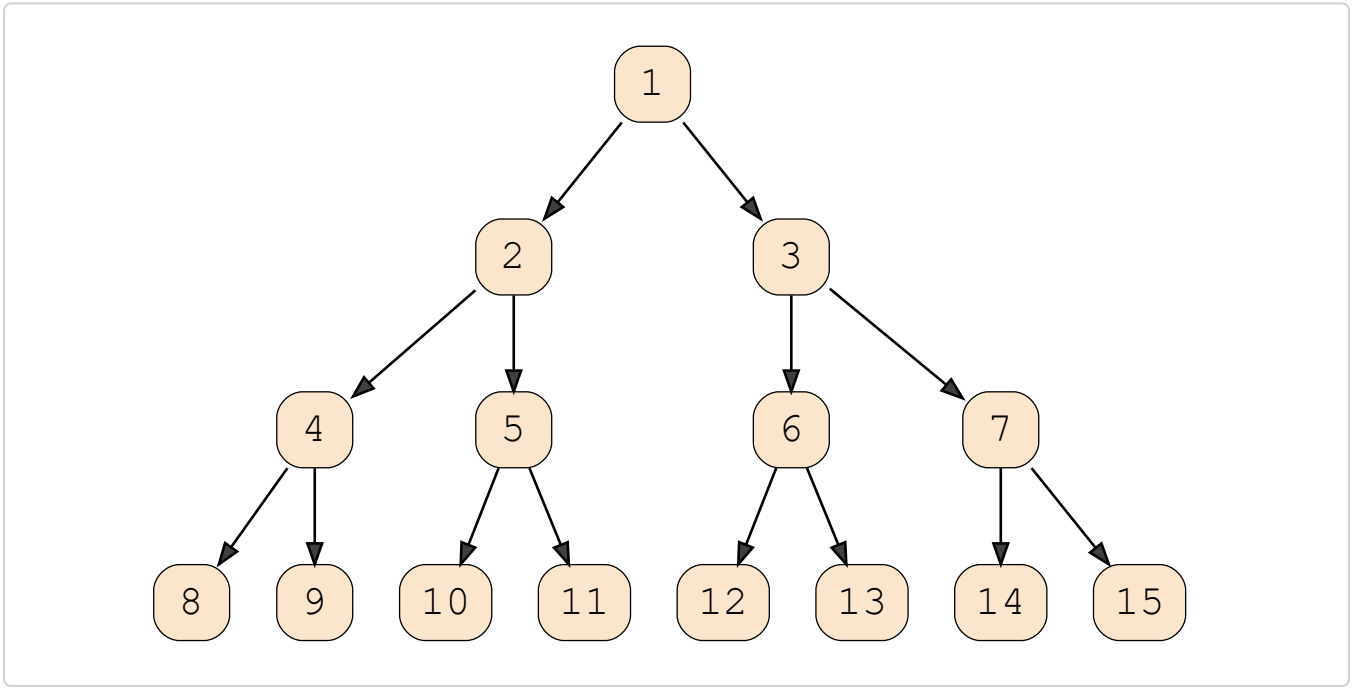
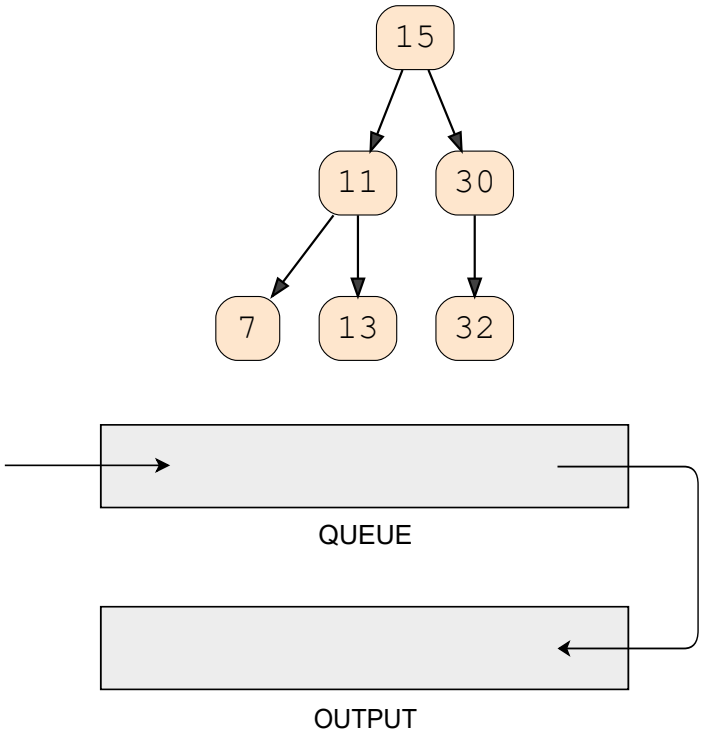
Breadth-first Traversal

