Introduction - Insertion and Search

In this lesson, you will learn how to implement Binary Search Trees in Python and how to insert and search elements within them.

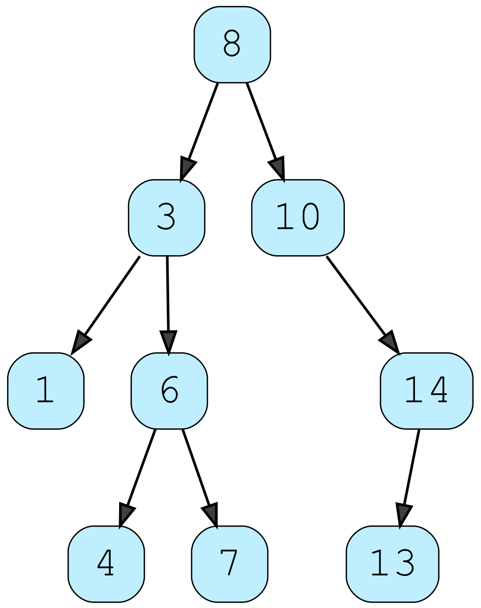
WE'LL COVER THE FOLLOWING

In this lesson, we will go over the binary search tree data structure. We will first cover the general idea of what a binary search tree is and how one may go about inserting data into this structure as well as how one searches for data. Once we cover the general idea, we will move over to the implementation of the binary search tree data structure in Python. We will construct two class methods that will implement the search and insertion algorithms.

A **binary search tree** is a tree data structure in which nodes are arranged according to the BST property which is as follows:

The value of the left child of any node in a binary search tree will be less than whatever value we have in that node, and the value of the right child of a node will be greater than the value in that node.

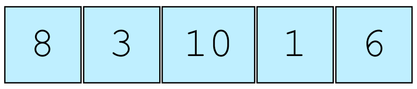
This type of pattern persists throughout the tree. Let's look at an example illustrated below:



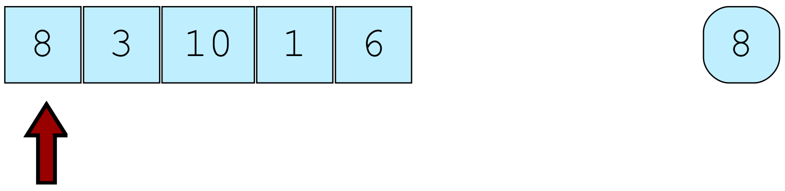
A Binary Search Tree

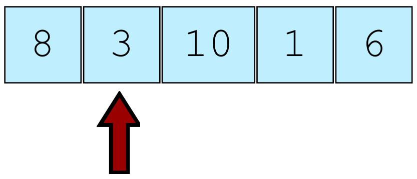
Insertion

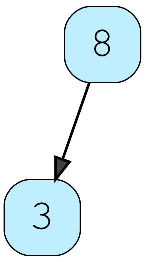
Let’s see how we can insert elements in a binary search tree. In the illustration below, we have an array of elements where we want to insert all the elements one by one into the binary search tree. The way we go about insertion is to start from the root node and compare the new value to be inserted with the current node’s value. If the new value is less than the value of the current node, then the new value must be inserted in the left subtree of the current node. On the other hand, if it’s greater, it must be inserted in the right subtree to satisfy the BST property. We keep comparing the values as we traverse the tree and insert wherever we find a null position.



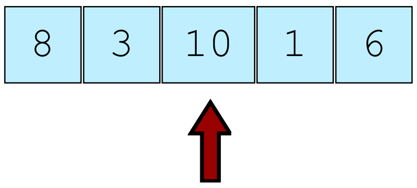
Let's make a BST using the values from the array.

