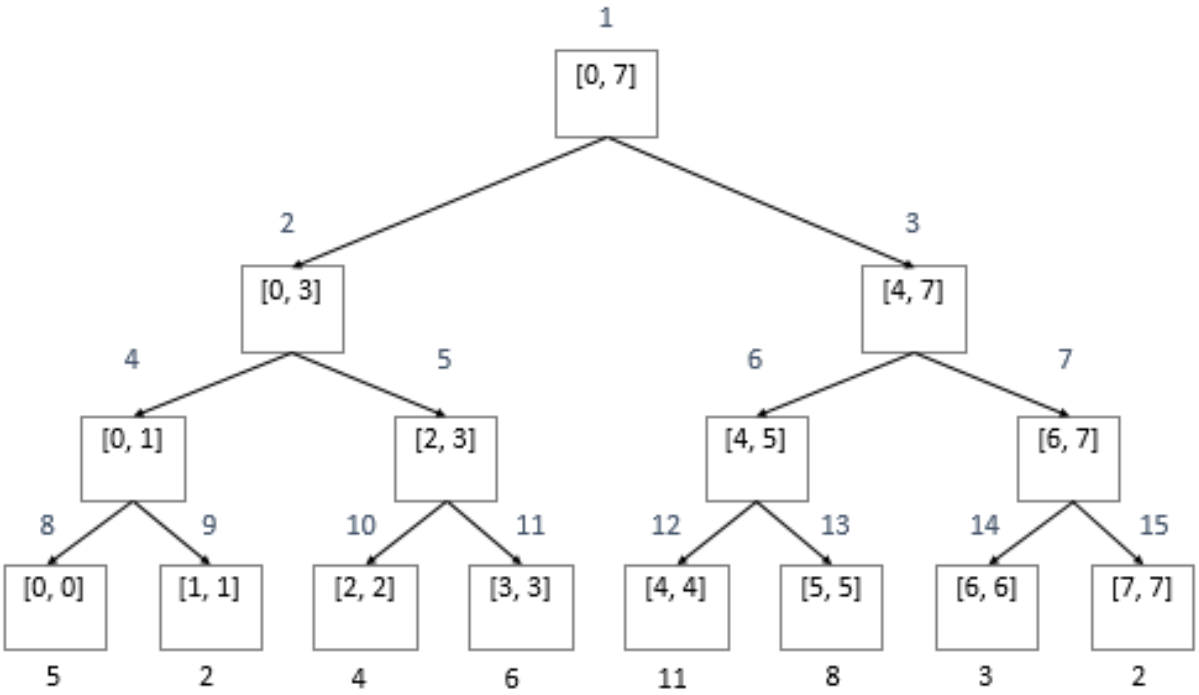
Segment Tree



Segment Tree Data Structure

```
public class SegmentTreeNode {
    int start, end;
    SegmentTreeNode left, right;

    public SegmentTreeNode(int start, int end) {
        this.start = start;
        this.end = end;
        this.left = this.right = null;
    }
}
```

Base on different problems, we can add extra instance variables in the Segment Tree Node class.

Start – start index (inclusive)

End – end index (inclusive)

Left – left sub-tree

Right – right sub-tree

Segment Tree Build

1. No extra instance variables (LintCode 201)

```
/**
 * Build the segment tree
 * @param start
 * @param end - Denote an segment / interval
 * @return The root of the segment tree.
 * @timecomplexity O(n) - create once for each node.
 */
public static SegmentTreeNode build(int start, int end) {
      if (start > end) {
            return null;
      }

      SegmentTreeNode rv = new SegmentTreeNode(start, end);

      if (start == end) {
            return rv;
      }

      int mid = start + (end - start) / 2;
      SegmentTreeNode leftTree = build(start, mid);
      SegmentTreeNode rightTree = build(mid + 1, end);
      rv.left = leftTree;
      rv.right = rightTree;

      return rv;
}
```

2. Given an array, implement a build method with a given array, so that we can create a corresponding segment tree with every node value represent the corresponding interval max value in the array, return the root of this segment tree. (LintCode 439)

```
 public class SegmentTreeNode {
      int start, end, max;
      SegmentTreeNode left, right;

      public SegmentTreeNode(int start, int end, int max) {
           this.start = start;
           this.end = end;
           this.max = max;
           this.left = this.right = null;
      }
 }



 /**
  * Create a corresponding segment tree with
  * every node value represent
  * the corresponding interval max value in the array
  * @param A - a list of integer
  * @return The root of Segment Tree
  */
public static SegmentTreeNode build(int[] A) {
    return build(A, 0, A.length - 1);
}

private static SegmentTreeNode build(int[] A, int start, int end)
{
 if (start > end || start < 0 || end >= A.length) {
      return null;
 }

 if (start == end) {
      return new SegmentTreeNode(start, end, A[start]);
 }

 int mid = start + (end - start) / 2;
 SegmentTreeNode leftTree = build(A, start, mid);
 SegmentTreeNode rightTree = build(A, mid + 1, end);
 int max = Math.max(leftTree.max, rightTree.max);
 SegmentTreeNode rv = new SegmentTreeNode(start, end, max);
 rv.left = leftTree;
 rv.right = rightTree;
 return rv;
}
```

