeof(a)/sizeof(a[0]);

        // build tree
        build(a);
```

```
        // print the sum in range(1,2) index-based
        cout << query(1, 3)<<endl;

        // modify element at 2nd index
        updateTreeNode(2, 1);

        // print the sum in range(1,2) index-based
        cout << query(1, 3)<<endl;

        return 0;
}
```

Video Explanation -> https://youtu.be/Oq2E2yGadnU

Segment Tree can be used for any binary associative operation.

Lazy Propagation in Segment Tree

```
// Program to show segment tree to demonstrate lazy
// propagation
#include <stdio.h>
#include <math.h>
#define MAX 1000

// Ideally, we should not use global variables and large
// constant-sized arrays, we have done it here for simplicity.
int tree[MAX] = {0}; // To store segment tree
int lazy[MAX] = {0}; // To store pending updates

/* si -> index of current node in segment tree
      ss and se -> Starting and ending indexes of elements for
                           which current nodes stores sum.
      us and ue -> starting and ending indexes of update query
      diff -> which we need to add in the range us to ue */
void updateRangeUtil(int si, int ss, int se, int us,
                                    int ue, int diff)
{
      // If lazy value is non-zero for current node of segment
      // tree, then there are some pending updates. So we need
      // to make sure that the pending updates are done before
      // making new updates. Because this value may be used by
      // parent after recursive calls (See last line of this
      // function)
      if (lazy[si] != 0)
      {
            // Make pending updates using value stored in lazy
            // nodes
            tree[si] += (se-ss+1)*lazy[si];

            // checking if it is not leaf node because if
            // it is leaf node then we cannot go further
            if (ss != se)
            {
                  // We can postpone updating children we don't
                  // need their new values now.
                  // Since we are not yet updating children of si,
                  // we need to set lazy flags for the children
                  lazy[si*2 + 1] += lazy[si];
                  lazy[si*2 + 2] += lazy[si];
            }
```

```
            // Set the lazy value for current node as 0 as it
            // has been updated
            lazy[si] = 0;
      }

      // out of range
      if (ss>se || ss>ue || se<us)
            return ;

      // Current segment is fully in range
      if (ss>=us && se<=ue)
      {
            // Add the difference to current node
            tree[si] += (se-ss+1)*diff;

            // same logic for checking leaf node or not
            if (ss != se)
            {
                  // This is where we store values in lazy nodes,
                  // rather than updating the segment tree itelf
                  // Since we don't need these updated values now
                  // we postpone updates by storing values in lazy[]
                  lazy[si*2 + 1] += diff;
                  lazy[si*2 + 2] += diff;
            }
            return;
      }

      // If not completely in rang, but overlaps, recur for
      // children,
      int mid = (ss+se)/2;
      updateRangeUtil(si*2+1, ss, mid, us, ue, diff);
      updateRangeUtil(si*2+2, mid+1, se, us, ue, diff);

      // And use the result of children calls to update this
      // node
      tree[si] = tree[si*2+1] + tree[si*2+2];
}

// Function to update a range of values in segment
// tree
/* us and eu -> starting and ending indexes of update query
      ue -> ending index of update query
      diff -> which we need to add in the range us to ue */
void updateRange(int n, int us, int ue, int diff)
{
updateRangeUtil(0, 0, n-1, us, ue, diff);
}


/* A recursive function to get the sum of values in given
      range of the array. The following are parameters for
      this function.
      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., tree[si]
      qs & qe --> Starting and ending indexes of query
                        range */
```

```
int getSumUtil(int ss, int se, int qs, int qe, int si)
{
	// If lazy flag is set for current node of segment tree,
	// then there are some pending updates. So we need to
	// make sure that the pending updates are done before
	// processing the sub sum query
	if (lazy[si] != 0)
	{
		// Make pending updates to this node. Note that this
		// node represents sum of elements in arr[ss..se] and
		// all these elements must be increased by lazy[si]
		tree[si] += (se-ss+1)*lazy[si];

		// checking if it is not leaf node because if
		// it is leaf node then we cannot go further
		if (ss != se)
		{
			// Since we are not yet updating children os si,
			// we need to set lazy values for the children
			lazy[si*2+1] += lazy[si];
			lazy[si*2+2] += lazy[si];
		}

		// unset the lazy value for current node as it has
		// been updated
		lazy[si] = 0;
	}

	// Out of range
	if (ss>se || ss>qe || se<qs)
		return 0;

	// At this point we are sure that pending lazy updates
	// are done for current node. So we can return value
	// (same as it was for query in our previous post)

	// If this segment lies in range
	if (ss>=qs && se<=qe)
		return tree[si];

	// If a part of this segment overlaps with the given
	// range
	int mid = (ss + se)/2;
	return getSumUtil(ss, mid, qs, qe, 2*si+1) +
		getSumUtil(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 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(0, n-1, qs, qe, 0);
}
```

```
// A recursive function that constructs Segment Tree for
// array[ss..se]. si is index of current node in segment
// tree st.
void constructSTUtil(int arr[], int ss, int se, int si)
{
	// out of range as ss can never be greater than se
	if (ss > se)
		return ;

	// If there is one element in array, store it in
	// current node of segment tree and return
	if (ss == se)
	{
		tree[si] = arr[ss];
		return;
	}

	// 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 = (ss + se)/2;
	constructSTUtil(arr, ss, mid, si*2+1);
	constructSTUtil(arr, mid+1, se, si*2+2);

	tree[si] = tree[si*2 + 1] + tree[si*2 + 2];
}

/* Function to construct segment tree from given array.
This function allocates memory for segment tree and
calls constructSTUtil() to fill the allocated memory */
void constructST(int arr[], int n)
{
	// Fill the allocated memory st
	constructSTUtil(arr, 0, n-1, 0);
}


// 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
	constructST(arr, n);

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

	// Add 10 to all nodes at indexes from 1 to 5.
	updateRange(n, 1, 5, 10);

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

	return 0;
}
```