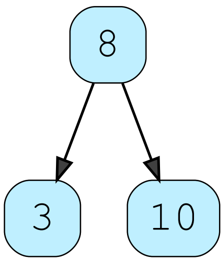




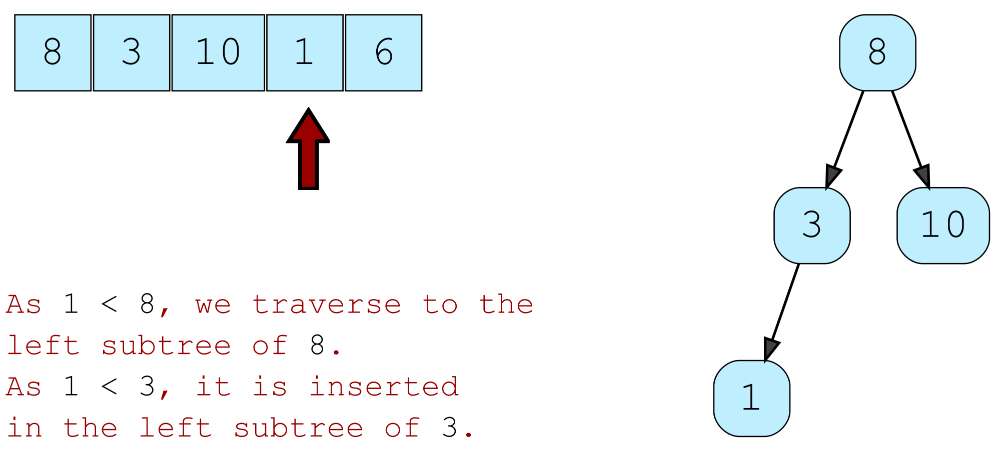


As 3 < 8, it is inserted
in the left subtree of 8.



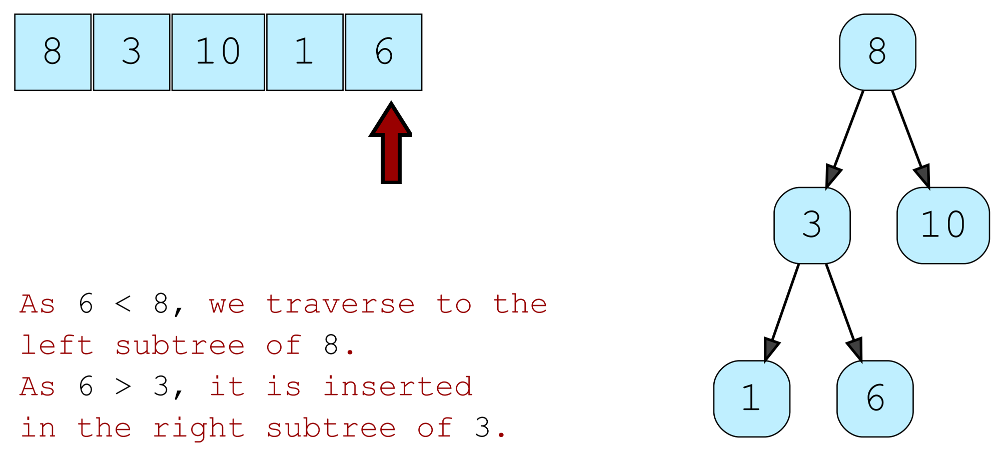


As 10 > 8, it is inserted
in the right subtree of 8.



As 1 < 8, we traverse to the left subtree of 8.
As 1 < 3, it is inserted in the left subtree of 3.

