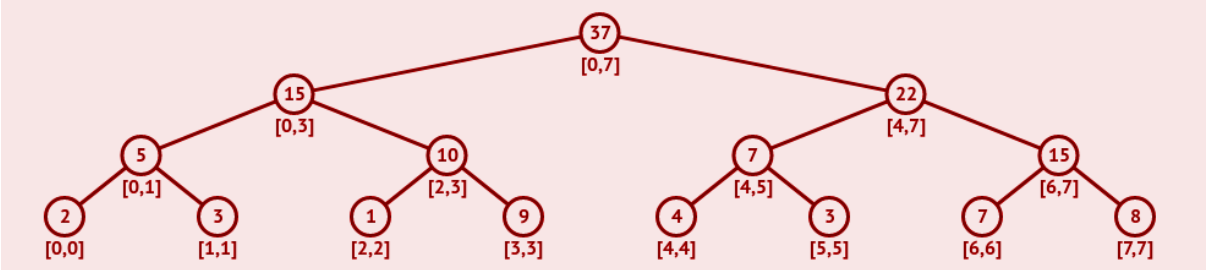
Implementation

Since a Segment Tree is a binary tree, a simple linear array can be used to represent the Segment Tree. Before building the Segment Tree, one must figure out what needs to be stored in the Segment Tree's node.
For example, if the question is to find the sum of all the elements in an array from indices L to R, then at each node (except leaf nodes) the sum of its children nodes is stored, which in this case will represent the sum of the elements from range L to R

For example A = [2, 3, 1, 9, 4, 3, , 7, 8]
The Segment Tree of array A of size 8 will look like this:



- **Construction of a segment tree**
  Let **tree** denote the segment tree array, **start** and **end** denote the interval length, and **node** denote the index of the tree array being processed.

```
function construct (node, start, end, tree)
{
      if(start == end)
      {
          //  Leaf node will have a single element
          tree[node] = A[start]
      }
      else
      {
          int mid = (start + end) / 2
          //  Recurse on the left child(left child index of any node = 2*node)
          construct(2*node, start, mid)
          //  Recurse on the right child(right child index of any node = 2*node + 1)
          construct(2*node+1, mid+1, end)
```

```
            //   Internal node will have the sum of both of its children
            tree[node] = tree[2*node] + tree[2*node+1]
        }
}
```

Time Complexity: O(n). There are total 2n-1 nodes, and the value of every node is calculated only once in tree construction.

There are two types of queries:

1. **Update**: Given idx and val , update array element A[idx] as A[idx] = A[idx]+val.
2. **Query**: Given an r, return the value of A[l]+A[l+1]+A[l+2]+.....+A[r−1]+A[r] such that ≤l≤r<N.

Queries and Updates can be in any order.

- **Update**: To update an element, look at the interval in which the element is present and recurse accordingly on the left or the right child.

```
function update( node, start, end, idx, val, tree)
{
        if(start == end)
        {
                //   Leaf node
                A[idx] += val
                tree[node] += val
        }
        else
        {
                int mid = (start + end) / 2
                if(start <= idx and idx <= mid)
                {
                        // If idx is in the left child i.e. between left and mid index
                        // , recurse on the left child
                        update(2*node, start, mid, idx, val, tree)
                }
```

```
                else
                {
                        // if idx is in the right child i.e. between mid+1 and right
                        // index, recurse on the right child
                        update(2*node+1, mid+1, end, idx, val, tree)
                }

                // Internal node will have the sum of both of its children
                tree[node] = tree[2*node] + tree[2*node+1]
        }
}
```

Time Complexity: O(logn). To update a value, we process one node at every level, and the number of levels is O(Logn).

- **Query:** To query on a given range, check 3 conditions.
  - The range represented by a node is completely inside the given range, return the value of the node which is the sum of all the elements in the range represented by the node
  - The range represented by a node is completely outside the given range, then return 0, as this range will not contribute to the sum of elements corresponding to the given range.
  - The range represented by a node is partially inside and partially outside the given range, return the sum of the left child and the right child.

```
//   start and end represent the segment tree range and l and r represent the query range.
function query(node, start,end, l, r, tree)
{
      if(r < start or end < l)
      {
            //   range represented by a node is completely outside the given range
            return 0
       }
```

```
    if(l <= start and end <= r)
    {
        //   range represented by a node is completely inside the given range
        return tree[node]
    }
    // range represented by a node is partially inside and partially outside the given range
    mid = (start + end) / 2
    p1 = query(2*node, start, mid, l, r, tree)
    p2 = query(2*node+1, mid+1, end, l, r, tree)
    return (p1 + p2)
}
```

Time Complexity: O(logn). To query a sum, we process at most four nodes at every level and the number of levels is O(Logn).