Segment Tree Query

1. Design a query method with three parameters root, start and end, find the maximum number in the interval [start, end] by the given root of segment tree. (LintCode 202)

```
/**
 * Find the maximum number in the interval [start, end]
 * by the given root of segment tree.
 * @param root - The root of segment tree
 * @param start
 * @param end - an segment / interval
 * @return The maximum number in the interval [start, end]
 * @timecomplexity - O(logn)
 */
public static int query(SegmentTreeNode root, int start, int end)
{
   if (start > end || start < root.start || end > root.end) {
      return -1;//Wrong input.
   }

   if (root.start == root.end) {
      return root.max;
   }

   int mid = root.start + (root.end - root.start) / 2;
   if (end <= mid) {
      return query(root.left, start, end);
   } else if (start > mid) {
      return query(root.right, start, end);
   } else {
      int leftMax = query(root.left, start, mid);
      int rightMax = query(root.right, mid + 1, end);
      return Math.max(leftMax, rightMax);
   }
}
```

2. For an array, we can build a Segment Tree for it, each node stores an extra attribute **count** to denote the number of elements in the array which value is between interval start and end. (The array may not fully filled by elements)

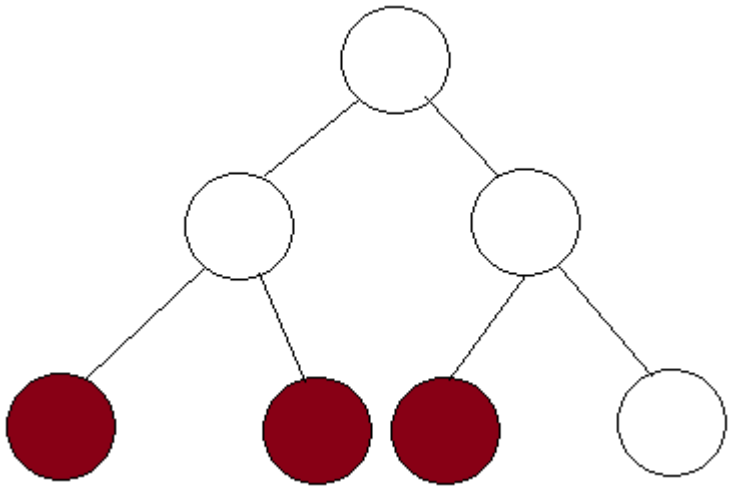
   Design a **query** method with three parameters **root**, **start** and **end**, find the number of elements in the in array's interval [*start*, *end*] by the given root of value Segment Tree. (LintCode 247)

```
/**
 * For an array, we can build a SegmentTree for it,
 * each node stores an extra attribute count
 * to denote the number of elements in the the array
 * which value is between interval start and end.
 * Find the number of elements in the interval [start, end]
 * by the given root of value SegmentTree.
 * @param root - The root of segment tree
 * @param start
 * @param end - an segment / interval
 * @return The count number in the interval [start, end]
 */
public static int query(SegmentTreeNode root, int start, int end)
{
 //Initially deal with Wrong Input
 if (root == null || start > end) {
       return 0;
    }
    start = start < root.start ? root.start : start;
    end = end > root.end ? root.end : end;

    if (root.start == start && root.end == end) {
       return root.count;
    }

    int mid = root.start + (root.end - root.start) / 2;
    if (start > mid) {
       return query(root.right, start, end);
    } else if (end <= mid) {
       return query(root.left, start, end);
    } else {
       return query(root.left, start, mid) +
                   query(root.right, mid + 1, end);
    }
 }
```

Query Time Complexity: O(logn)

The claim is that there are at most 2 nodes which are expanded at each level. We will prove this by contradiction.

Consider the segment tree given below.



Let's say that there are 33 nodes that are expanded in this tree. This means that the range is from the left most colored node to the right most colored node. But notice that if the range extends to the right most node, then the full range of the middle node is covered. Thus, this node will immediately return the value and won't be expanded. Thus, we prove that at each level, we expand at most 2 nodes and since there are logn levels, the nodes that are expanded are $2 \cdot \log n = \Theta(\log n)$.
(Reference: https://cs.stackexchange.com/questions/37669/time-complexity-proof-for-segment-tree-implementation-of-the-ranged-sum-problem/39594#39594?newreg=66ea671b3e0245179116ef52fbe753cf)

Segment Tree Modify

For a **Maximum Segment Tree**, which each node has an extra value **max** to store the maximum value in this node's interval.

Implement a **modify** function with three parameter **root**, **index** and **value** to change the node's value with ***[start, end] = [index, index]*** to the new given value. Make sure after this change, every node in segment tree still has the **max** attribute with the correct value. (LintCode 203)

```
/**
 * Change the node's value with [start, end] = [index, index]
 * to the new given value.
 * Make sure after this change,
 * every node in segment tree still has the max attribute
 * with the correct value.
 * @param root - The root of segment tree
 * @param index
 * @param value - change the node's value with [index, index]
 * to the new given value.
 * @timecomplexity - O(logn)
 */
public static void modify(SegmentTreeNode root, int index, int value)
{
  if (index < root.start || index > root.end) {
      return;
  }

  if (root.start == root.end) {
      root.max = value;
      return;
  }

  int mid = root.start + (root.end - root.start) / 2;
  if (index <= mid) {
      modify(root.left, index, value);
  } else {
      modify(root.right, index, value);
  }

  root.max = Math.max(root.left.max, root.right.max);
}
```