5 of 6

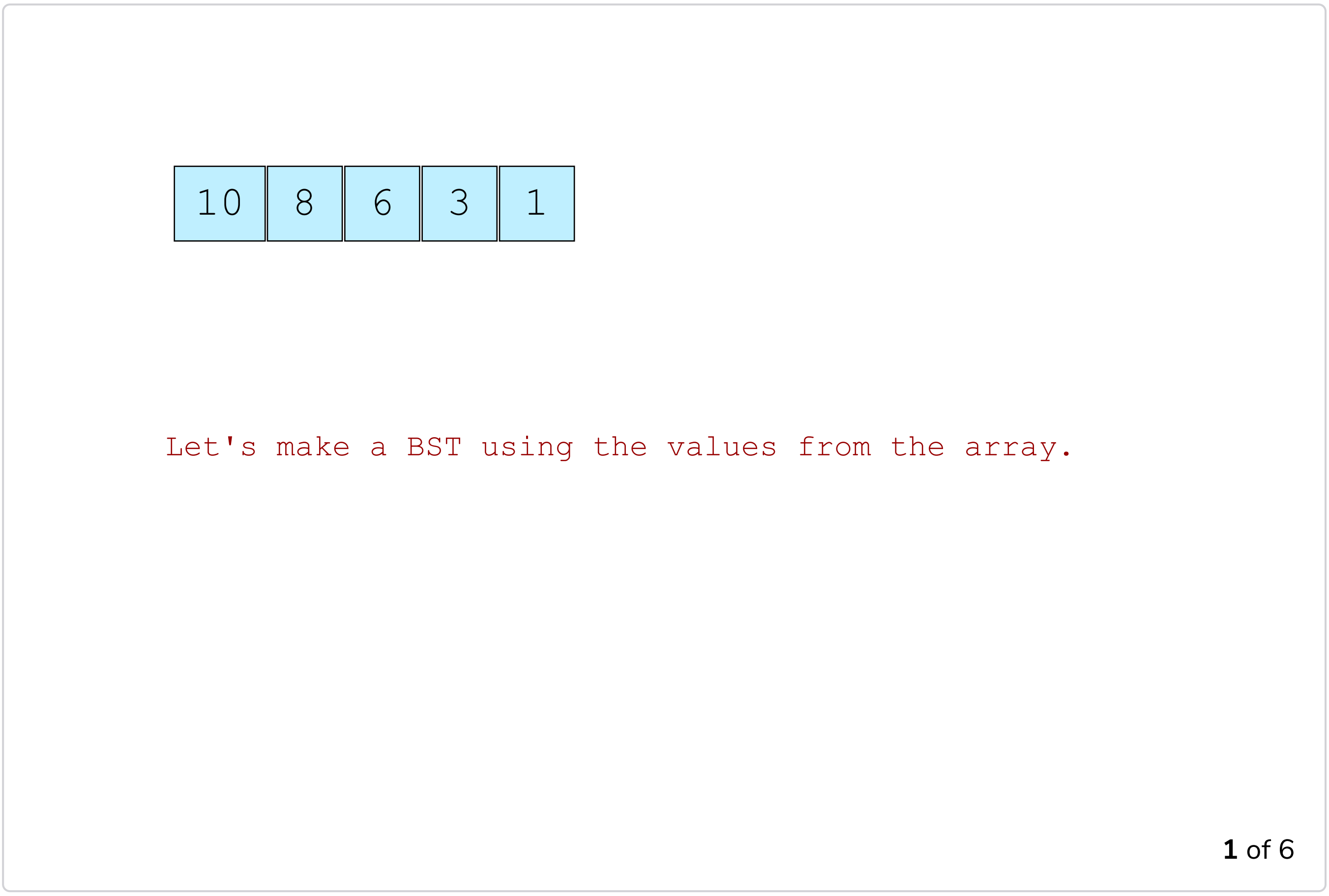


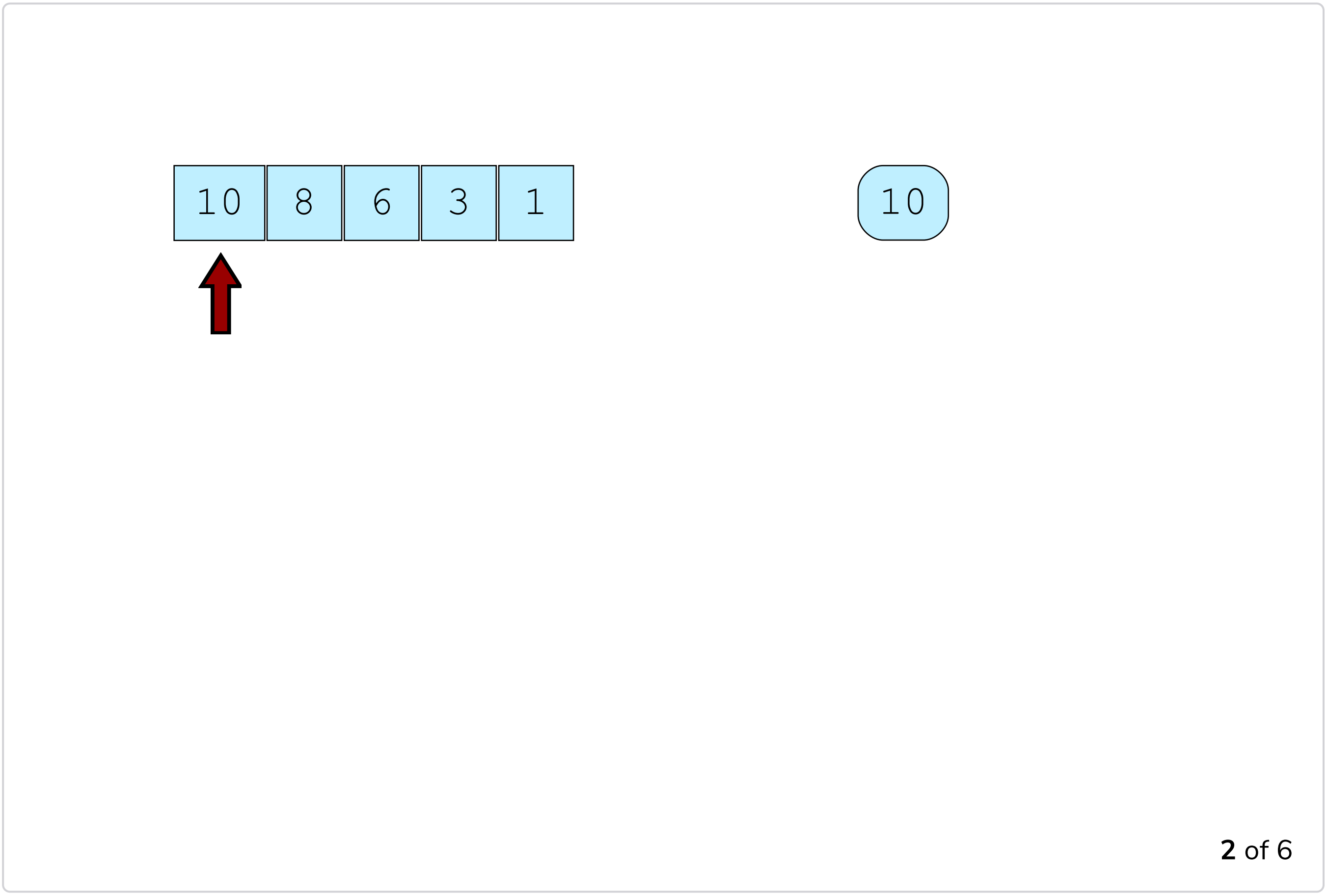
As 6 < 8, we traverse to the left subtree of 8.
As 6 > 3, it is inserted in the right subtree of 3.

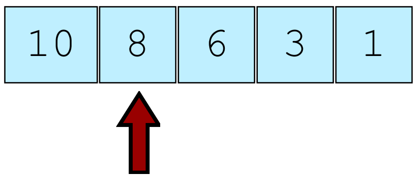
6 of 6

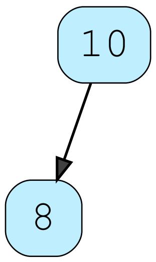
Insertion of a Reverse Sorted List

Let's see what happens if we insert elements from a reverse sorted list one by one in a binary search tree.

