**Segment Trees**

Table of Contents

[Initialization 5](#_Toc66036348)

[Updating Values 8](#_Toc66036349)

[Making Queries 9](#_Toc66036350)

[Displaying the Tree 12](#_Toc66036351)

Say we have an array of integers with 10,000 items and we need to perform a large number of queries on them, say 10,000 queries. The queries are all of a certain type called a range-based query, i.e. they ask for some information about the members of the array within some range. For example, a query could ask to find the sum of all numbers within a range to , or the maximum value or the minimum value.

A simple approach to solving a range-based query is to run a loop over the range. To find the sum, we would add the values of all the members from to . However, this is a linear solution. When we have a very large number of queries, this will make the time complexity O(n2), since we essentially have to go over the range of length , times.

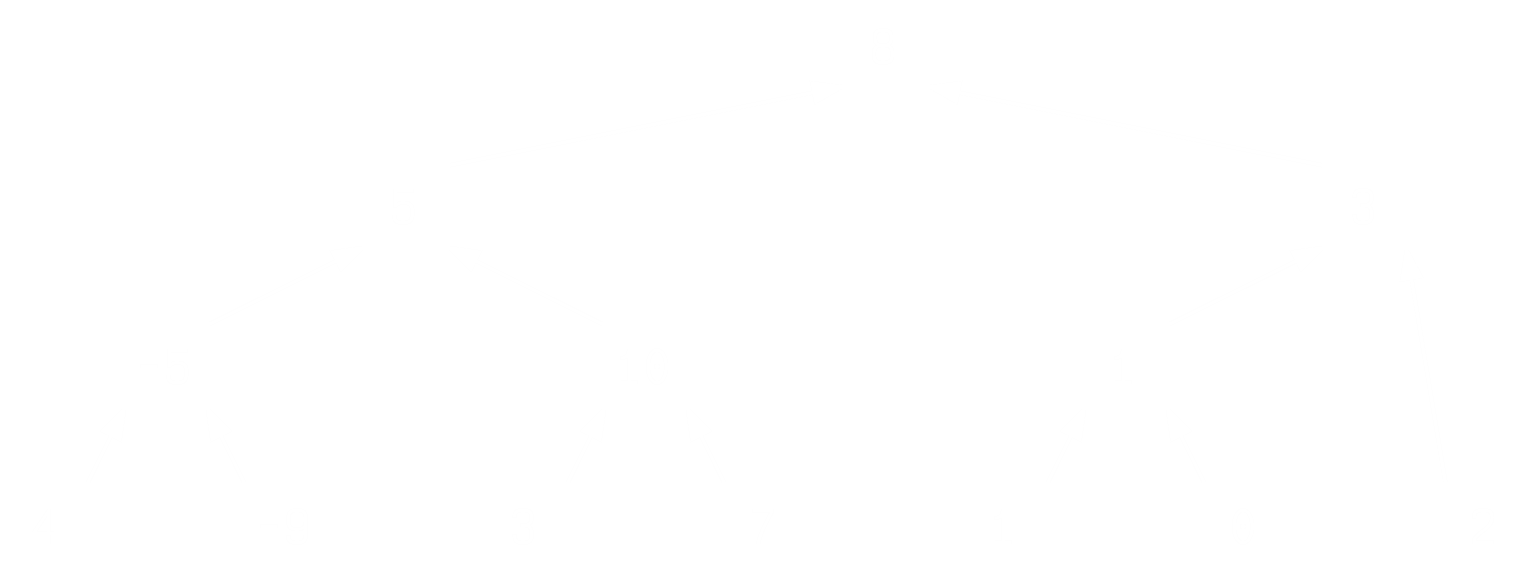
Another approach could be to keep track of the cumulative sum at every index position, so each index would also keep track of the total sum of all the members up to and including itself. To find the total sum within a range, we simply retrieve the cumulative sum value at the th index and subtract the cumulative sum value at the th index. In this case, performing the actual query has a time complexity of O(1), but setting up the array of cumulative sums has a time complexity of O(n). The problem here is if even one of the values in the array is changed, the entire array for cumulative sums needs to be updated. Thus, even this is a poor method to use if our system needs to be updated frequently.

