Binary Trees & Binary Search Trees

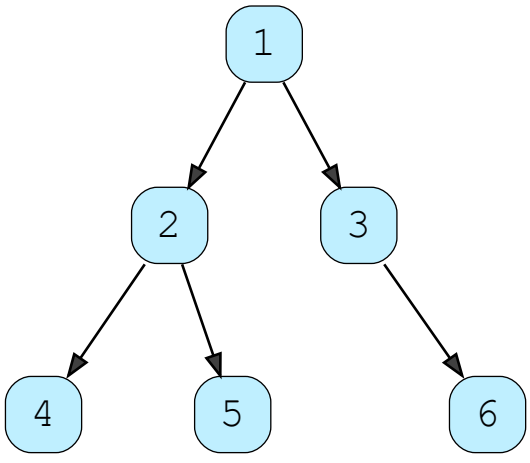
Introduction

A binary tree is a linked data structure where each node points to two child nodes (at most). The child nodes are called the ***left child*** and ***right child***.

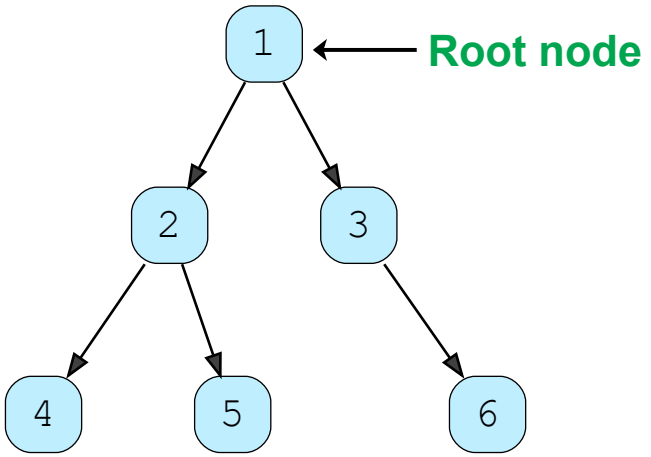
Unlike other linear data structures that we looked, it's a hierarchical data structure. Here are the properties of a binary tree.

1. Each node can point to two children at most.
2. The top most element in the tree is called ***root***.
3. Two Children are usually referred as ***Left Child*** and ***Right Child***.
4. The nodes which don't point to any children are called ***leaf nodes***.
5. Non-leaf nodes are called ***internal nodes***. Root is also an internal node if it's not a leaf node.

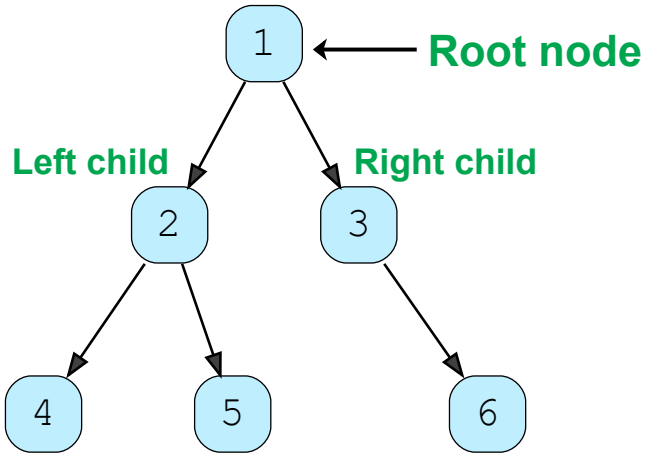
Let's step through a few visualizations to internalize these properties.



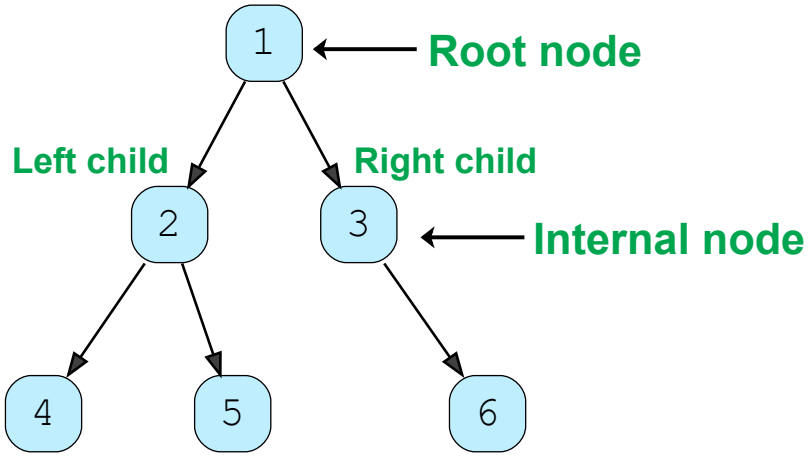
Here's our binary Tree



Node with value 1 is the root node



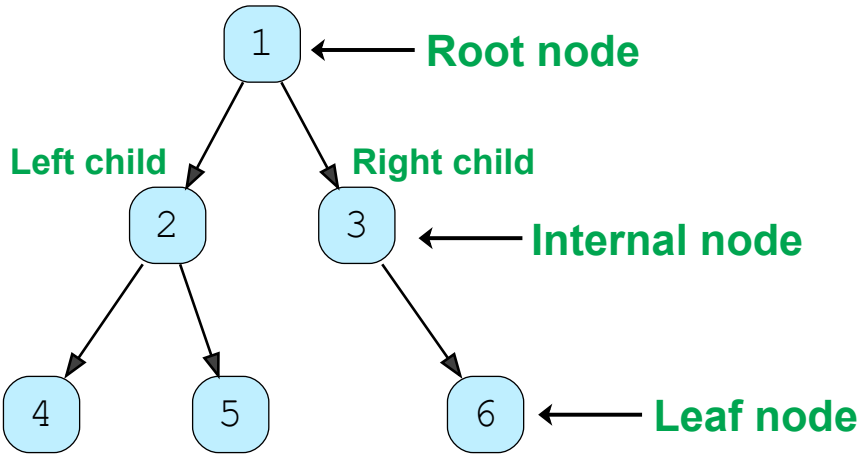
Nodes with value 2 & 3 are root's children



Nodes with value 2, 3 are internal nodes

Nodes with value 2, 3 are internal nodes

4 of 5



Nodes with value 4, 5, 6 are leaf nodes

5 of 5

Tree Basics Quiz

1 If a Tree has only 1 node, what type of node is it (select all that apply)

2 If a Tree node has one child, what type of node is it?

Check Answers

Height and Depth of the Tree

Now let's look at a couple more properties of the nodes and tree that are very useful for the discussion on trees.

The ***Depth of the node*** is the number of nodes on the path from the root to the node. Root has depth 0.

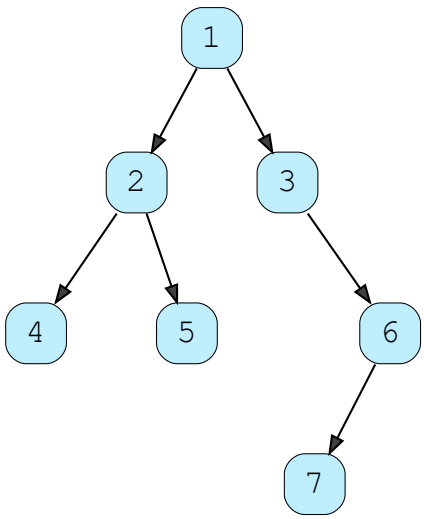
The ***Height* of the node** is the number of nodes on the path from the node to the root. All leaf nodes have height 0.

You can see that for depth, we begin counting from the root, down towards to the node, starting at 0. For height, we start counting from the node, up towards the root, again starting at 0. Some people start counting depth and height from 1 instead of 0 and it's just a matter of preference.

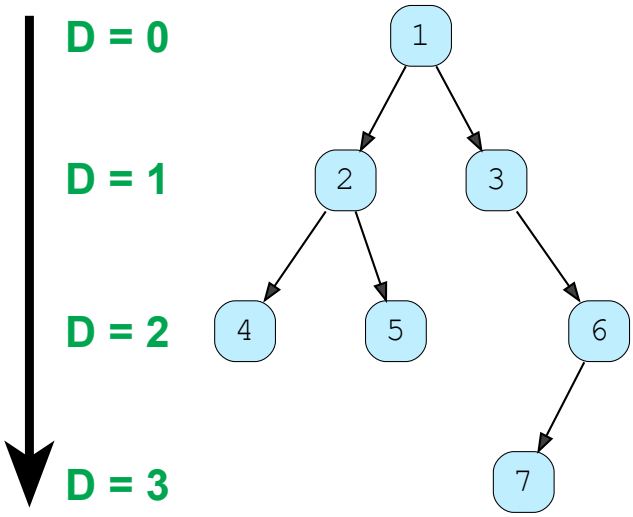
The ***Height of the tree*** is the maximum height of any node in the tree.

The ***Depth of the tree*** is the maximum depth of any node in the tree.

Hence, depth of the tree is always equal to the height of the tree. Let's step through a visualization to look at the height and depth properties.

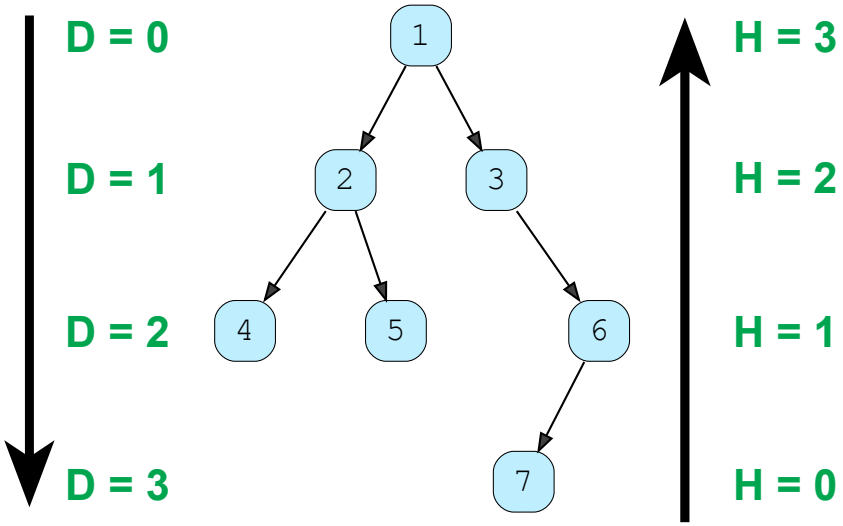


1 of 3



Depth of nodes.
Depth of Tree = Max Depth = 3

2 of 3



Height of nodes.
Height of Tree = Max Height = 3

3 of 3

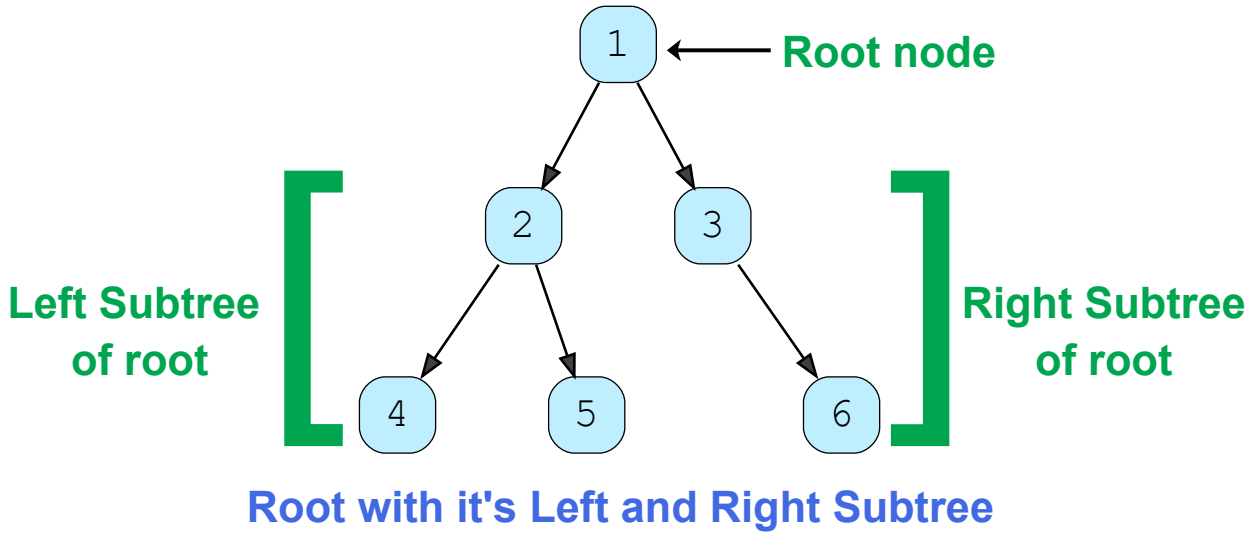
Key of the node and Subtree

In a binary tree, the node might contain a lot of data with a key. Our examples are a little simplistic as we are only working with a single integer which acts

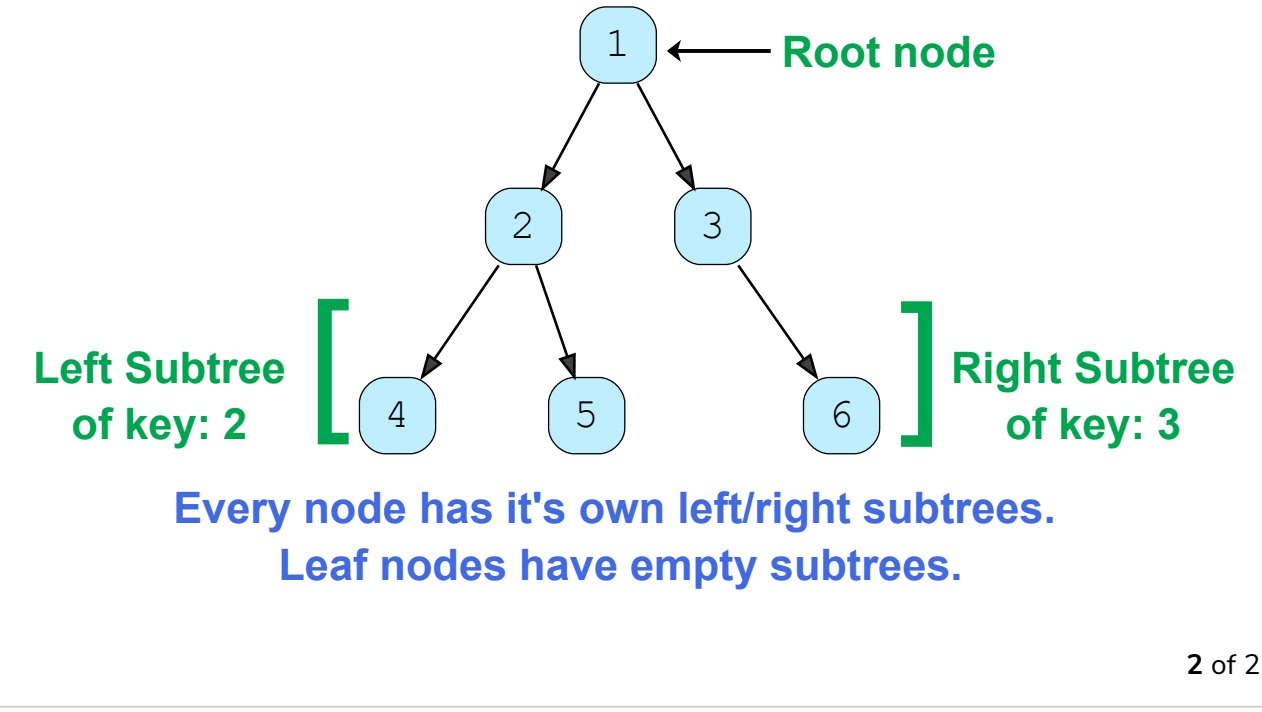
both as a key and value. However, in many practical situations, a node contains several fields of data with a key. In our examples below, we will continue to use a single value as data and key, but it's a good thing to remember that, in practice, a node might contain more data.