In breadth-first traversal, we traverse the binary search tree from left to right. (Reading time: under 5 minutes)

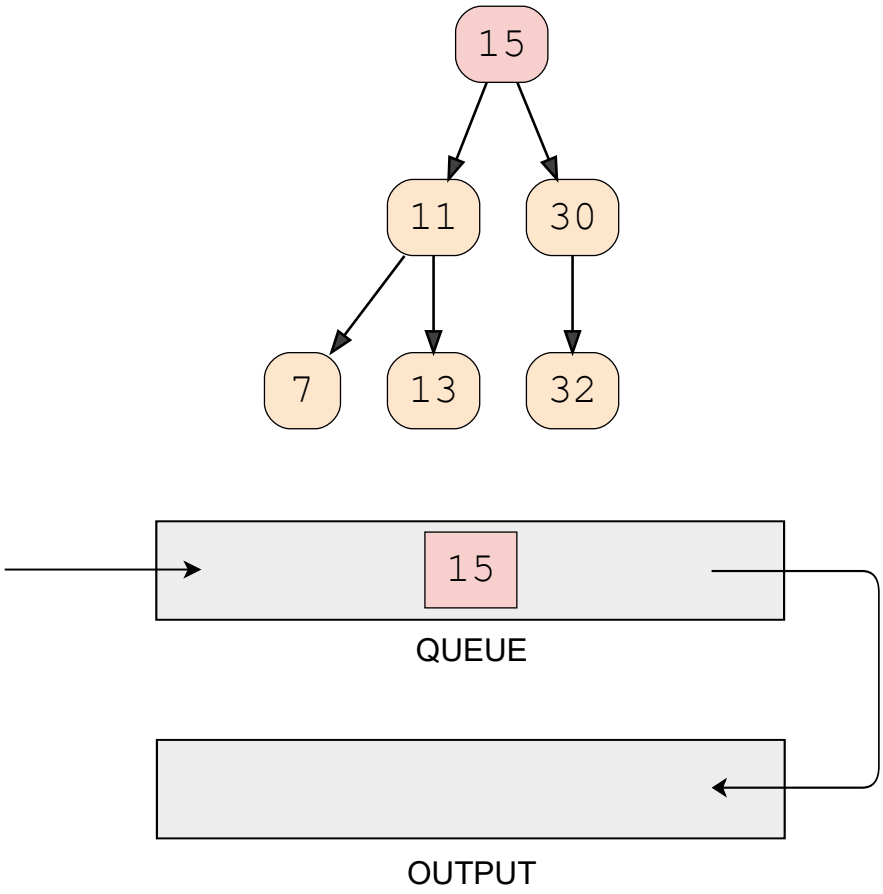


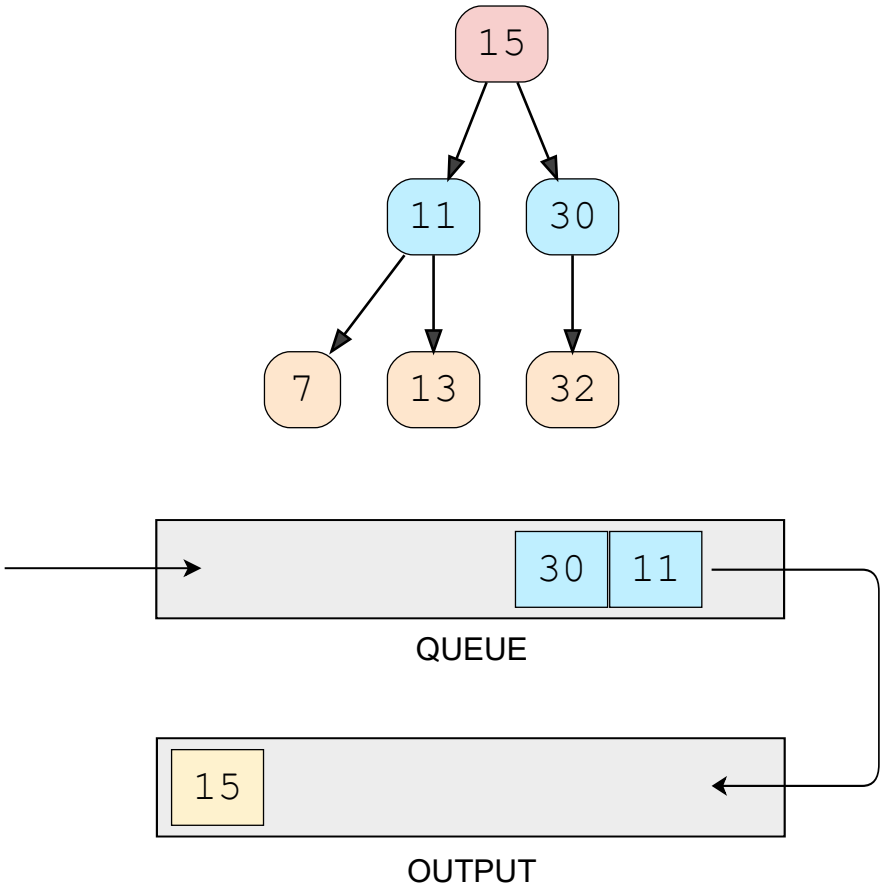
With breadth-first traversal, we first traverse through one **level of children nodes**, then through one level of grandchildren nodes, then through one level of grand-grandchildren nodes, etc. In the above tree, the numbers show the sequence in which the nodes will be traversed, not the data! With breadth-first search, we can't use a stack like we could with depth-first search: when we go