Now to look at segment trees. Segment trees help solve range-minimum queries. They can get the maximum or minimum value in a range, and also the sum of all values in the range. It has a time complexity of O(log n). It can also update values maintaining the same time complexity, and even update multiple values within a range. The latter requires the use of Lazy-propagation, which will be studied later on.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | -9 | 3 | 7 | 1 | 0 | 2 |

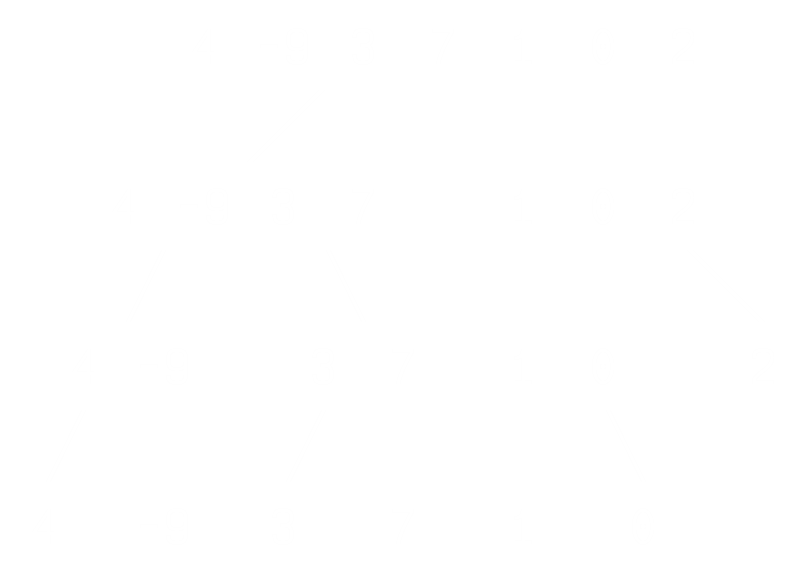
A segment tree, though it is stored in an array, can be visualized using a conceptual tree. This is similar to how we visualized heaps. The idea is that each of the values of the array are leaf nodes, and any intermediate nodes will be the result of the range-minimum query on the left and right subtrees.

Array:



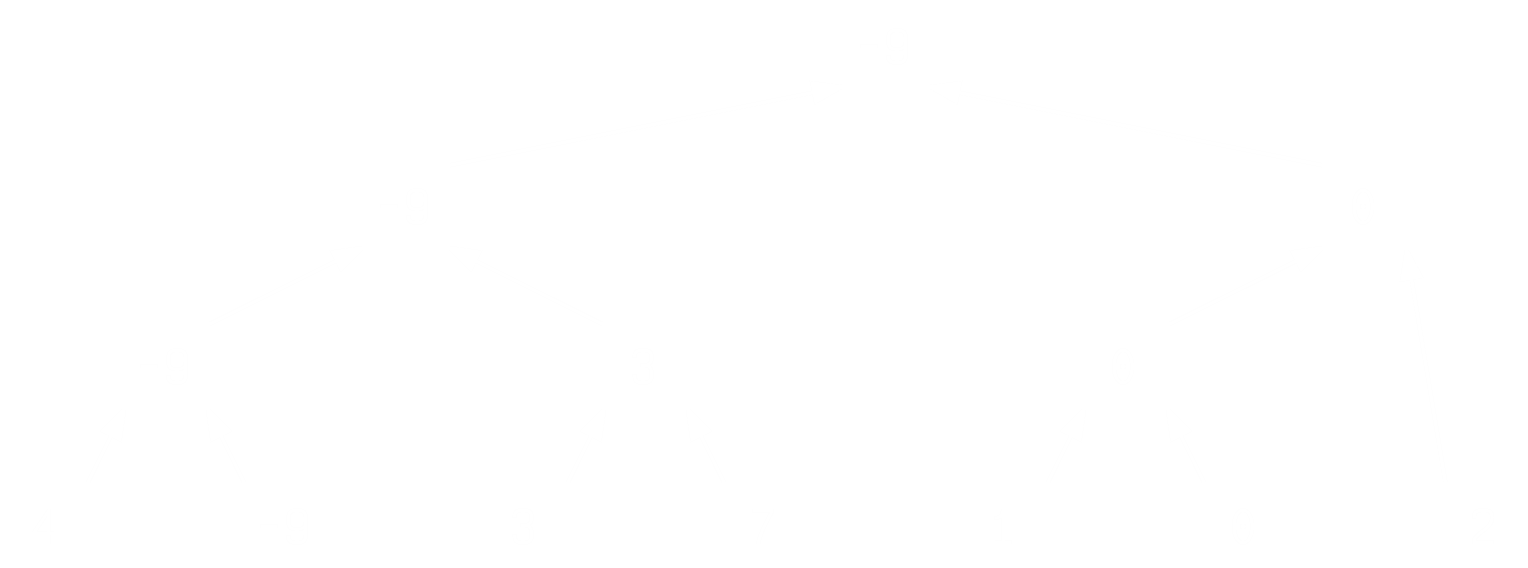
Here, the value of each intermediate node is the result of the summation of the left and right subtrees, i.e. the sum of a segment (hence the naming).