A subtree of the node is the tree formed by its left and right children. Each node in a binary tree has a left subtree and a right subtree.

Let's step through the following visualization to understand the subtrees.



1 of 2



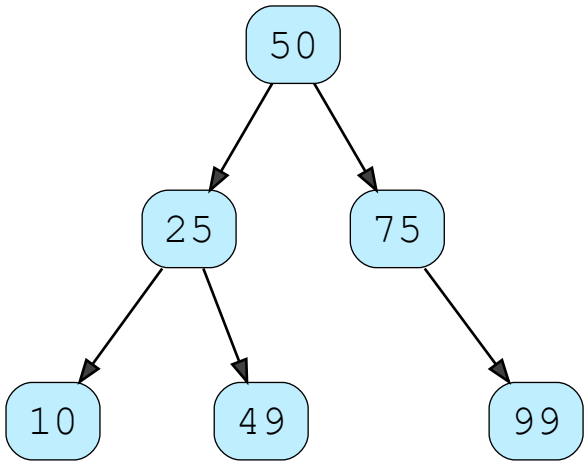
2 of 2

Binary Search Tree

The most common binary tree is called a Binary Search Tree. Why are we discussing a specialized form of Binary Tree without even getting to the code? Because the structure and layout of data in a Binary Tree would make more sense to you if we discuss insertion/deletion/lookup in a Binary Search Tree.

So what's a Binary Search Tree (BST hereafter)? The name already gives a decent hint. It's a binary tree that helps in searching the data in the tree. We'll shortly look at how a BST facilitates efficient lookups. To understand this, first let's understand the properties of a BST.

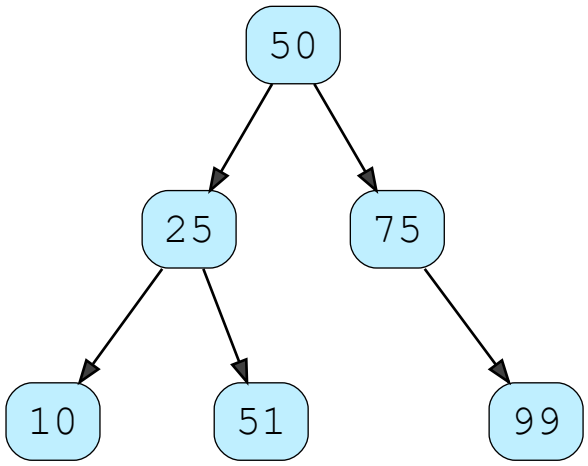
A binary tree is a BST if the key of the node is greater than ***all*** the nodes in its left subtree and is smaller than ***all*** the nodes in its right subtree. Let's look at a binary search tree.



A Binary Search Tree

Let's take a closer look at the above BST. Root has the key *50*. Look at all the nodes in its left subtree. They are all less than *50*. Now, look at the right subtree of the root. All the keys in the right subtree are greater than *50*. Hence, it's a BST.

Let's look at another example and see whether it's a BST or not.



Is it a BST?

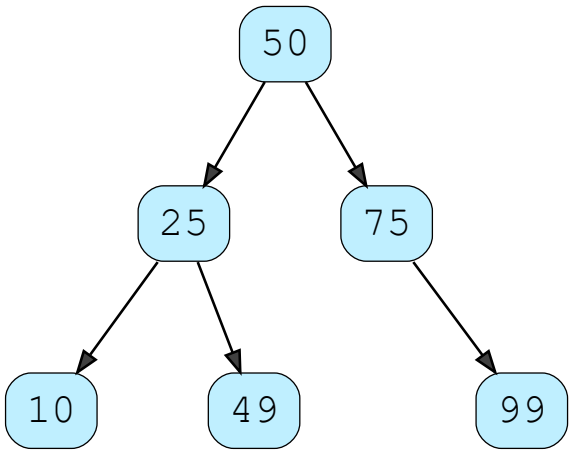
The tree above is *not* a BST. Why? Because of the leaf node containing *51*. It's in the left subtree of *50* which breaks our invariant.

Search in a Binary Search Tree

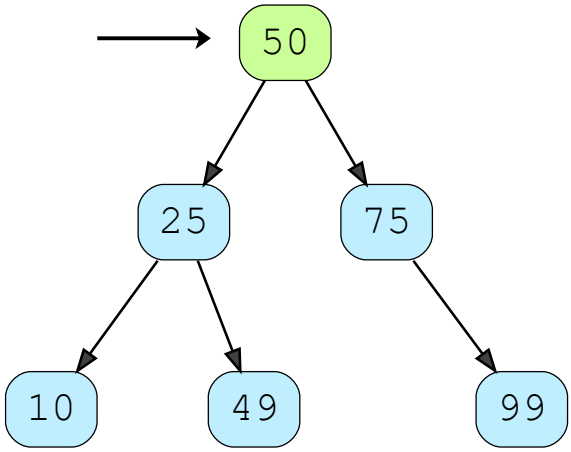
Now that we know what a BST looks like, let's see how can it help with lookups. It's pretty simple actually. Suppose you are looking for key X. Here's what you do at each step starting at the root.

1. Compare current node's key with X. If it's equal, we've found the key. All done.
2. If X is less than node's key, we start looking at node's left subtree. It's because we know that right subtree cannot contain anything greater than X.
3. If X is greater than node's key, we start looking in the right subtree.
4. We repeat this process until we find the key or we reach the leaf node. If we reach the leaf node and haven't found the key as yet, we return not found.

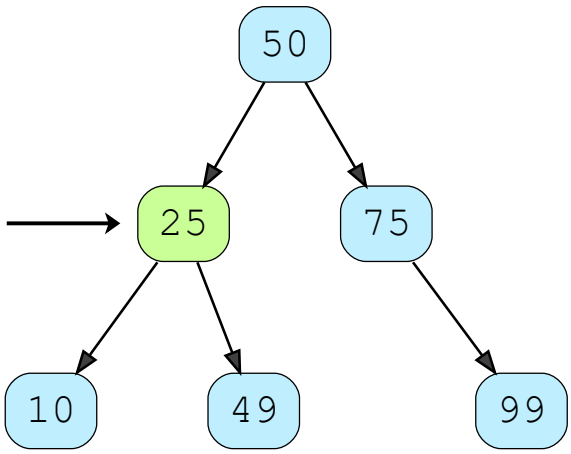
Let's look at search in a BST in action.



Let's search for key 49



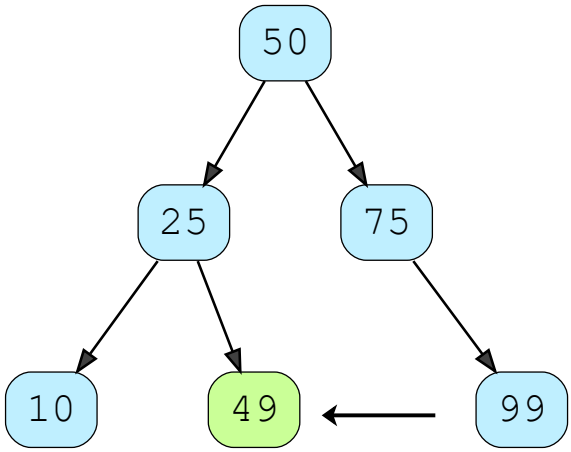
We start the search at root.
Root is 50 (> 49) so we go the left subtree.



25 is less than 49.

We go to the right subtree.

3 of 4



We found node containing 49

4 of 4

Notice that tree has six elements but we only needed to do three comparisons before we found our desired node. If you look closely, you can see that at each step, we eliminate almost half of the tree.

One thing to remember here is that this is the best case for a BST. There are many cases where your BST might not give you an optimal performance, but they are beyond the scope of this introduction.

We have seen a lot of theory regarding Binary trees and Binary Search trees. Let's look at some code.

Defining the Tree Node

We'll define a Node class with a data field and two fields to point to the left and the right child.

```
function Node(data) {
  this.data = data;
  this.left = null;
  this.right = null;
}
```

Defining Binary Search Tree

The following code defines how the tree would be represented in Javascript.

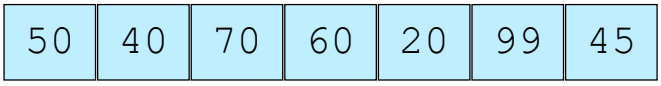
```
function BST() {
  this._root = null;
}
```

Inserting into BST

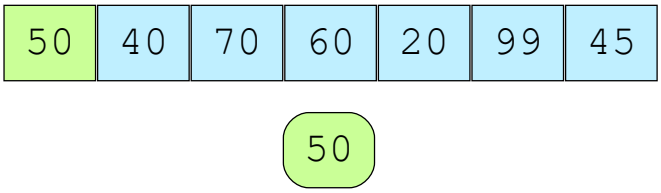
In a BST, we need to find the correct location for the Node to be inserted. It depends on the value of the key. Assume, we want to insert a node with key K. Starting at the root, here's a quick description of how we are going to locate the correct location.

1. Compare current node's key with K.
2. If K is less than the current node,
    1. If left child of current node is Null, we insert K as the left child of current node and return.
    2. If the left child is not Null, the left child becomes the new current node, and we repeat the process from step 1.
3. If K is greater than the current node,
    1. If right child of current node is Null, we insert K as the right child of the current node and return.
    2. If the right child is not Null, the right child becomes the new current node, and we repeat the process from step 1.

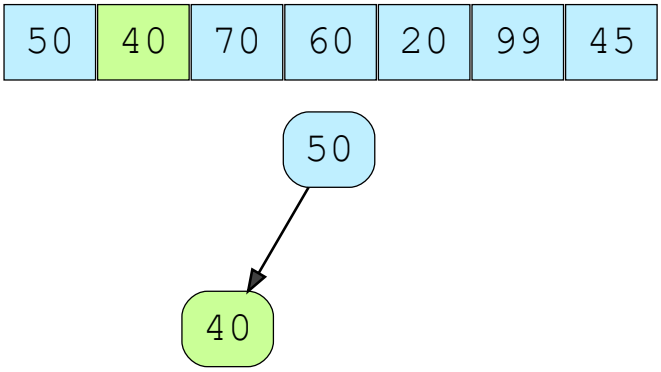
Using the steps described above, we insert K at the correct location so that BST is quickly searchable later on. Let's step through a quick visualization to see insertion in action.



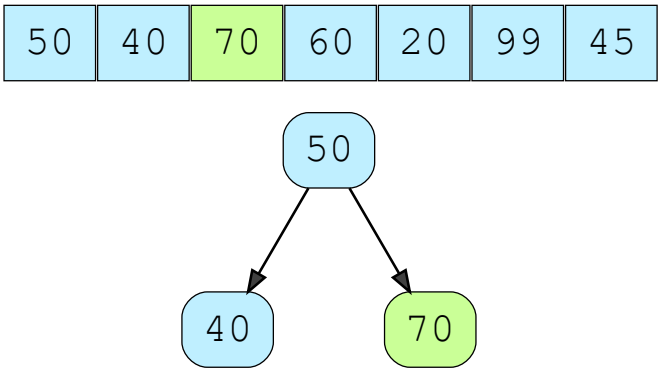
We'll insert the list into BST



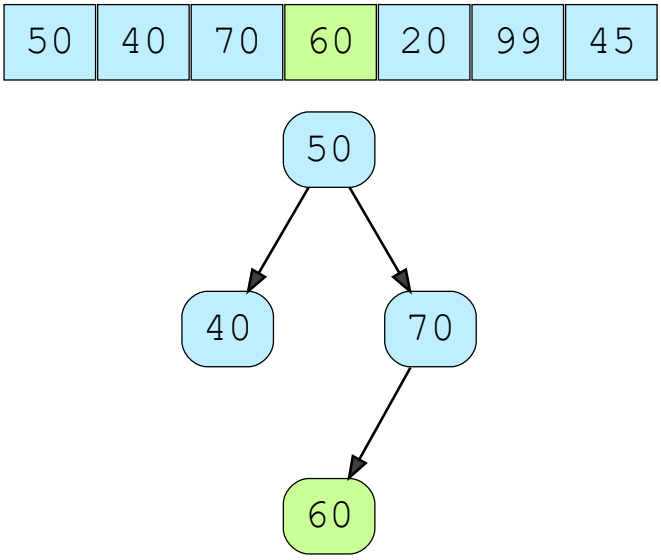
Insert 50 as root



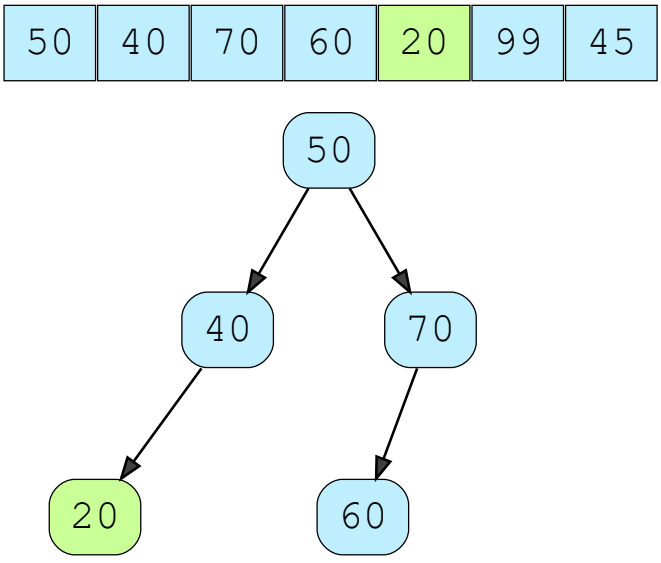
Insert 40



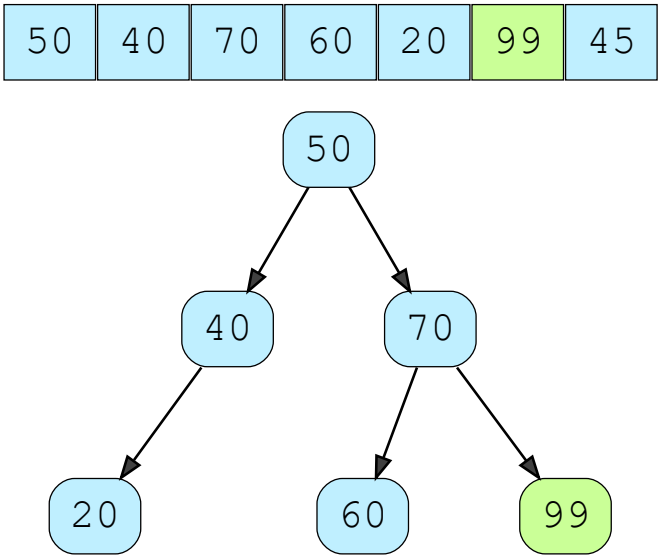
Insert 70



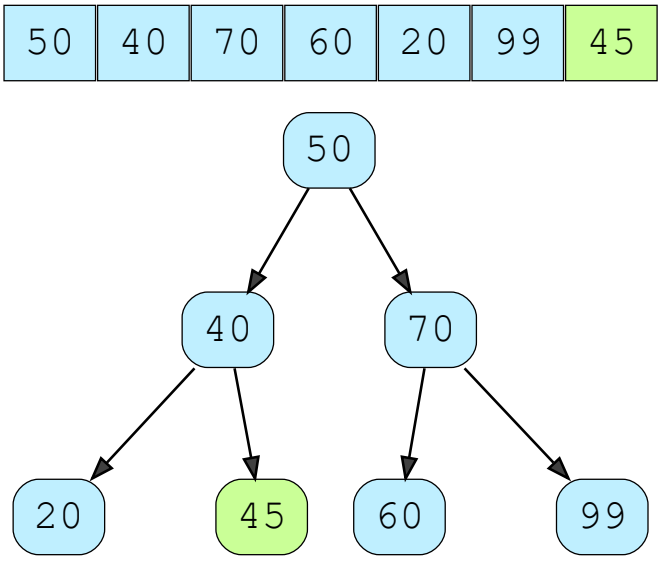
Insert 60



Insert 20



Insert 99



Insert 45.
... and we're done.