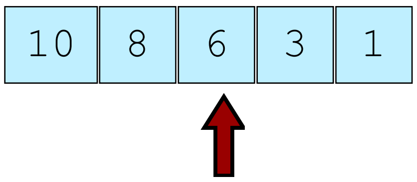


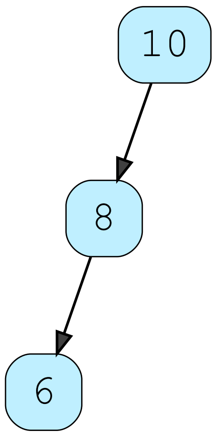




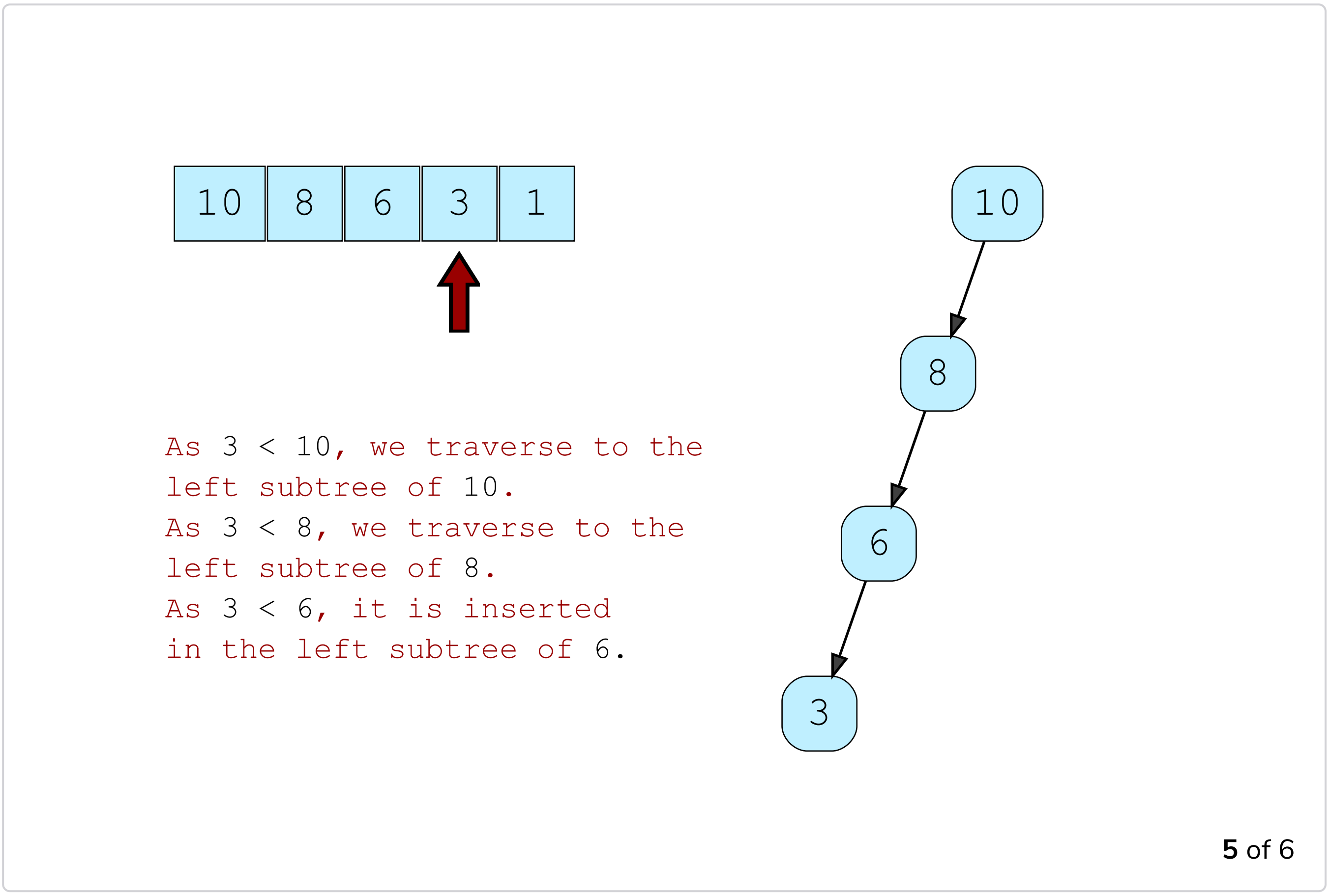


As 8 < 10, it is inserted
in the left subtree of 10.