To be able to create this sort of structure, we need to divide our array in this manner:

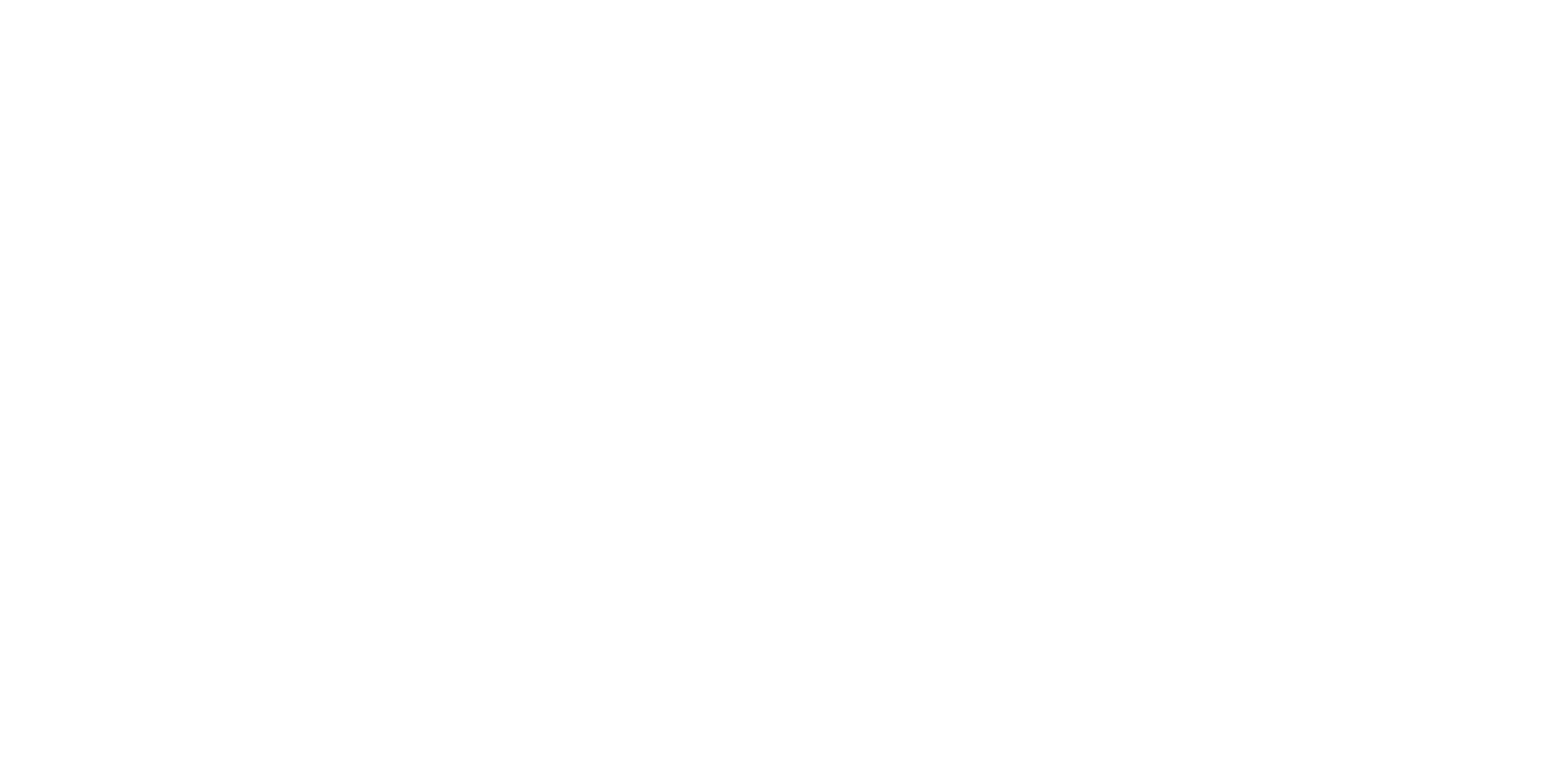


We have made partitions in the middle repeatedly until we are left with single items. Note that this is **not** a segment tree. It is just a figure to help us see how the division was done.