You might be wondering what do we do in case of a duplicate. For now, we simply ignore it. There are some binary trees in which we can allow

duplicates but if you are searching/sorting on a Key, then duplicated keys don't make sense in many other scenarios.

Let's look at the insertion code for a BST.

```
BST.prototype.insert = function(data) {
  var node = new Node(data);

  // If it's the first node
  if (this._root === null) {
    this._root = node;
    return;
  }

  var current = this._root;

  while (current) {
    if (data < current.data) {
      if (current.left === null) {
        current.left = node;
        return;
      }
      current = current.left;
    } else if (data > current.data) {
      if (current.right === null) {
        current.right = node;
        return;
      }
      current = current.right;
    } else {
      // Duplicates are not supported
      return;
    }
  }
};
```

Implementing Search in a BST

We've already described how lookup works in Binary Search Tree. Let's look at the code for the lookup method as well. It returns True if the key is found and returns false otherwise.

```
BST.prototype.contains = function(data) {
  var current = this._root;

  while (current) {
    if (data === current.data) {
          return true;
    }

    if (data < current.data) {
      current = current.left;
    } else {
      current = current.right;
```

```
    }
  }

  return false;
};
```

BST Insert and Search in action

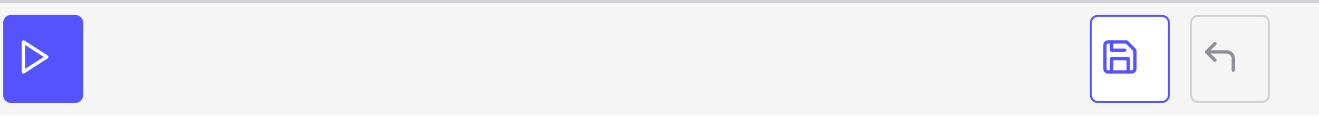
> Run the following code to see BST insert and search.

JavaScript

HTML

CSS (SCSS)

```
var bst = new BST();
console.log('Insert into BST: 5');
bst.insert(5);

console.log('BST contains 5? Returns ' + bst.contains(5));
console.log('BST contains 6? Returns ' + bst.contains(6));
```



Console

Clear

```
Insert into BST: 5
BST contains 5? Returns true
BST contains 6? Returns false
```

Tree traversals

We've covered search in a BST. When you want to traverse all the elements in a tree, there are a few well-known techniques to do that. These methods define the ***order in which nodes are visited***, and each one of these traversal orders helps us solve different problems. Let's look at the three most common traversal techniques.

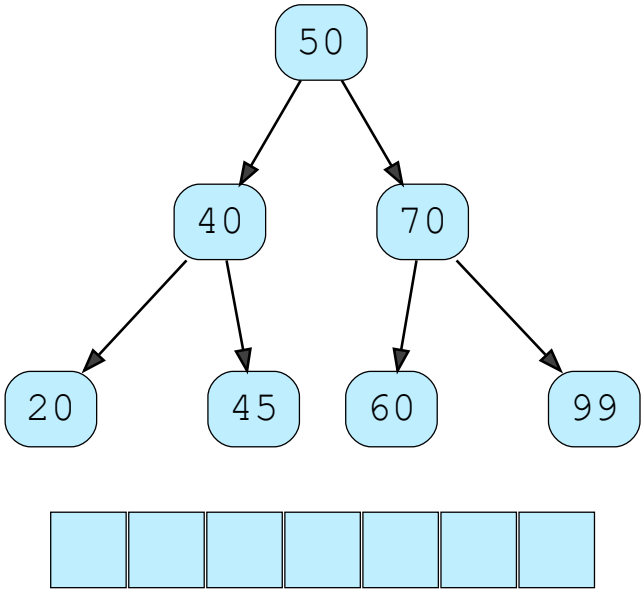
Pre-Order Traversal

In Pre-Order traversal, we traverse in the following order:

1. First visit the node itself.
2. Then visit the left subtree of the node.
3. Then visit the right subtree of the node.

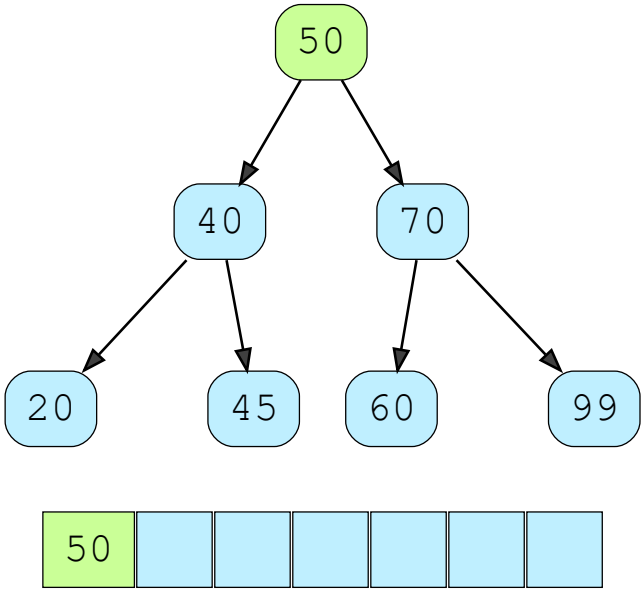
Let's step through a visualization to understand it better (Highly recommended to go to the last slide to see the complete order).

Let's traverse in Preorder

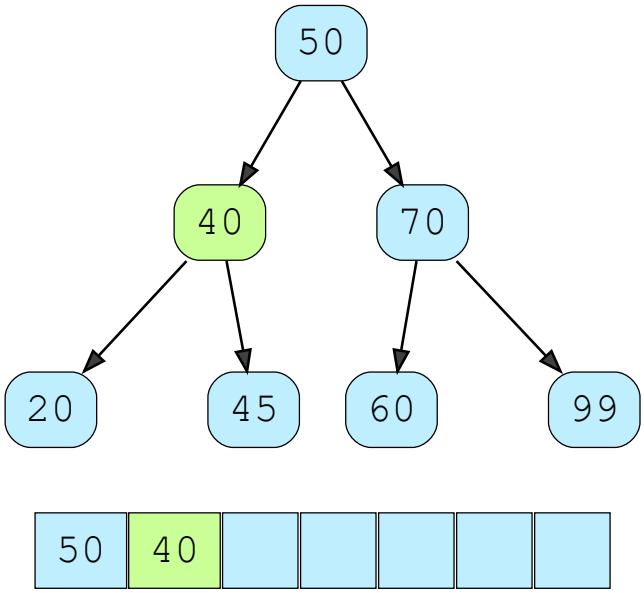


1 of 9

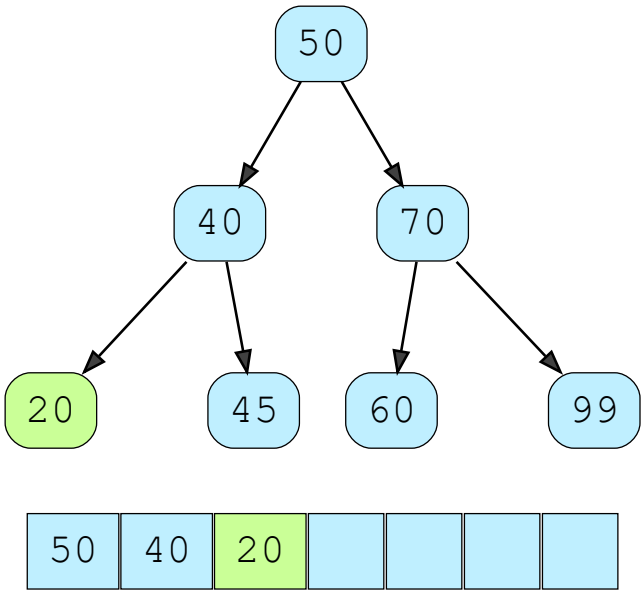
Start at Root 50. We visit it.



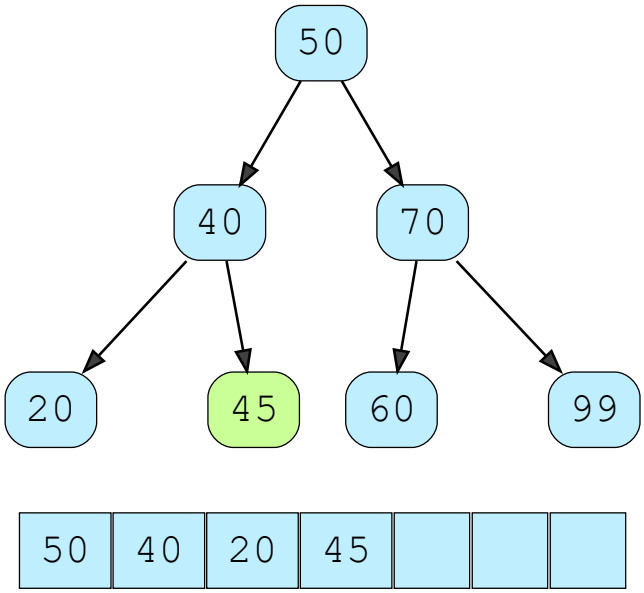
Traverse to left of 50. Visit it.



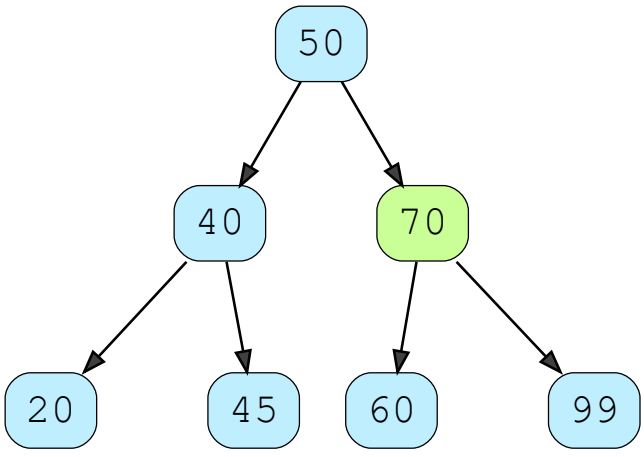
Traverse to left of 40. Visit it.

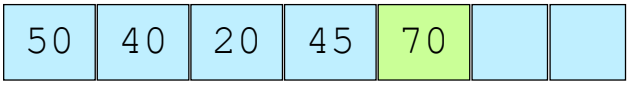


20 is leaf node. Visit right of 40

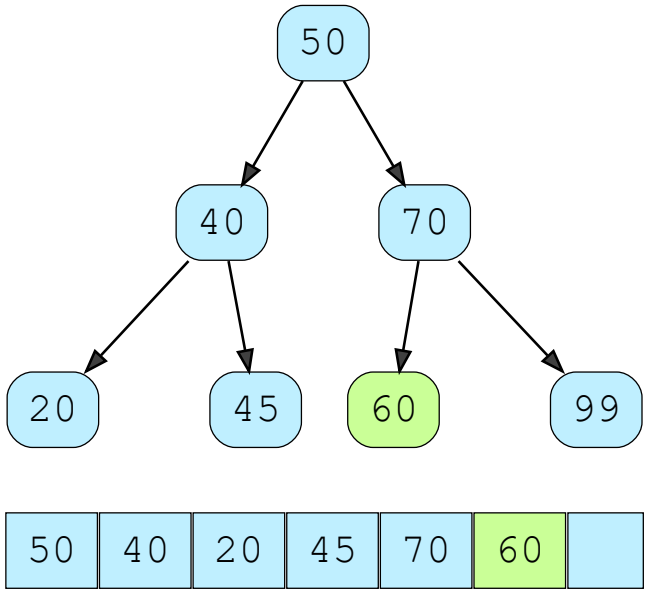


Done with left side.
Visit 50's right child 70.