from node 1 to node 2, node 2 doesn't have any connection to node 3! Instead of a stack, breadth-first traversal uses a **queue**. Queues allow us to store a reference to the nodes that we want to visit in the future, but haven't visited yet!



With breadth-first search, we want to go from 15, to 11, to 30, to 7, to 13, to 32 in this case. The currently visited node will be pushed to the queue. Once that node has been pushed to the output, its child nodes are also pushed to the queue!

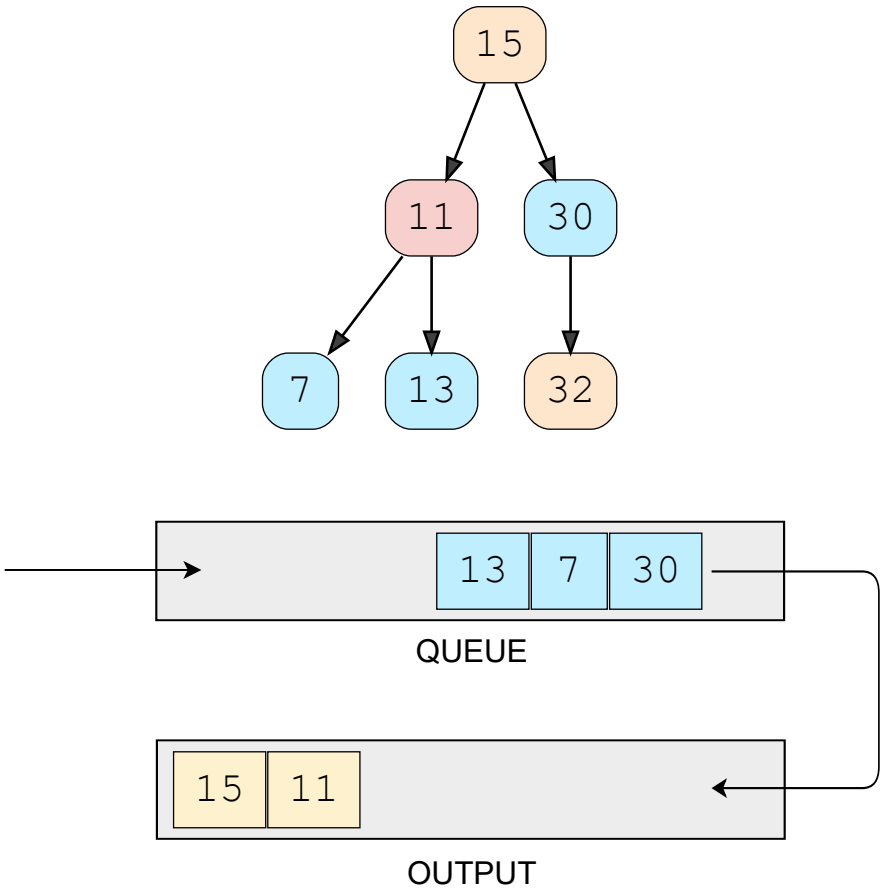
Now let’s consider the above example in a bit more detail.