The segment tree we saw earlier was specifically for a scenario where the summation of the values in a range is of importance. If we had a different query, like the maximum or minimum value in a range, the segment tree would look entirely different. The thought process however, remains the same.



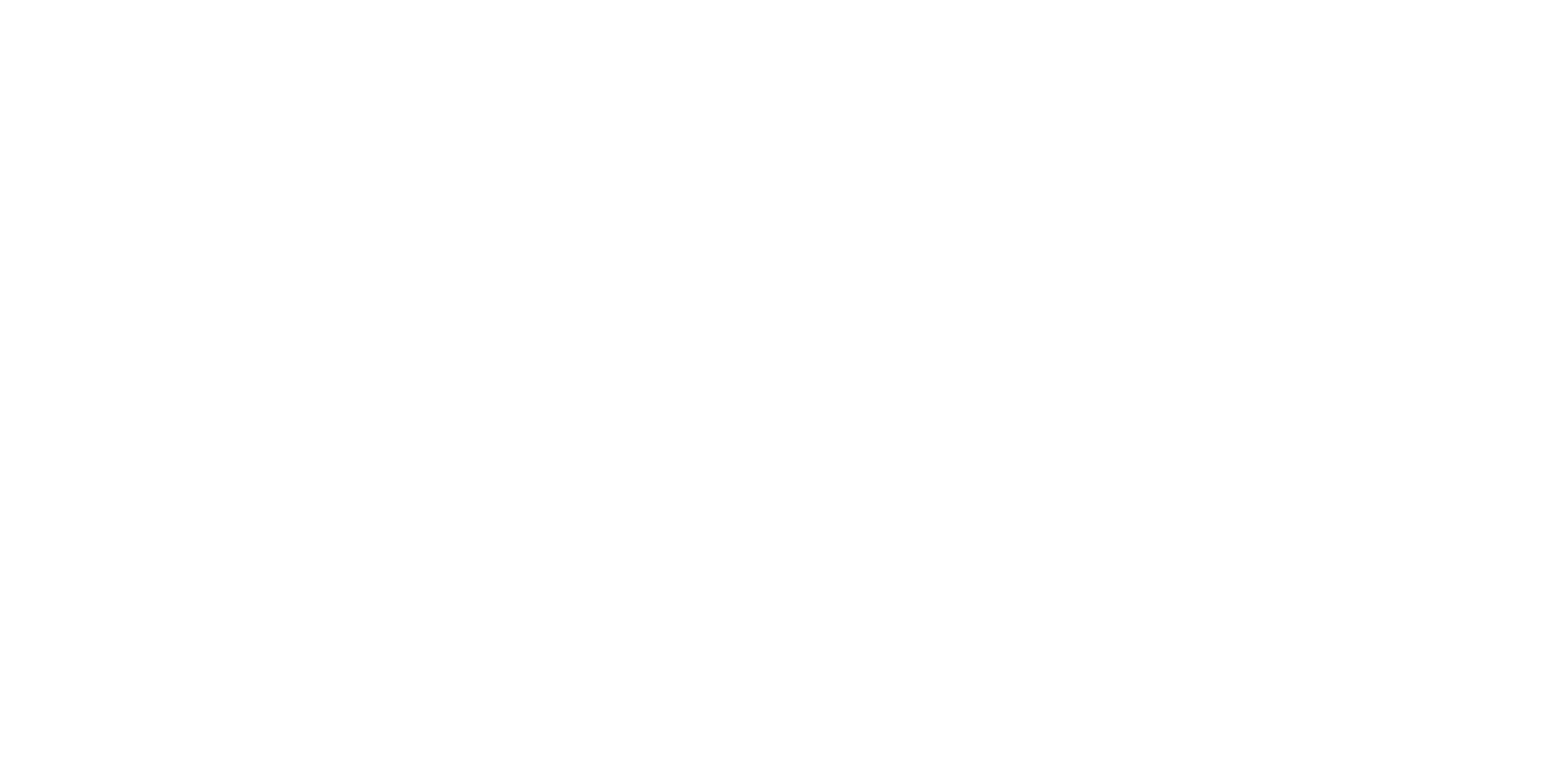
Since these are conceptual binary trees, we can store the entire tree in an array, with the root node being index 1.