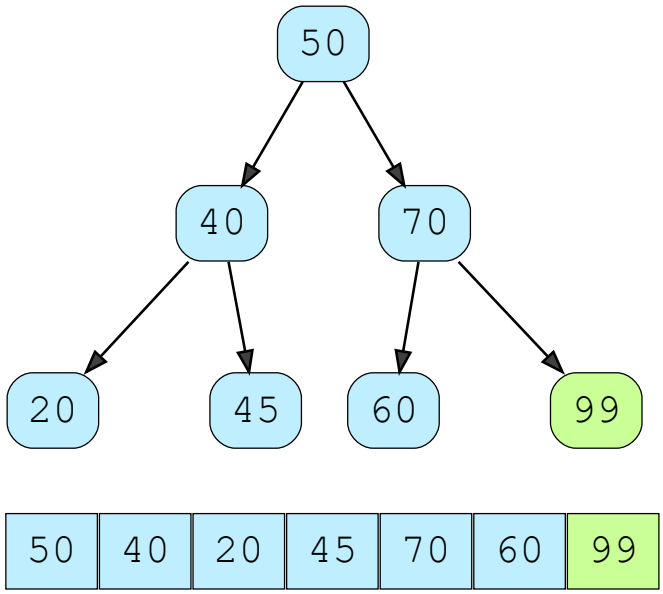


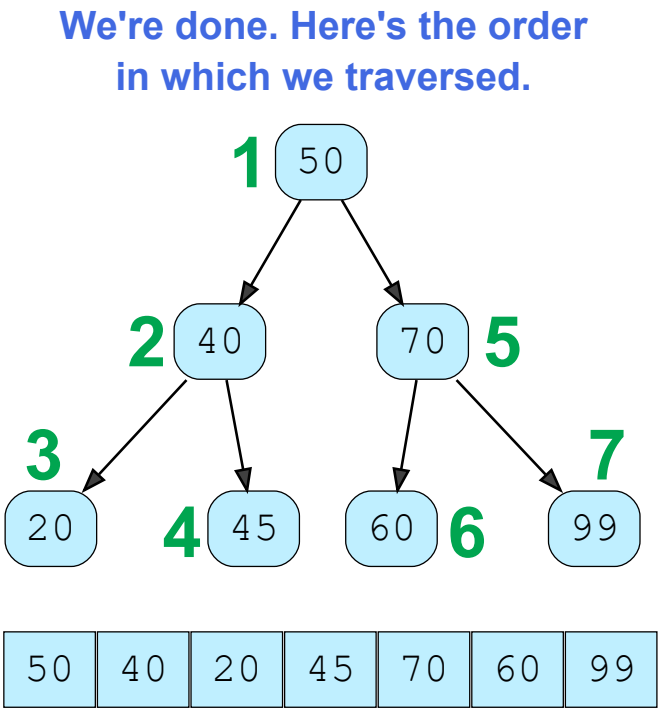


Visit 60 as it's left child of 70



Visit 70's right Child 99.





Tree traversals are implemented using recursion. Here's the code that fills the array in pre-order traversal.

```
// Returns keys in pre-order traversal
BST.prototype.preOrder = function() {
  var output = [];

  function preOrderImpl(node) {
    if (node === null) {
      return;
    }

    // Visit the node itself.
    output.push(node.data);

    // Visit left subtree
    preOrderImpl(node.left);

    // Visit the right subtree
    preOrderImpl(node.right);
  }

  // Call the internal function
  // with Root as the starting point.
  preOrderImpl(this._root);

  return output;
```

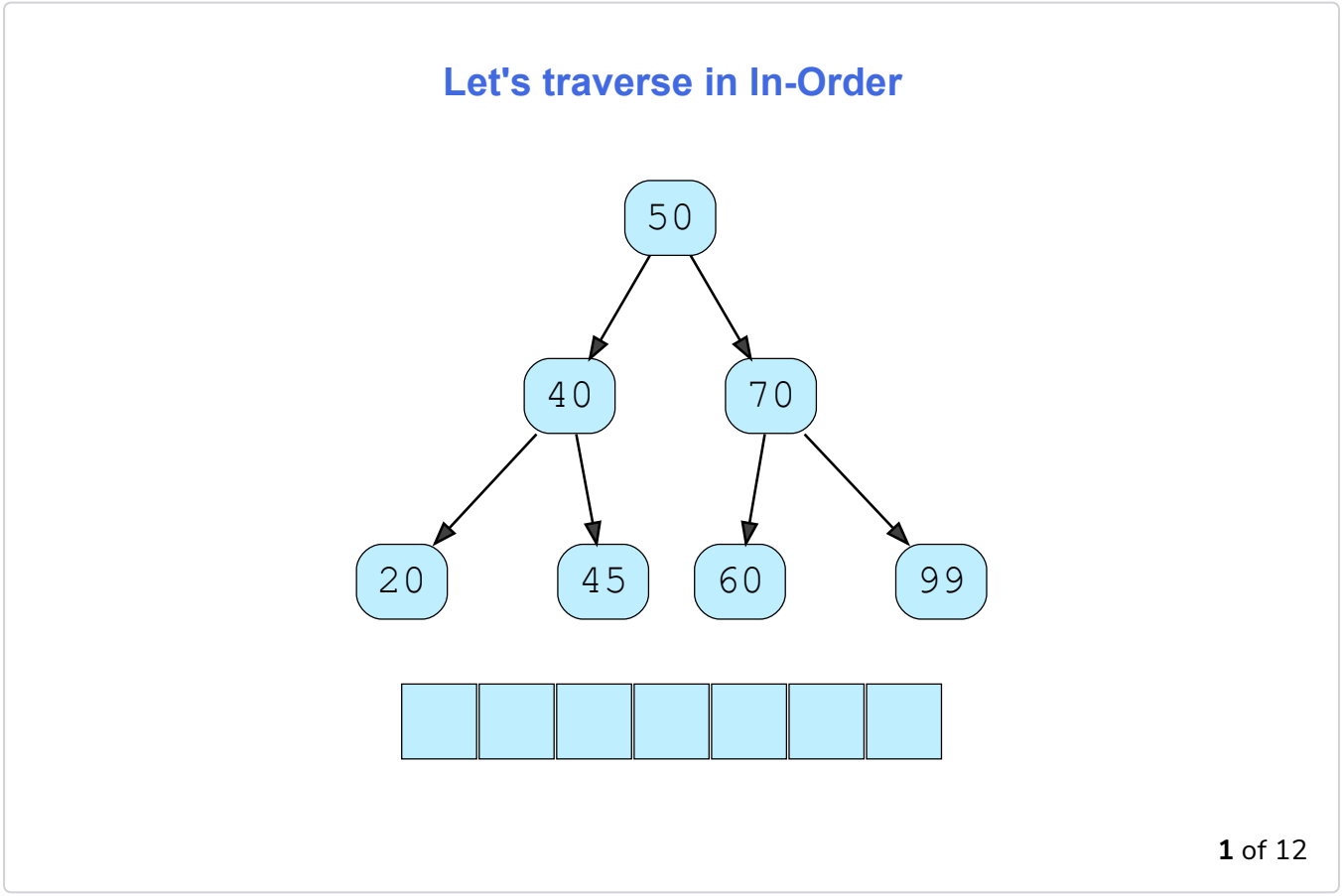
```
}
```

In-Order Traversal

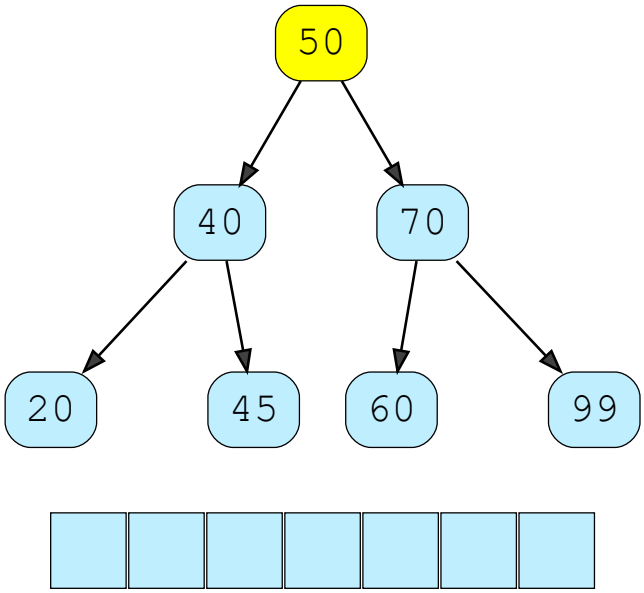
In in-Order traversal, we traverse in the following order:

1. First visit the left subtree of the node.
2. Then visit the node itself.
3. Then visit the right subtree of the node.

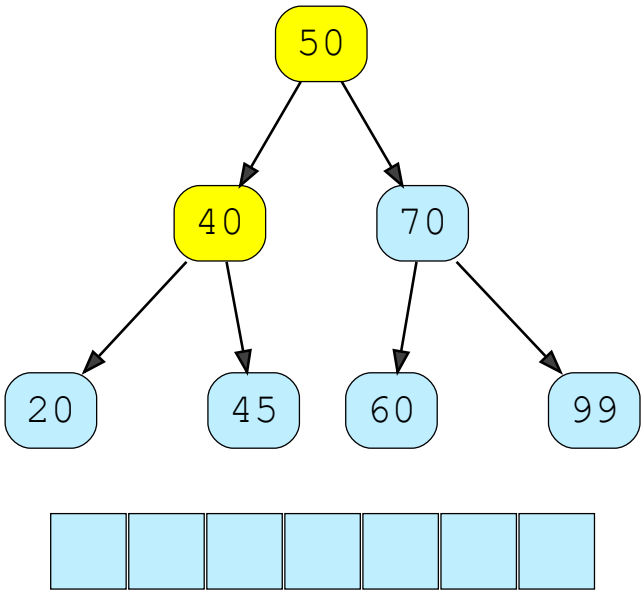
Let's step through a visualization to understand it better (Highly recommended to go to the last slide to see the complete order).



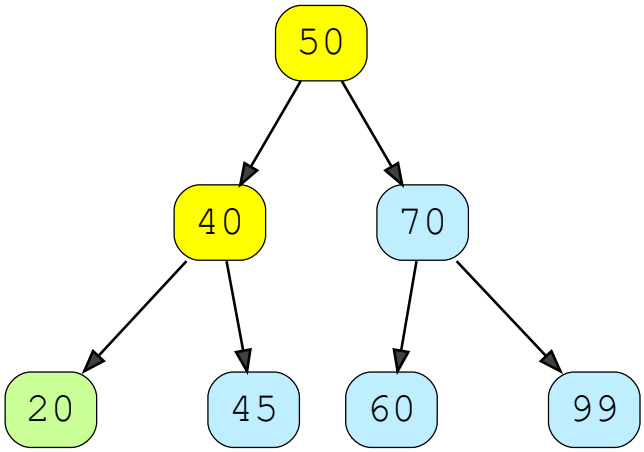
Start at Root 50. It has left subtree
Move to left child.



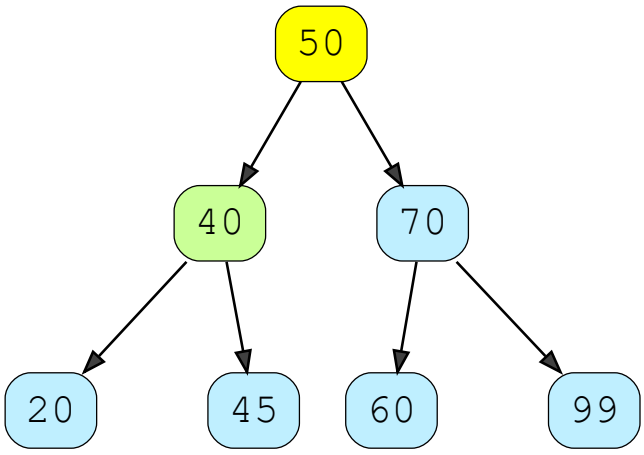
40 still has a left subtree.
Move to left child.



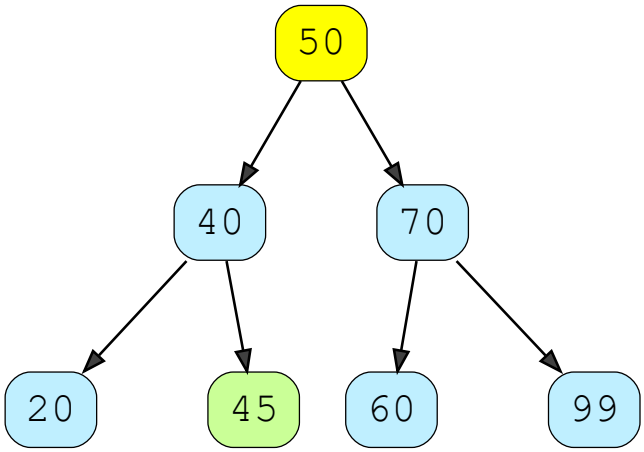
20 is a leaf node. Visit it.



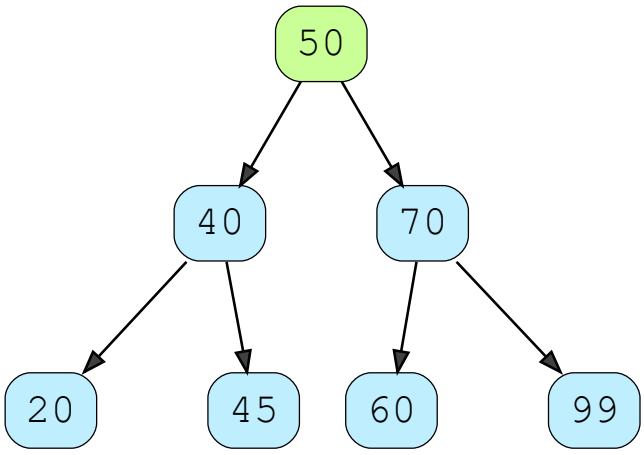
Done with 40's left subtree
Visit 40 itself.



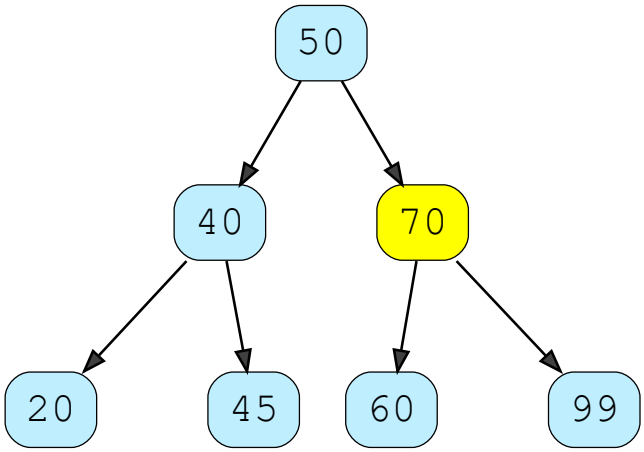
Start visiting 40's right subtree.
Visit 45.

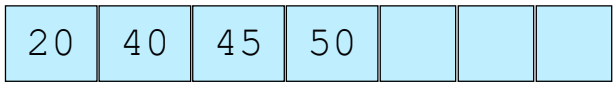


Done with visiting 50's left subtree.
Visit 50 itself.

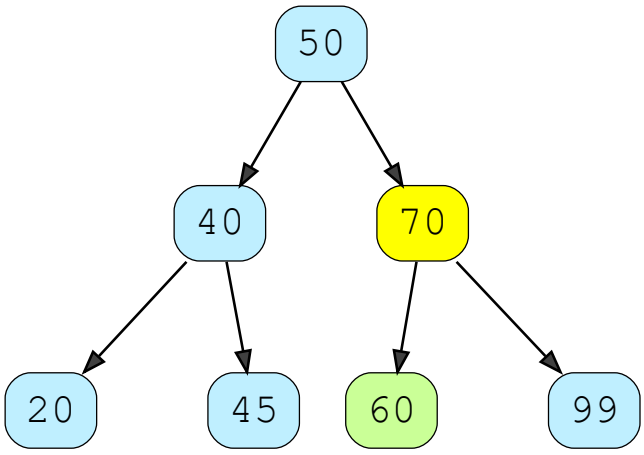


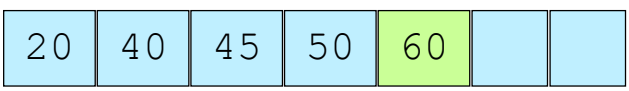
Start traversing 50's right subtree.
70 has left subtree so we go there.