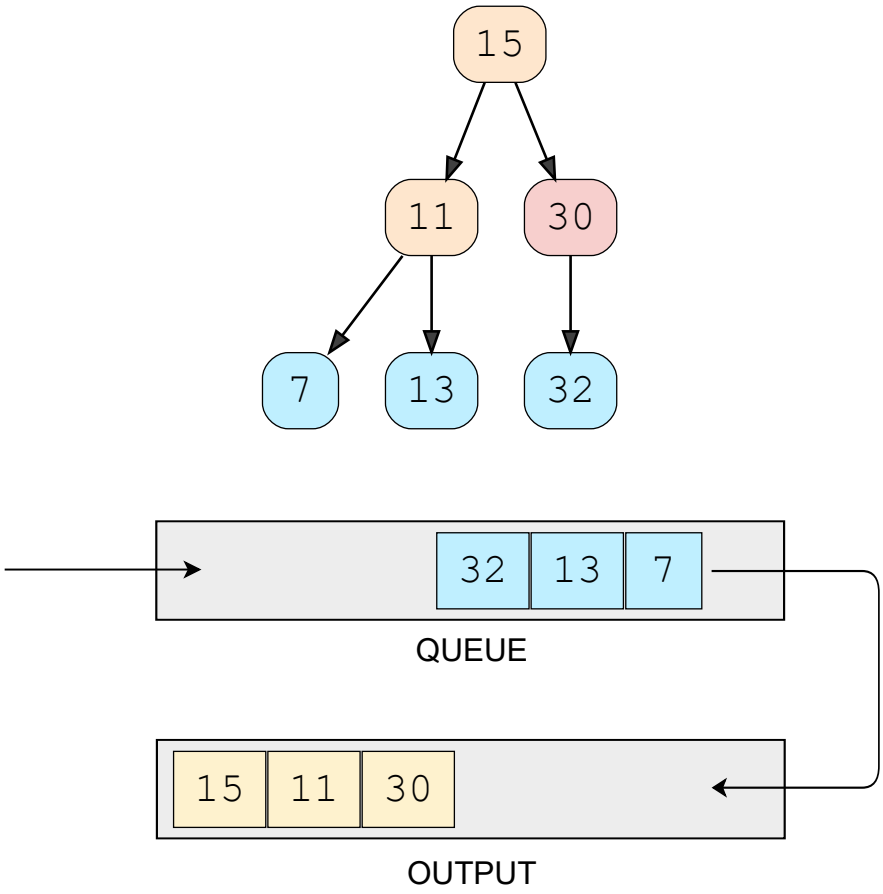

15
11
30
7
13
32
15
QUEUE
OUTPUT

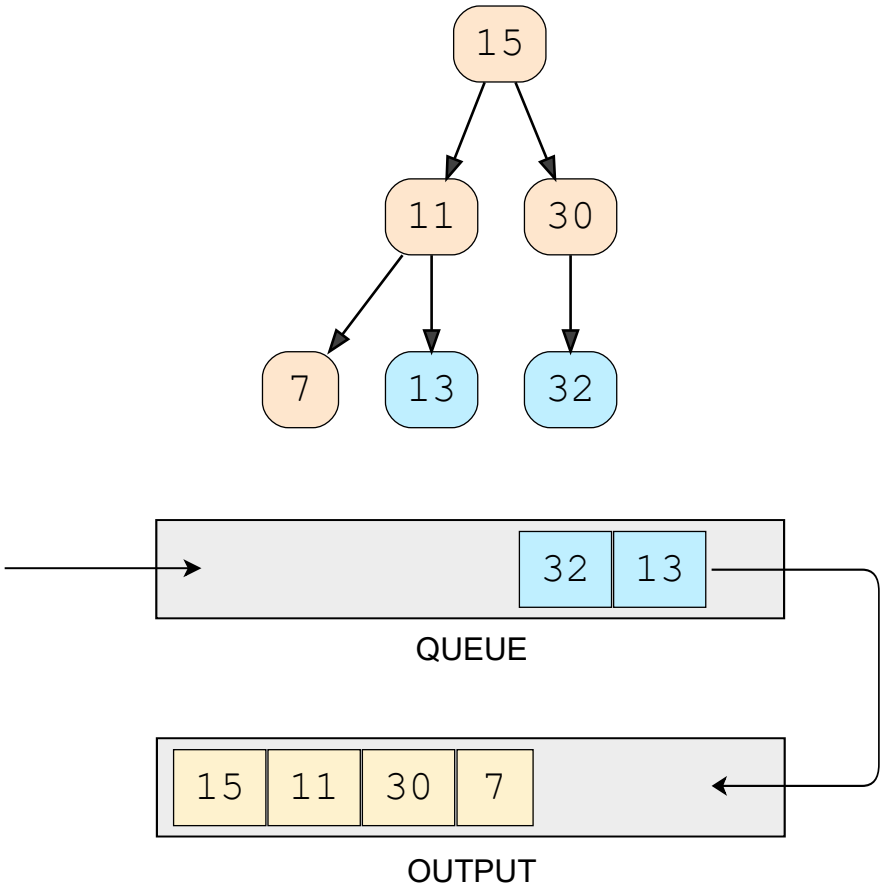

15
11
30
7
13
32
30
11
QUEUE
15
OUTPUT

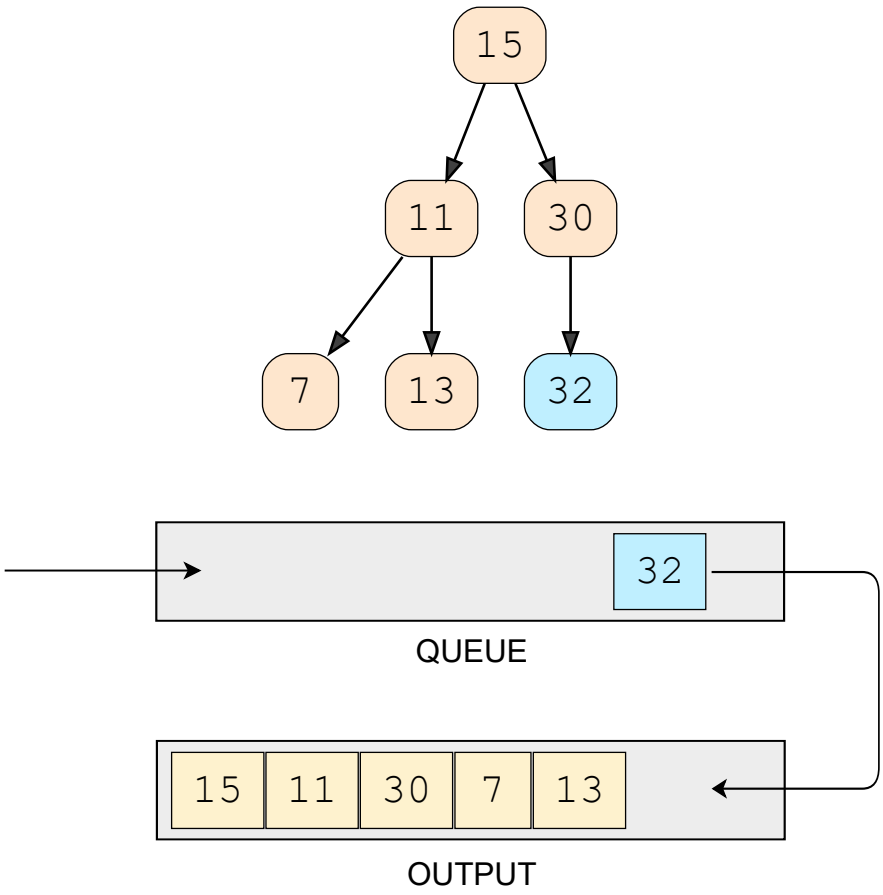
The node with the value **15** is first put in the queue, but immediately goes to the output. Now that it is in output, its child nodes get pushed to the queue.