Just like a heap, the index of a node can be found using these formulae

* Index of left child of a node = 2 x index of parent node
* Index of right child of a node = (2 x index of parent node) + 1
* Index of parent of a node = floor (index of either child / 2)

## Initialization



Before we can start building our segment tree, we need to make an important observation. Leaf nodes have the same start and end indices (from the original array) for their range. We can use this information to detect leaf nodes in our segment tree.

For every node from the top one, we find the value by adding up the values in its left and right children. To find these, we have to add up the values in each of their respective children and so on until we have leaf nodes, which have no children. Obviously, this requires a recursive algorithm. This will require a bottom-up approach.

int main()  
{  
 int n=7, m=(4\*n), arr[n+1], tree[m+1];  
 for (int i=1; i<=n; i++) cin>>arr[i]; *// taking input* init(1, 1, n, tree, arr); *// initializing segment tree*}

C++

From the main function, we are calling an init function with five parameters. The first number represents which node we are building from. The second number represents the beginning of the range. The third number represents the end of the range. The other two parameters are the original array and the array for the tree. Note that we are considering arrays to start from index 1, which also means our arrays have one index more than needed. Also note that the size of the array for the tree is four times the size of the original array. It has been found that the maximum index any leaf node will be at falls into this range, since not all indices are filled.

void init(int node, int begin, int end, int tree[], int arr[])  
{  
 if (begin == end) *// leaf node; insert directly* {  
 tree[node] = arr[begin];  
 return;  
 }  
 int leftChild = node \* 2; *// left child of tree node* int rightChild = (node \* 2) + 1; *// right child of tree node* int midPoint = (begin + end) / 2; *// mid-point of range* init(leftChild, begin, midPoint, tree, arr); *// initialize children* init(rightChild, midPoint + 1, end, tree, arr);  
  
 tree[node] = tree[leftChild] + tree[rightChild]; *// store sum*}

C++

Inside the init function, we first check if the node we are working with is a leaf node. If it is, we insert it directly into the tree array. If it is not a leaf node, then we need to perform calculations. First, we find the segment tree indices for the left and right children of the current node, and then we find the mid-point of the range. For the mid-point, we use the floor value. The left child will hold the result for the range from begin to midPoint and the right child will hold the result for the range from midPoint + 1 to end. Next, we call the init function on the left and right children, and we store their sum in the current node once they have been initialized.

The entire set of recursive function calls for the array we used as an example is given below. The stack is used in actual computers to maintain the recursive call order. Matching it with the visual tree should make things clear.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| init (1, 1, 7) | | | | |
|  | init (2, 1, 4) | | | |
|  | init (4, 1, 2) | | |
|  | init (8, 1, 1) | |
|  | tree[8] = arr[1] = 4 |
| init (9, 2, 2) | |
|  | tree[9] = arr[2] = -9 |
| tree[4] = tree[8] + tree[9] = -5 | |
| init (5, 3, 4) | | |
|  | init (10, 3, 3) | |
|  | tree[10] = arr[3] = 3 |
| init (11, 4, 4) | |
|  | tree[11] = arr[4] = 7 |
| tree[5] = tree[10] + tree[11] = 10 | |
| tree[2] = tree[4] + tree[5] = 5 | | |
| init (3, 5, 7) | | | |
|  | init (6, 5, 6) | | |
|  | init (12, 5, 5) | |
|  | tree[12] = arr[5] = 1 |
| init (13, 6, 6) | |
|  | tree[13] = arr[6] = 0 |
| tree[6] = tree[12] + tree[13] = 1 | |
| init (7, 7, 7) | | |
|  | tree[7] = arr[7] = 2 | |
| tree[3] = tree[6] + tree[7] = 3 | | |
| tree[1] = tree[2] + tree[3] = 8 | | | |

The time complexity is O(n log n). The log n comes from the height of the tree, and we have to traverse the height n times for n leaves. Of course, this is not exact, since we do not repeatedly traverse from the top of the tree to a leaf node for every leaf node. However, it is a roughly accurate estimation, which is what the time complexity is supposed to be.