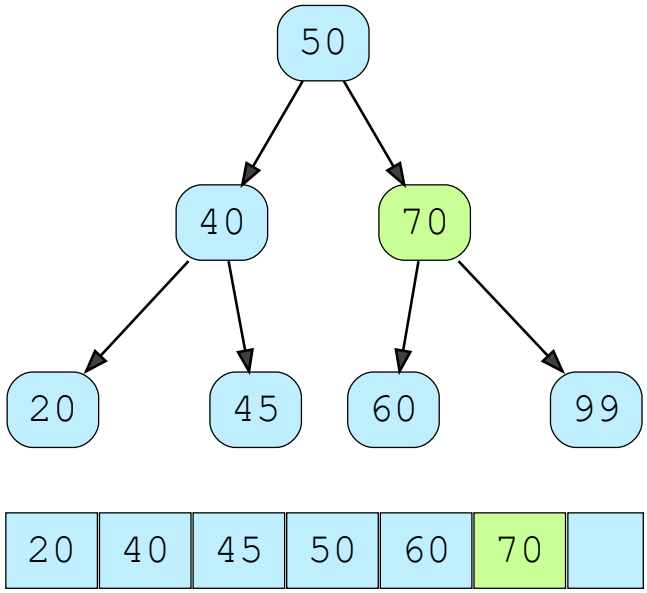


60 has no subtree.
Vist 60.

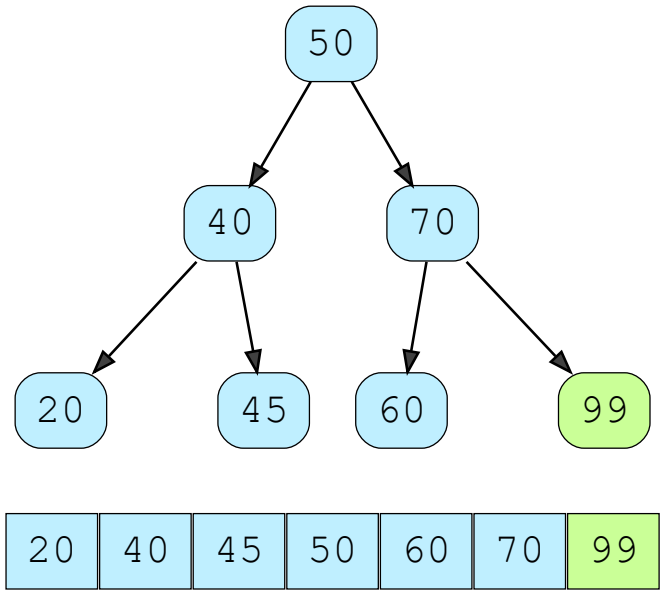


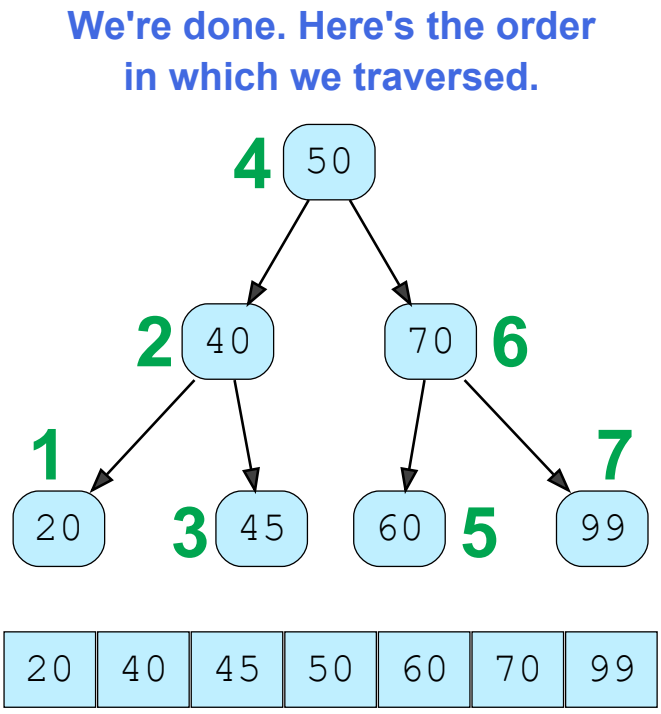


Done with 70's left subtree.
Visit 70 itself.



Visit 70's right subtree.
99 is leaf node. Visit it.





12 of 12

If you haven't noticed as yet - there is a nice *property* of In-Order traversal. It traverses the BST in ascending order. Now you know what traversal to use if you want to print/get all the nodes of tree in **sorted order.**

Let's look at the code for In-Order traversal.

```
// Returns Keys in the InOrder traversal
BST.prototype.inOrder = function() {
  var output = [];

  function inOrderImpl(node) {
    if (node === null) {
      return;
    }

    // Visit left subtree
    inOrderImpl(node.left);

    // Visit the node itself.
    output.push(node.data);

    // Visit the right subtree
    inOrderImpl(node.right);
  }
```

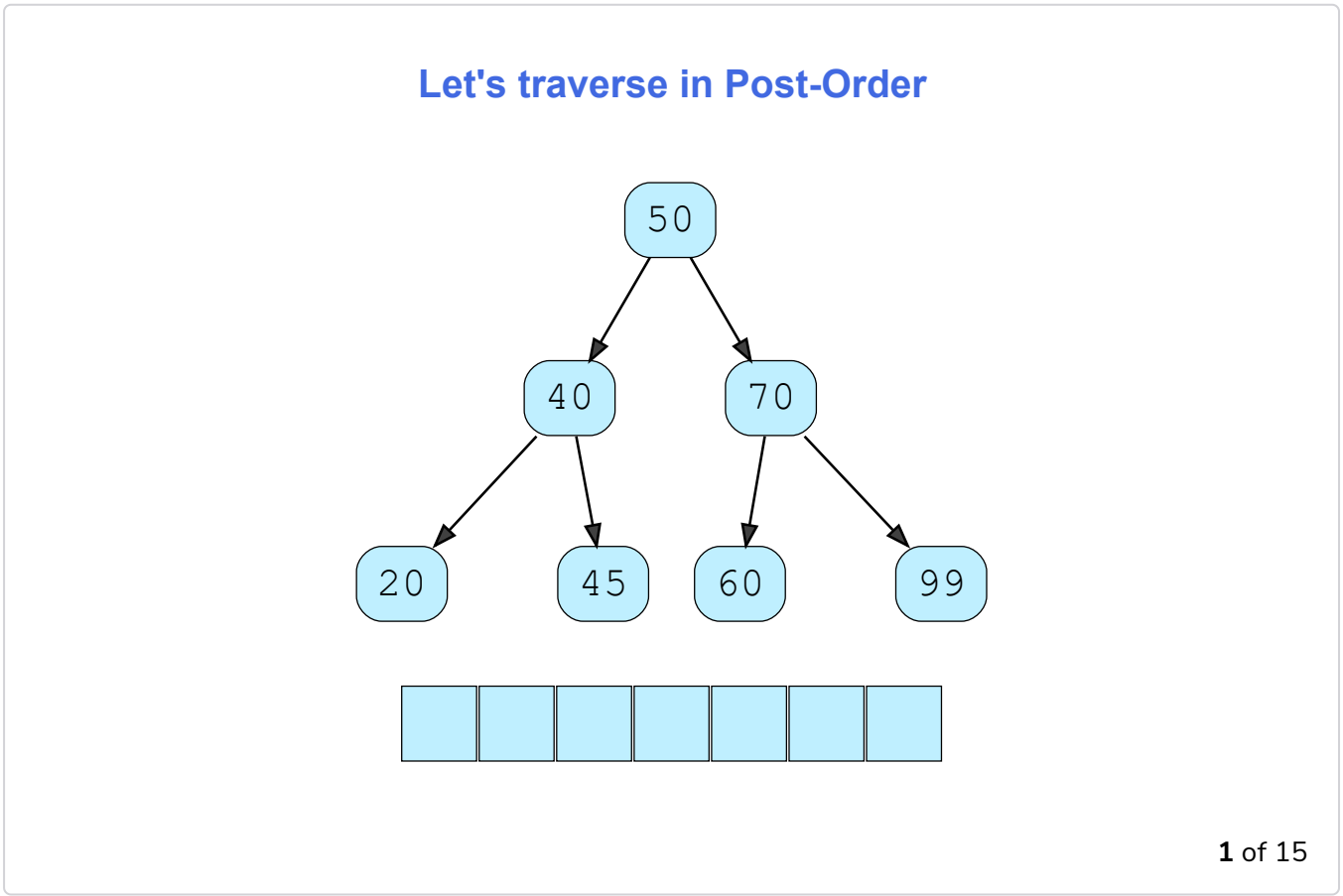
```
  // Call the internal function
  // with Root as the starting point.
  inOrderImpl(this._root);


  return output;
}
```

Post-Order Traversal

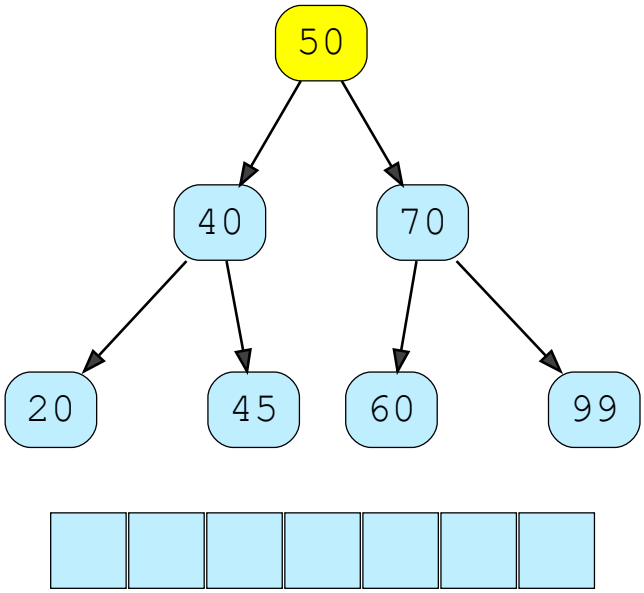
In Post-Order traversal, we traverse in the following order:

1. First visit the left subtree of the node.
2. Then visit the right subtree of the node.
3. Then visit the node itself.

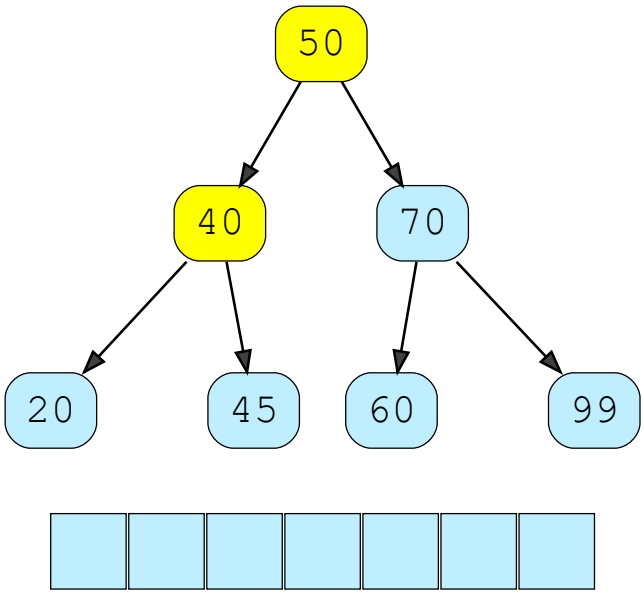
Let's step through a visualization to understand it better (Highly recommended to go to the last slide to see the complete order).



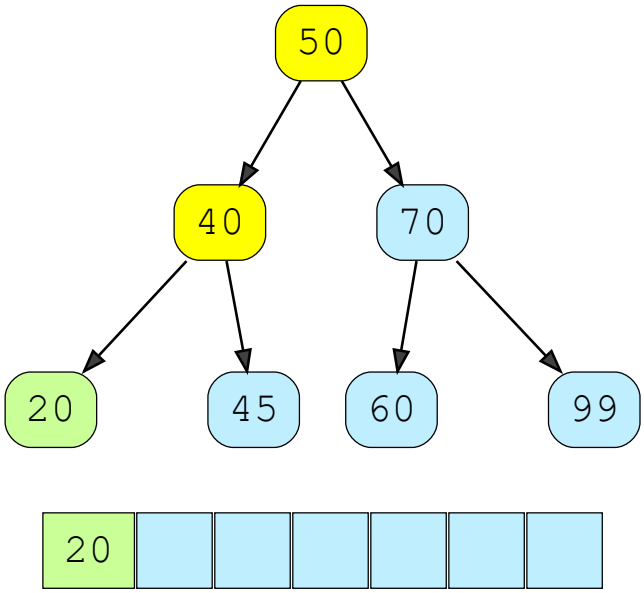
Start at Root 50. It has left subtree
Move to left child.



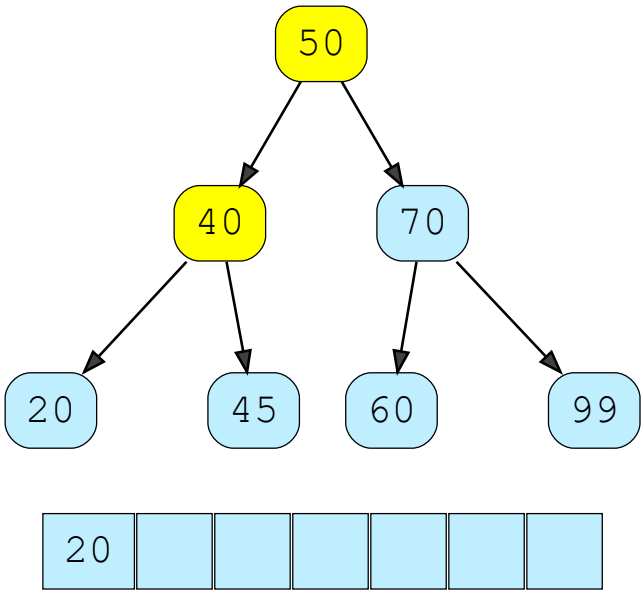
40 still has a left subtree.
Move to left child.



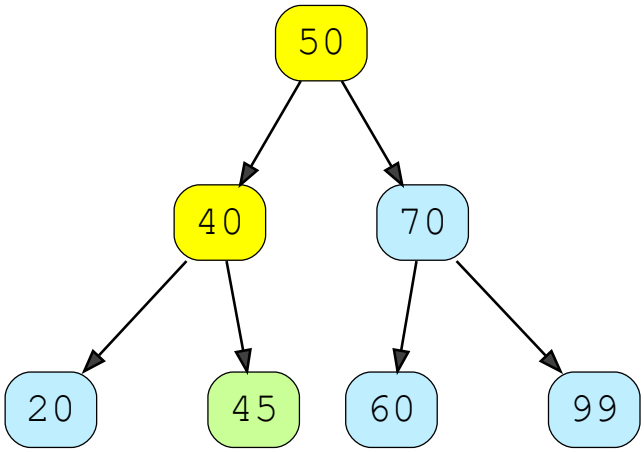
20 is a leaf node. Visit it.