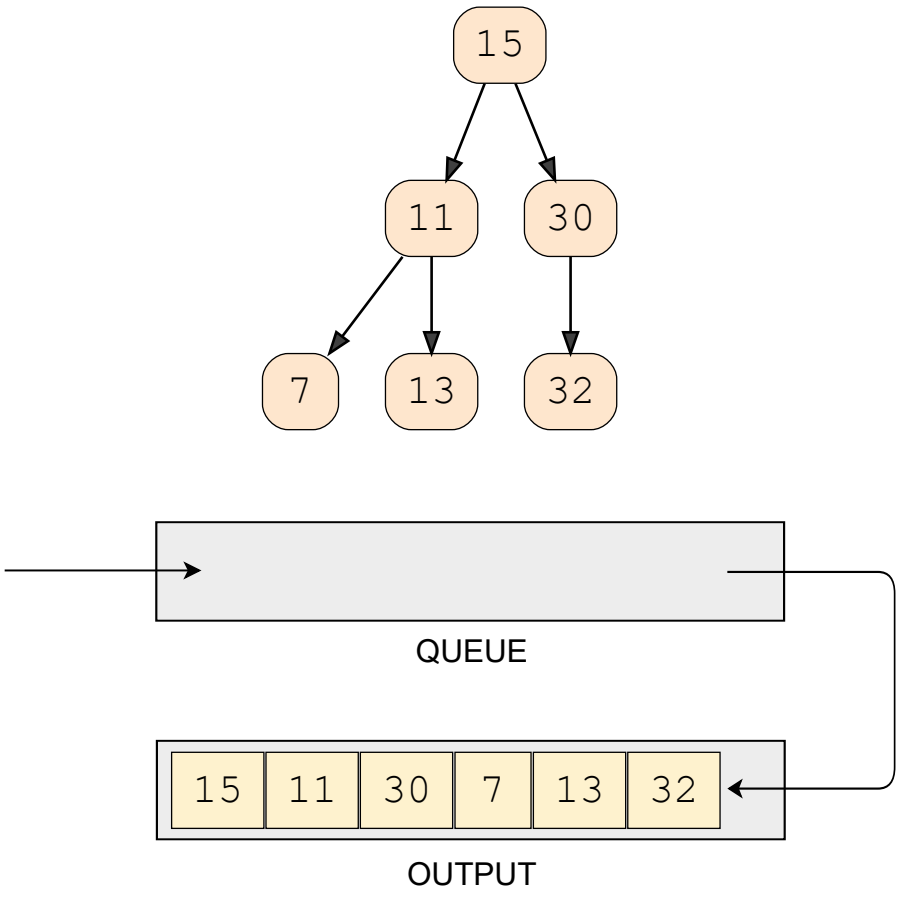
The first one in the queue gets added to the output sequence, and its child nodes get pushed to the queue. This keeps on going until we've reached the end of the tree!




15
11
30
7
13
32
32
13
7
QUEUE
15
11
30
OUTPUT


15
11
30
7
13
32
32
13
QUEUE
15
11
30
7
OUTPUT


15
11
30
7
13
32
32
QUEUE
15
11
30
7
13
OUTPUT


15
11
30
7
13
32
QUEUE
15
11
30
7
13
32
OUTPUT

Now let's write the code for breadth-first traversal of a tree:

```
traverseBFS() {
  if (!this.root) return;
  this.queue = [];
  this.queue.push(this.root);
  this.output = [];

  while (this.queue.length) {
    const node = this.queue.shift();
    if (node.left) {
       this.queue.push(node.left);
    }
    if (node.right) {
       this.queue.push(node.right);
    }
    this.output.push(node.data);
  }
  return this.output;
}
```

First, we need to check whether the tree has nodes at all. If that's not the case, we can't traverse anything, so we return from the function. Then, we initialize the queue. The first node that needs to be pushed to the queue, is the root. This means that the queue has a length, and the while condition returns true. We declare a variable called node, and set it equal to the last item in the queue, which now gets removed from the queue. Does this item have a node on the left? If yes, then that item gets pushed to the queue, the same logic gets repeated for its right node. The node that got removed from the queue gets returned!

```
15
11
30
7
13
32
```

A successful breadth-first traverse!

Finding Minimum and Maximum Value

Finding the minimum and maximum value in a binary search tree is rather easy, as you always know that the left subtree's values are lower than the current node, and the right subtree's values are higher than the current node.

```
getMin() {
  let node = this.root;
  while (node.left) {
      node = node.left;
  }
  return node.data;
}
```

Find the minimum

```
getMax() {
  let node = this.root;
  while (node.right) {
      node = node.right;
  }
  return node.data;
}
```

Find the maximum

From the above codes, it is evident that we keep on going left/right to find the absolute minimum/maximum value, and return that node's value!

In the next lesson, I will discuss the time complexity of the various binary search tree functions.