## Updating Values

Say we need to update the value of nodes within range 3…3. Since the beginning and end of the range is the same, we must be updating a particular leaf node. Of course, it would be invalid to update anything else. Updating a leaf node will cause all its ancestors to be updated as well. However, the entire subtree will not be affected due to updating a single leaf node. Thus, the maximum time complexity is O(log n).

First, we need to locate the leaf node. At each node from the top, we will check if both the beginning and end of the range match the given index. If they do not, then we will check both its left and right children. If a node has a range that does not contain the required index, we will immediately return from that node since its subtrees cannot contain the index either. Once the node is located, we update its value and then keep returning upwards and updating its ancestor’s values in turn.

void update(int node, int begin, int end, int index, int tree[], int newValue)  
{  
 if (index > end || index < begin) return; *// out of range* if (begin == index && end == index)  
 {  
 tree[node] = newValue; *// node found* return;  
 }  
 *// node not found*  
 int leftChild = node \* 2;  
 int rightChild = (node \* 2) + 1;  
 int midPoint = (begin + end) / 2;  
 update(leftChild, begin, midPoint, index, tree, newValue); *// check left* update(rightChild, midPoint+1, end, index, tree, newValue);*// check right* tree[node] = tree[leftChild] + tree[rightChild]; *// update ancestor*}