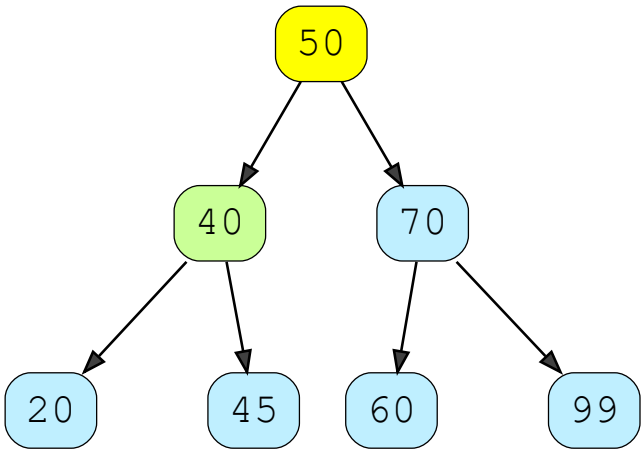
Done with 40's left subtree
Visit 40's right subtree.



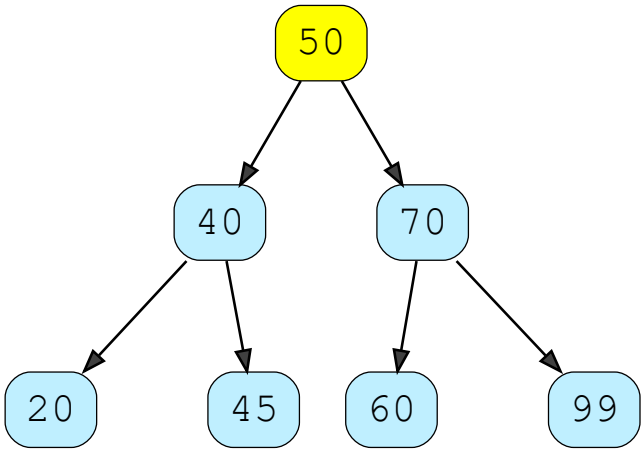
Start visiting 40's right subtree.
Visit 45.

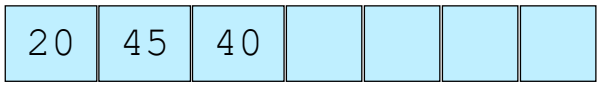


Done with visiting 40's both subtrees.
Visit 40 itself.

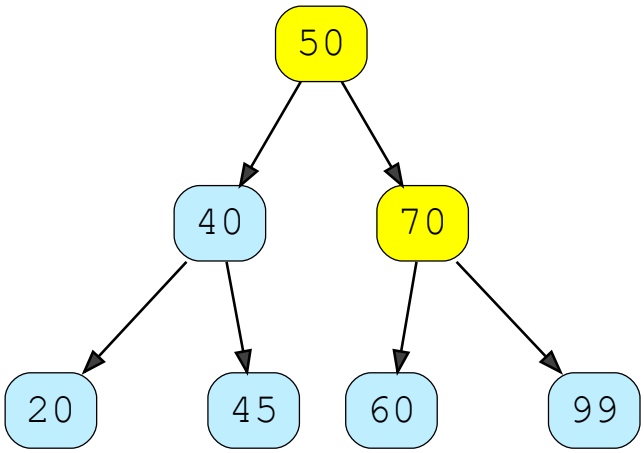


Done with 50's left subtree.
Start traversing 50's right subtree.

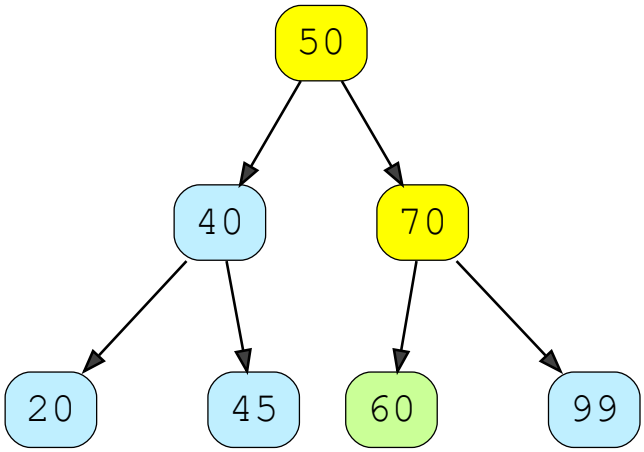


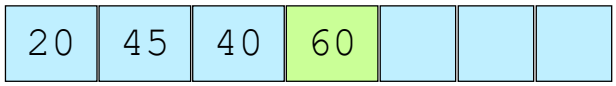


70 has left subtree.
Start traversing 70's left subtree.

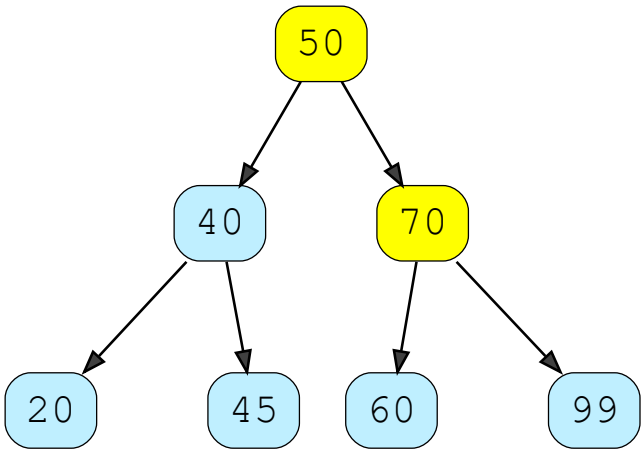


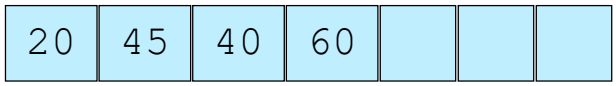
60 is leaf node. Visit it.



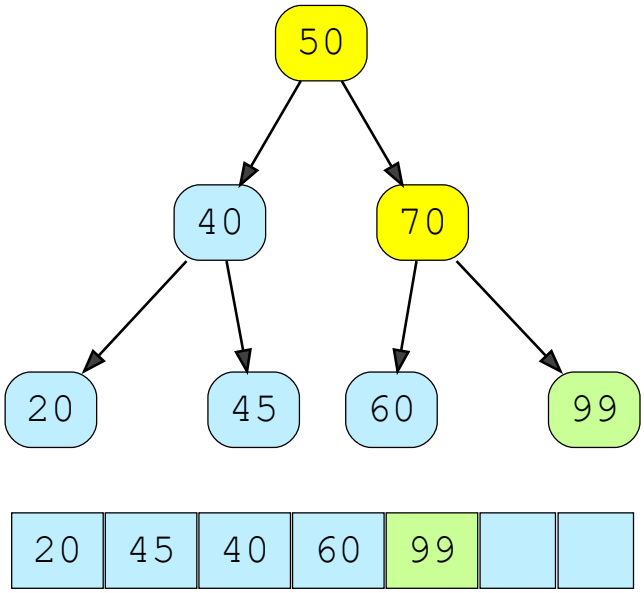


Done with 70's left subtree.
Start Visiting its right subtree.

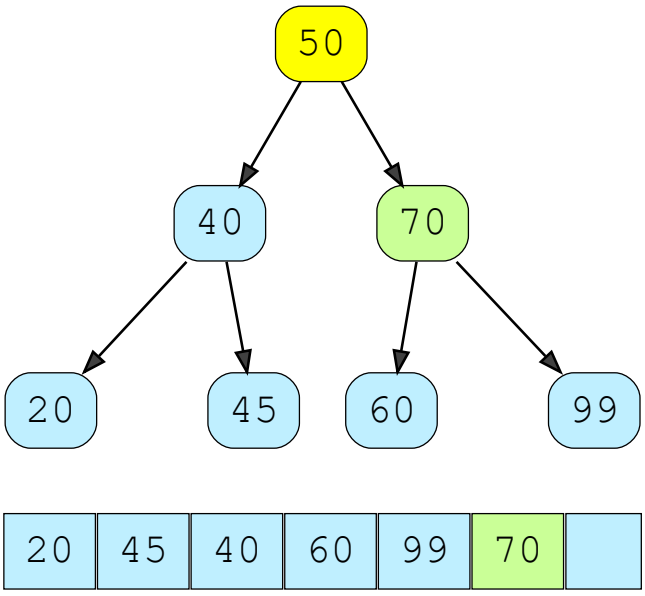


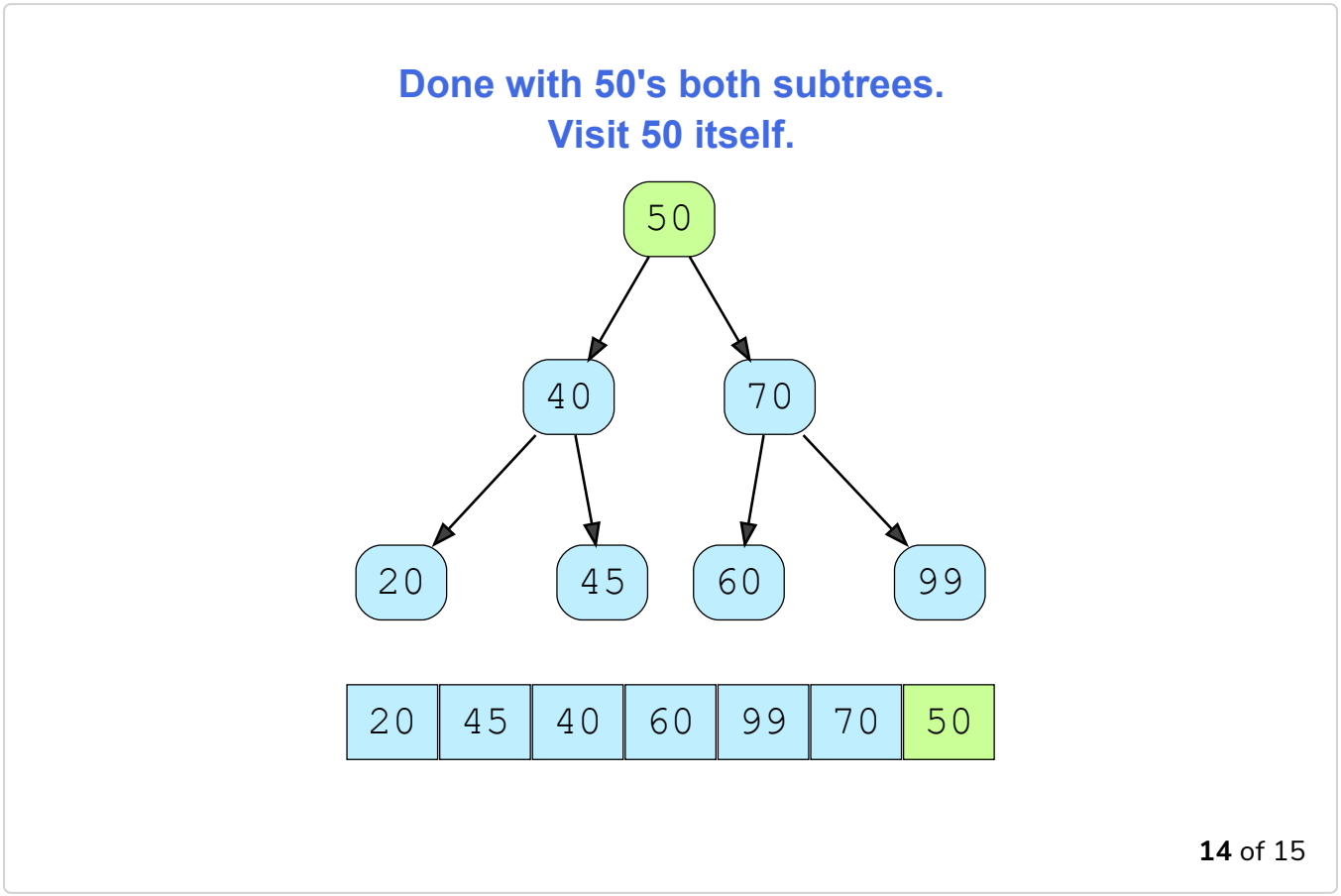


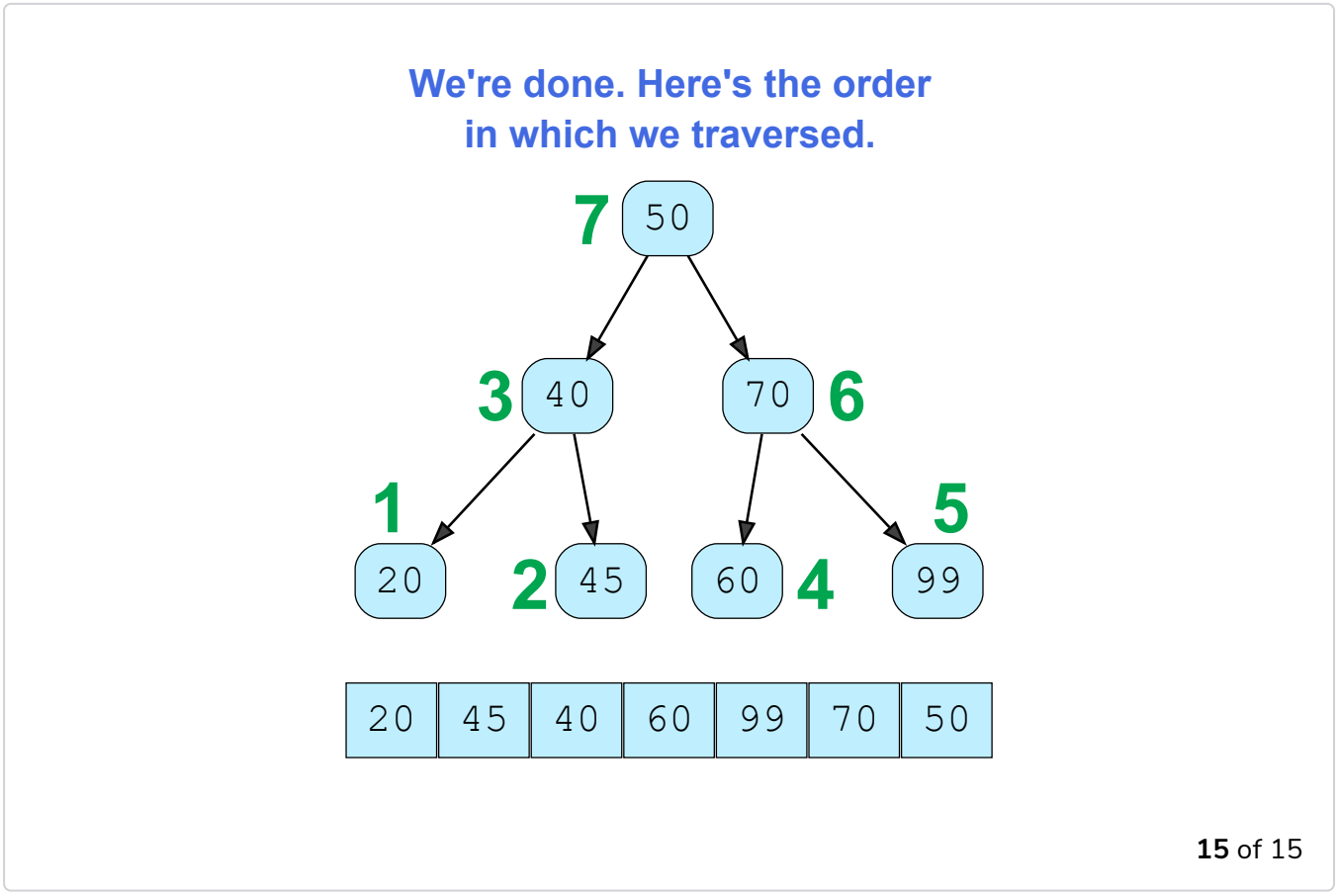
99 is leaf node. Visit it



Done with 70's both subtrees.
Visit 70 itself.







Let's take a look at the code for Post Order implementation (which is pretty obvious if you've already looked at the pre-order and in-order traversals).

```
// Returns Keys in the Post Order traversal
BST.prototype.postOrder = function() {
  var output = [];

  function postOrderImpl(node) {
    if (node === null) {
      return;
    }

    // Visit left subtree
    postOrderImpl(node.left);

    // Visit the right subtree
    postOrderImpl(node.right);

    // Visit the node itself.
    output.push(node.data);
  }

  // Call the internal function
  // with Root as the starting point.
  postOrderImpl(this._root);

  return output;
}
```

Traversals in action

Let's look at the traversals in action. We'll insert into a BST and then run all three traversals.

> Run the following code to see traversals in action

JavaScript

HTML

CSS (SCSS)

```
var data = [50, 40, 70, 60, 20, 99, 45];
var bst = new BST();

for (var i = 0; i < data.length; i++) {
  bst.insert(data[i]);
}

console.log("Pre-Order = " + bst.preOrder());
console.log("In-Order = " + bst.inOrder());
console.log("Post-Order = " + bst.postOrder());
```



Console

Clear

```
Pre-Order = 50,40,20,45,70,60,99
In-Order = 20,40,45,50,60,70,99
Post-Order = 20,45,40,60,99,70,50
```

In-Order Predecessor and In-Order Successor

Why do we need to understand In-Order predecessor and In-Order successor?

You might have noticed that we haven't discussed deletion in a BST (we only discussed insertion and search). The reason is that deletion is a little tricky in a BST and understanding In-Order predecessor and successor, helps in implementing deletion from a BST.

For a given node X,

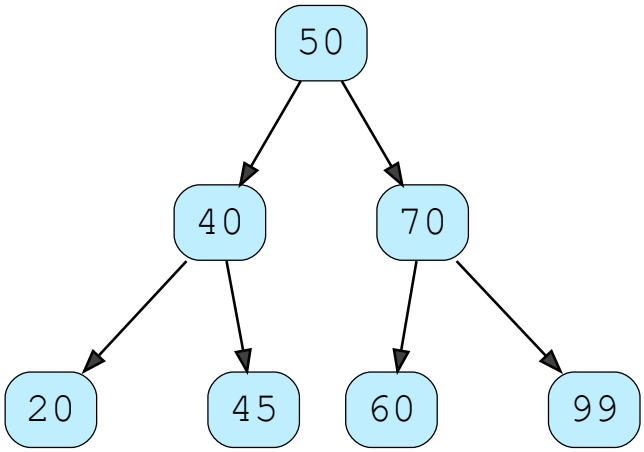
> Node X's ***predecessor*** is the node that comes just before X in InOrder traversal. Also remember that In-Order traversal visits the nodes in a sorted order. Hence, X's predecessor is the node with the largest key smaller than the key of X.

Similarly,

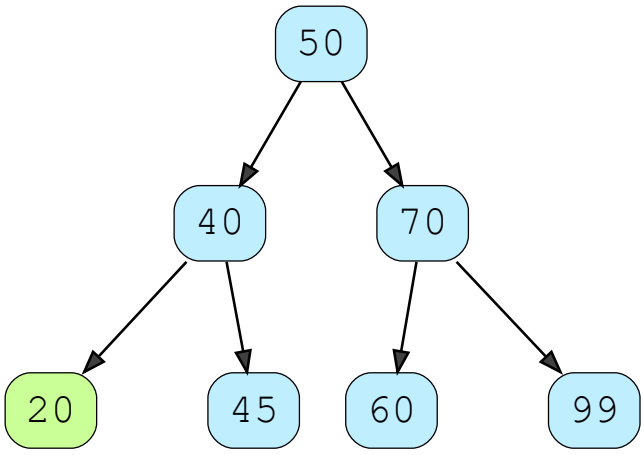
> Node X's ***successor*** is the node that comes right after X in tree's InOrder traversal. As In-Order traversal visits the nodes in a sorted order, it means that X's successor is the node with the smallest key larger than the key of X.

Time for some examples.

Let's look at
Predecessors and Successors

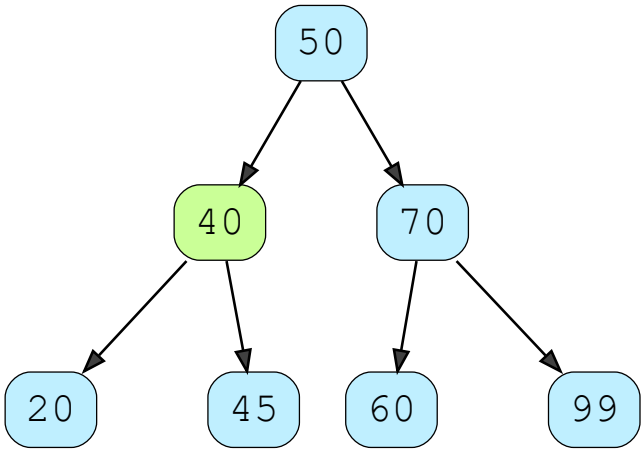


Let's look at 20's
Predecessor and Successor



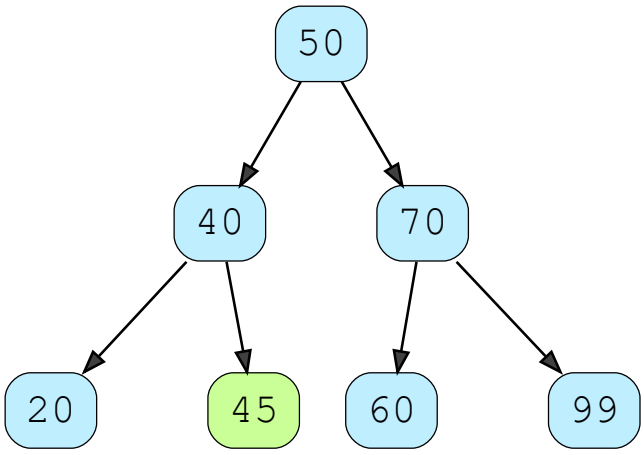
Predecessor is Null ***(no node smaller than 20)***
Successor is 40.

Let's look at 40's Predecessor and Successor

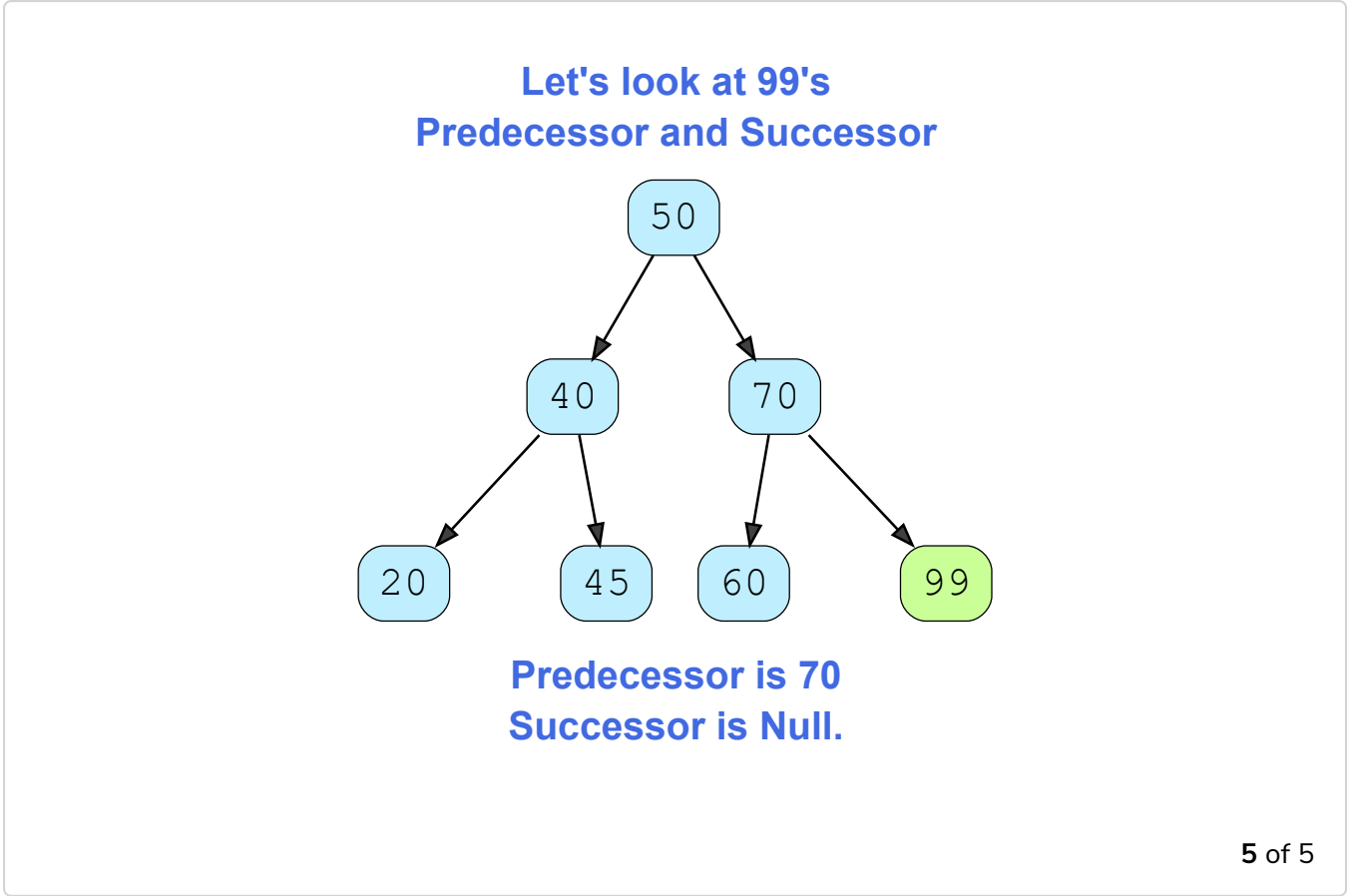


Predecessor is 20
Successor is 45.

Let's look at 45's Predecessor and Successor



Predecessor is 40
Successor is 50.



Deleting a node in a BST

Now that we've understood a lot of terminology about BST's and traversals, let's dive into deleting a Node in a BST. It's a little complicated as we need to ensure that the removal of the node doesn't break the invariant of the BST (left children smaller than the current key and right children larger than the current key).

When deleting a node in BST, there are three cases.

1. It's a leaf node and has no children.
2. The node has one child (either left or right).
3. The node has both left and right children.

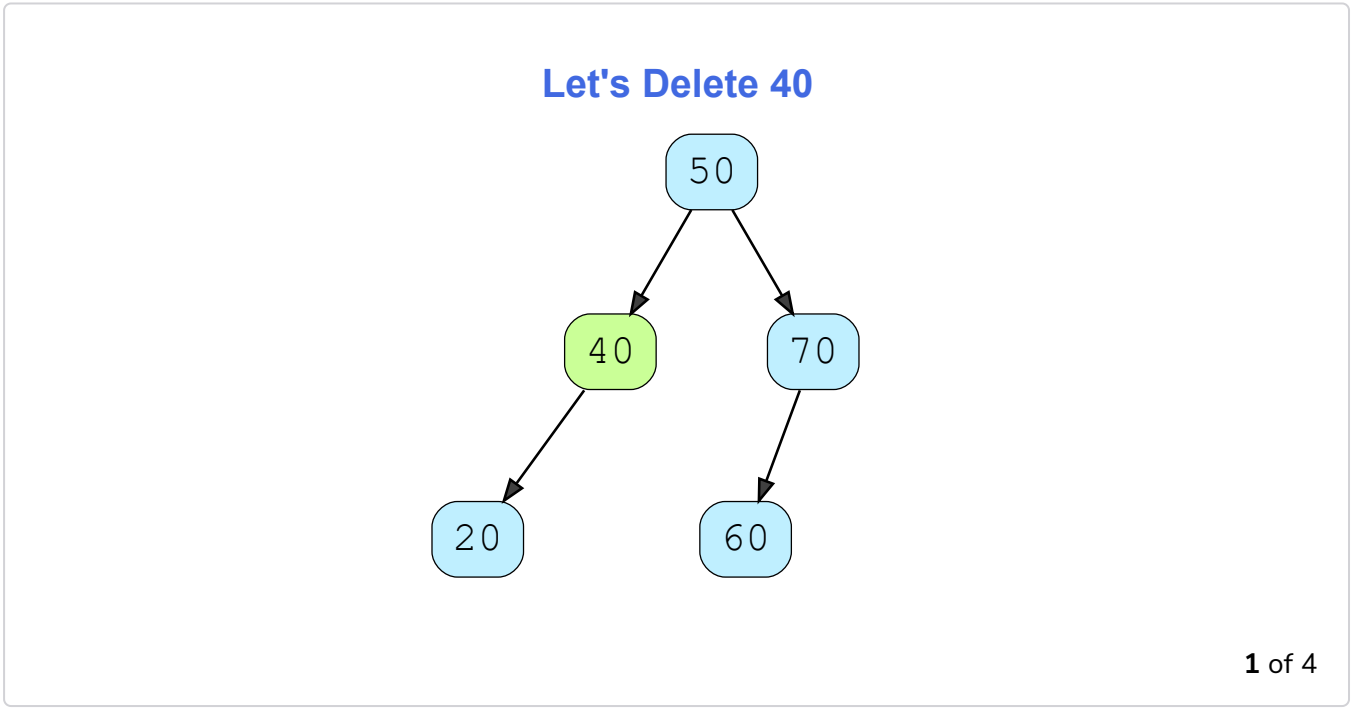
First two cases are easier to handle than the third one. Let's look at the easy ones first.