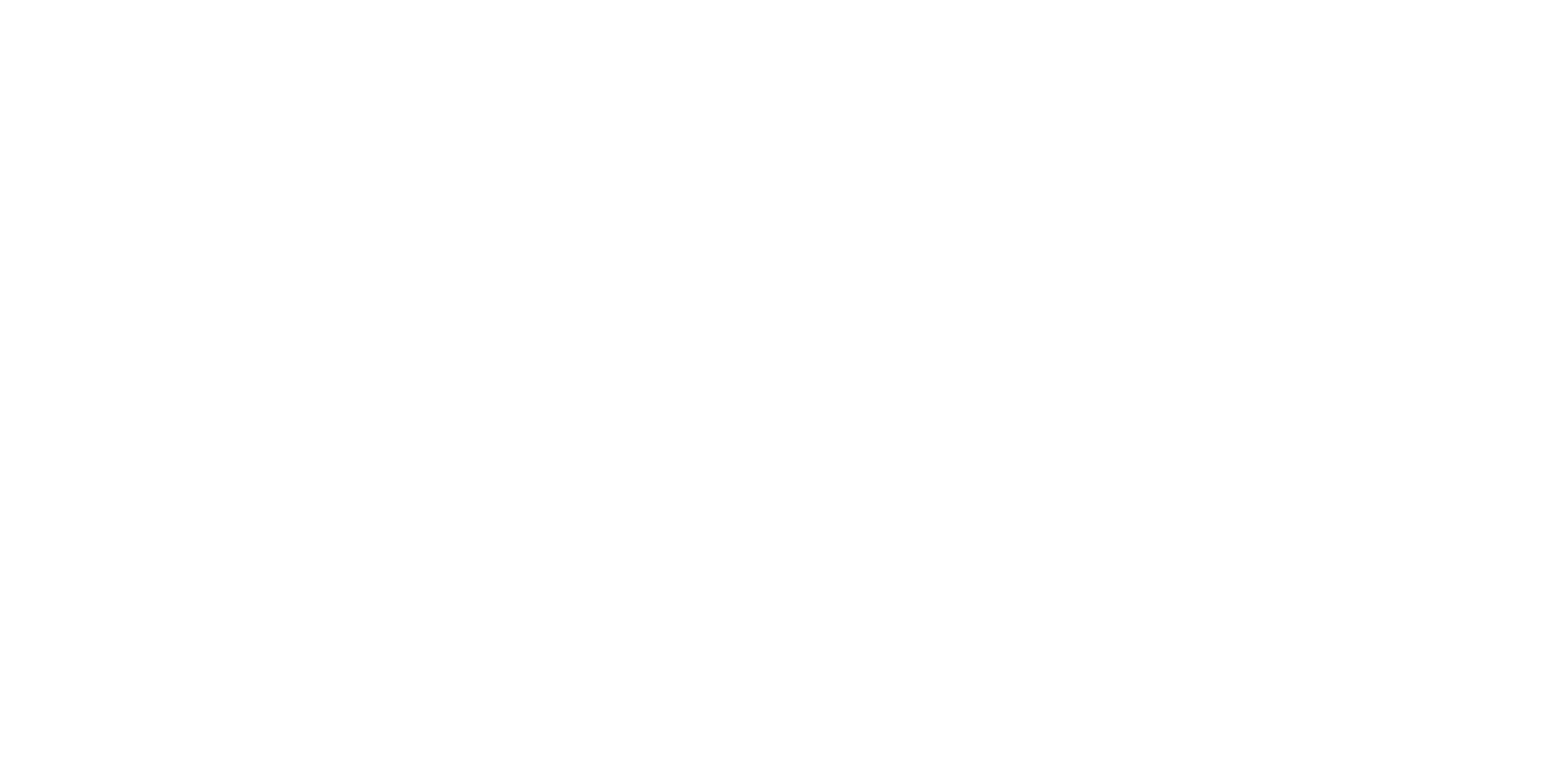
C++

From the main function, the update function is called like this:

update(1, 1, 7, 6, tree, 43);

C++

## Making Queries



If we want to know the sum for a range of values that is readily present in the tree, then it should be easy enough. However, we need to create a method that will allow us to find the summation in any range we want.

Say we want to find the summation of the values in the range 1 to 5. Starting from the root, we can see that the root has a range that is larger than the required range. Thus, we need to check its left and right subtrees.

Checking the left subtree (index 2), we can see that its range falls completely within the required range. This is called a complete intersection. In this case, we do not need to go further. We can simply return the value at this position and go back to the parent.

Checking the right subtree (index 3), we can see that its range falls partially within the required range. This is called a partial intersection. In this case, we need to go further.

From index 3, checking the left subtree, we can see another partial intersection so we need to go to the left child (index 6).

From index 6, the left subtree has a complete intersection and simply returns its value. The right subtree has no intersection and thus returns 0. If we were looking for the maximum or minimum value, the right subtree would return the minimum or maximum allowed integer value respectively instead, so as to not affect the results. Thus, index 6 returns the sum of the values returned by its left and right subtree, which is 1 + 0 = 1. We can now return to index 3.

Back at index 3, we check the right subtree to find that there is no intersection. Thus, index 3 will return 1 + 0 = 1. We can now go back to index 1.

At index 1, we have checked both left and right subtrees and can now return the result which is 5 from index 2 added with 1 from index 3, thus 6.

For no intersection, both the beginning and end of the index’s range is outside the required range. For partial intersections, either the beginning or the end is outside the required range. For complete intersections, both are inside the required range. These three statements will help us create the function we need.

int sumQuery (int node, int rangeBegin, int rangeEnd, int sumBegin, int sumEnd, int tree[])  
{  
 *// no intersection*

if (rangeBegin > sumEnd || rangeEnd < sumBegin) return 0; *// complete intersection* if (rangeBegin >= sumBegin && rangeEnd <= sumEnd) return tree[node];  
 *// partial intersection; return sum of left and right subtrees* int leftChild = node \* 2;  
 int rightChild = (node \* 2) + 1;  
 int midPoint = (rangeBegin + rangeEnd) / 2;  
 return (sumQuery(leftChild, rangeBegin, midPoint, sumBegin, sumEnd, tree) + sumQuery(rightChild, midPoint + 1, rangeEnd, sumBegin, sumEnd, tree));  
}

C++

From the main function, we call the query like this:

cout<<sumQuery(1, 1, 7, 1, 5, tree);

C++

## Displaying the Tree

The display function for the tree uses the same traversal method that the initialization function used. We simply keep going downwards until we locate a leaf node, we print the leaf node first, and then we go back upwards and print its ancestors. This uses the post-order traversal method. We could of course use a different traversal method simply by switching the positions of the print statements.

Note that if we tried to print the entire array for the tree instead of following this traversal pattern, we would end up printing a few garbage values. This is because there are indices in the array that were unused.

*// post-order traversal*

void display(int node, int begin, int end, int tree[], int treeLength){  
 if (begin == end) *// leaf node; display and return* {  
 cout<<tree[node]<<" ";  
 return;  
 }  
 int leftChild = node \* 2; *// left child of tree node* int rightChild = (node \* 2) + 1; *// right child of tree node* int midPoint = (begin + end) / 2; *// mid-point of range* display(leftChild, begin, midPoint, tree, treeLength);  
 display(rightChild, midPoint + 1, end, tree, treeLength);  
 cout<<tree[node]<<" ";  
}

C++