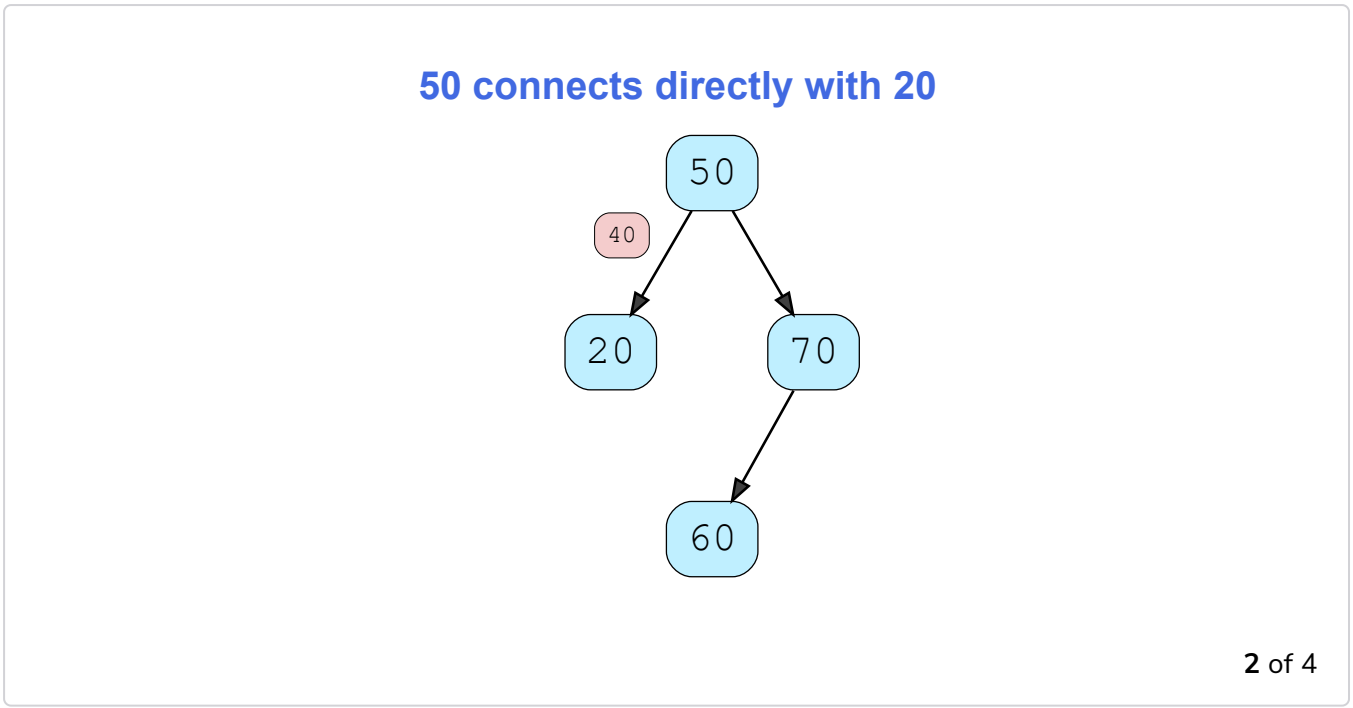
Deleting a Leaf Node

This is the easiest to delete. We just delete it and make sure that it's parent is not pointing to it anymore.

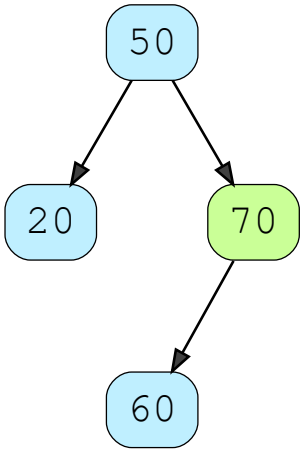
Deleting a node with 1 child

When deleting a node with 1 child, we connect the node's parent with node's child node. In other words, grand parent connects directly with grand child.



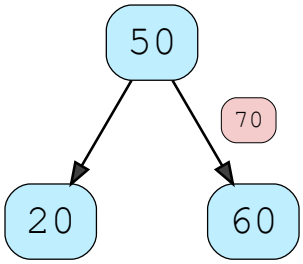


Now let's delete 70



3 of 4

50 connects directly with 60



4 of 4

Deleting a node with two children

When we have to delete a node with two children, we have two options. Let's say that the node to be deleted has key X.

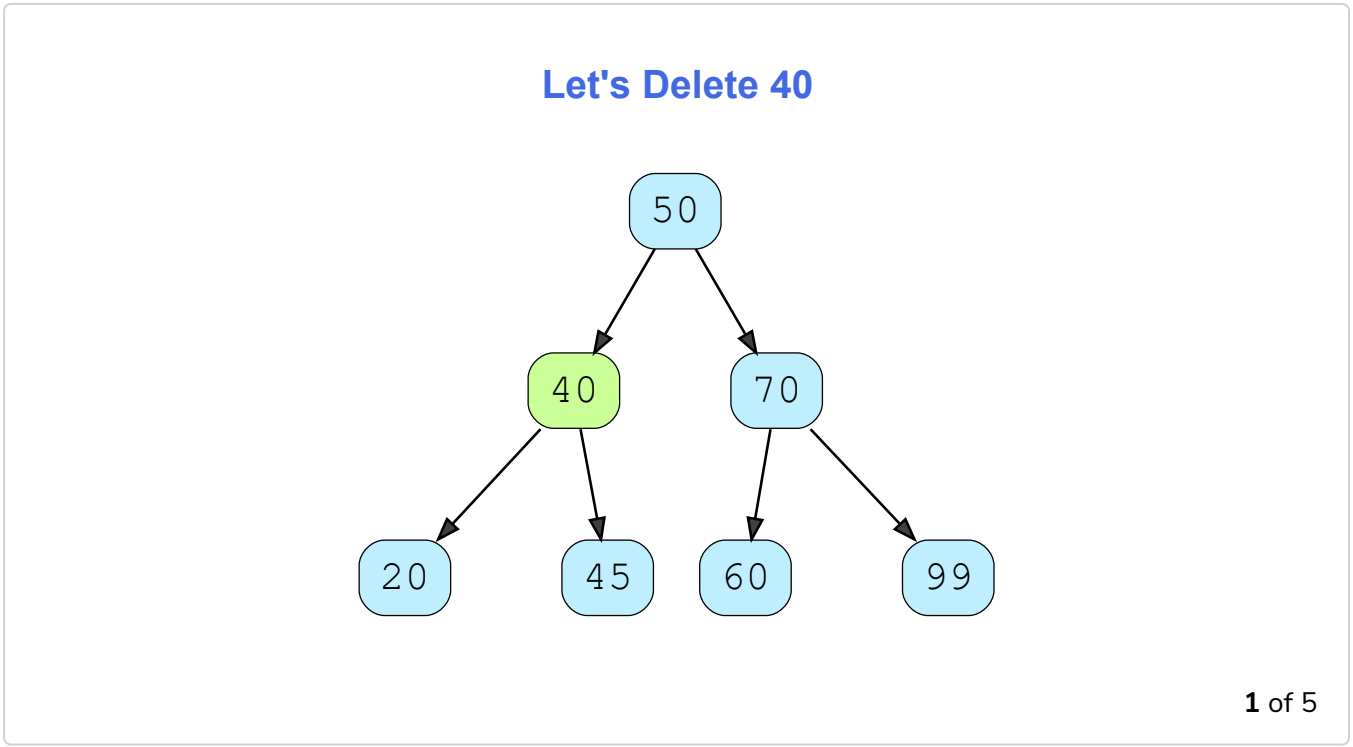
1. Replace the current node's key with its predecessor and then trigger delete for predecessor in node's left subtree.
2. Replace the current node's key with its successor and then trigger delete for successor in node's right subtree.

Why would it work?

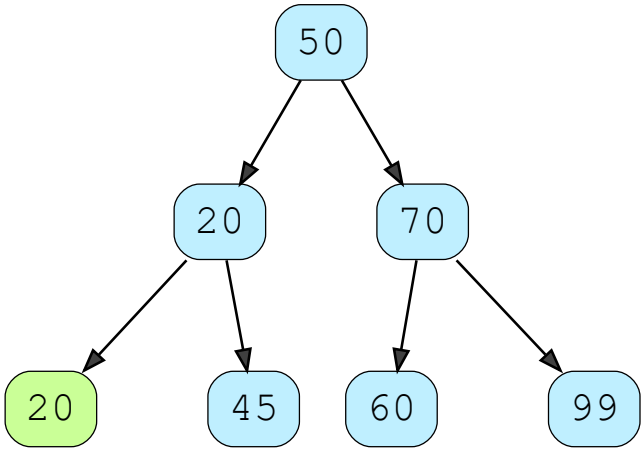
Remember predecessor is the highest value smaller than the current key. Now we know that this node has a left subtree. Hence, the predecessor to the current node is somewhere in the left subtree. In fact predecessor is the highest key in the node's left subtree. In addition, as it's the highest node in left subtree, it cannot have a right child. Otherwise that right child would have been the predecessor which ensures that we are now trying to delete a node with zero or one child. We already know that deleting a node with zero or one child is simpler and hence we've reduced the problem from a node with two children to deletion of node with zero or one child.

Similarly, successor is the smallest value larger than the current key. We already know that the node has a right subtree. Hence, the successor to the current node is somewhere in the right subtree. In fact, successor is the smallest key in node's right subtree. Hence, it's also a node with zero or one child.

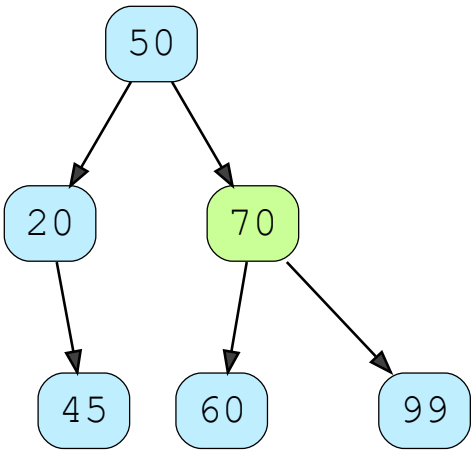
Let's look at a quick visualization



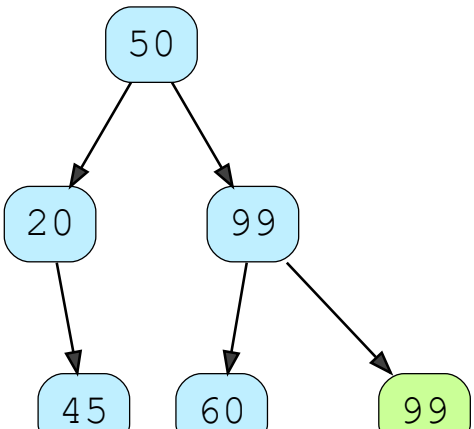
Replace 40 with 20, its predecesssor.
Then delete 20 in node's left subtree.



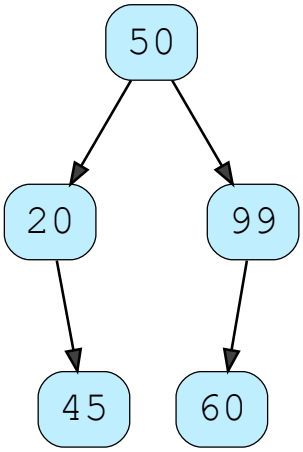
Now let's delete 70



Replace 70 with 99, it's successor.
Then delete 99 in node's right subtree.



Both deletions done.



Implementing Node Removal in BST

We'll implement a remove function that would take a key and delete the key ensuring that BST is left in the correct state after removal of the node. However, before we implement remove function, we need to decide what to do when the node to be deleted has two children. We have the option of either going the route of replacing it with predecessor or with its successor.

Let's pick predecessor. We know that the predecessor in the left subtree of the node will be the highest value of this subtree. Hence, we'll implement a quick maximum function.

```
BST.prototype.maximum = function(node) {
  while (node.right) {
    node = node.right;
  }

  return node.data;
}
```

Now let's implement the actual removal function.

```
BST.prototype.remove = function(key) {
  this.removeImpl(key, this._root);
}

BST.prototype.removeImpl = function removeImpl(key, node) {
  if (node != null) {
    if (key < node.data) {
      // Key might be in the left subtree.
      node.left = this.removeImpl(key, node.left);
    } else if (key > node.data) {
      node.right = this.removeImpl(key, node.right);
    } else {
      // Node found.
      // Let's see if it has two children.
      if (node.left && node.right) {
        // Replace current node with
        // predecessor data
        node.data = this.maximum(node.left);
        node.left = this.removeImpl(node.data, node.left);
      } else {
        // Only 1 child.
        // Let's return the child that's valid.
        node = node.left || node.right;
      }
    }
  }

  return node;
}
```

Let's run the removal code. We'll insert into BST, then delete a few nodes and see that In-Order traversal always returns sorted data.

> Run the following code to try out node removal

JavaScript

HTML

CSS (SCSS)

```
var data = [50, 40, 70, 60, 20, 99, 45];
var bst = new BST();

for (var i = 0; i < data.length; i++) {
  bst.insert(data[i]);
}

console.log("In-Order = " + bst.inOrder());

bst.remove(40);
console.log("In-Order BST after removing 40 = "
            + bst.inOrder());

bst.remove(70);
console.log("In-Order BST after removing 70 = "
            + bst.inOrder());
```

```
        + bst.inOrder());
```

Console

Clear

```
In-Order = 20,40,45,50,60,70,99
In-Order BST after removing 40 = 20,45,50,60,70,99
In-Order BST after removing 70 = 20,45,50,60,99
```

Quick Quiz about Trees

1 Trees are what type of data structure

2 Which Traversal returns data in sorted order

Check Answers

Exercise

Exercise

Implement a function to count the number of elements in the tree.

> Some Tests are failing as size method is not implemented. Implement it to fix the test cases.

JavaScript

HTML

CSS (SCSS)

```
BST.prototype.size = function() {
  // Implement this
  return undefined;
}

// This is the method that runs test cases.
// Don't remove this call.
runEvaluation();
```



Console

Clear

```
*** There is some bug lurking there. See failed test cases ***
Here are the tests that ran:
Test case FAILED for bst.size(BST = []. Result: undefined. Expected: 0
Test case FAILED for bst.size(BST = [50]. Result: undefined. Expected: 1
Test case FAILED for bst.size(BST = [40,50]. Result: undefined. Expected: 2
Test case FAILED for bst.size(BST = [40,50,70]. Result: undefined. Expected: 3
```

Summary

- Binary tree is a hierarchical data structure.
- Binary Search Tree (BST) is a binary tree where a node has all smaller keys to its left and keys greater than the node on its right.
- Insertion and Search in a BST are relatively straight forward.

- Removal is a little tricky as deletion of a node can cause the BST to violate its properties.
- Binary Trees can be traversed in many orders. Most common are Pre-Order, In-Order and Post-Order traversals.