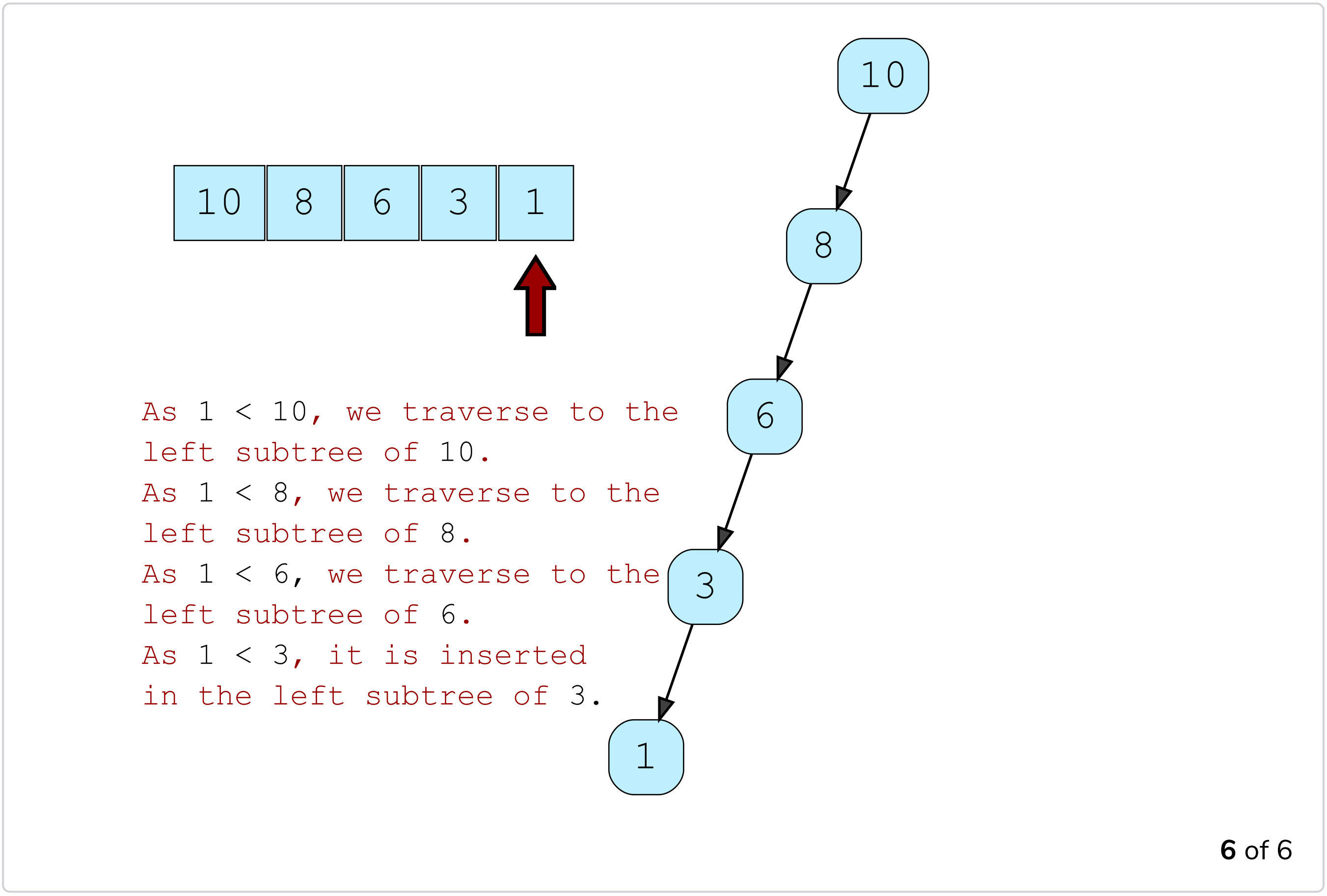




As 6 < 10, we traverse to the
left subtree of 10.
As 6 < 8, it is inserted
in the left subtree of 8.



As 3 < 10, we traverse to the left subtree of 10.
As 3 < 8, we traverse to the left subtree of 8.
As 3 < 6, it is inserted in the left subtree of 6.



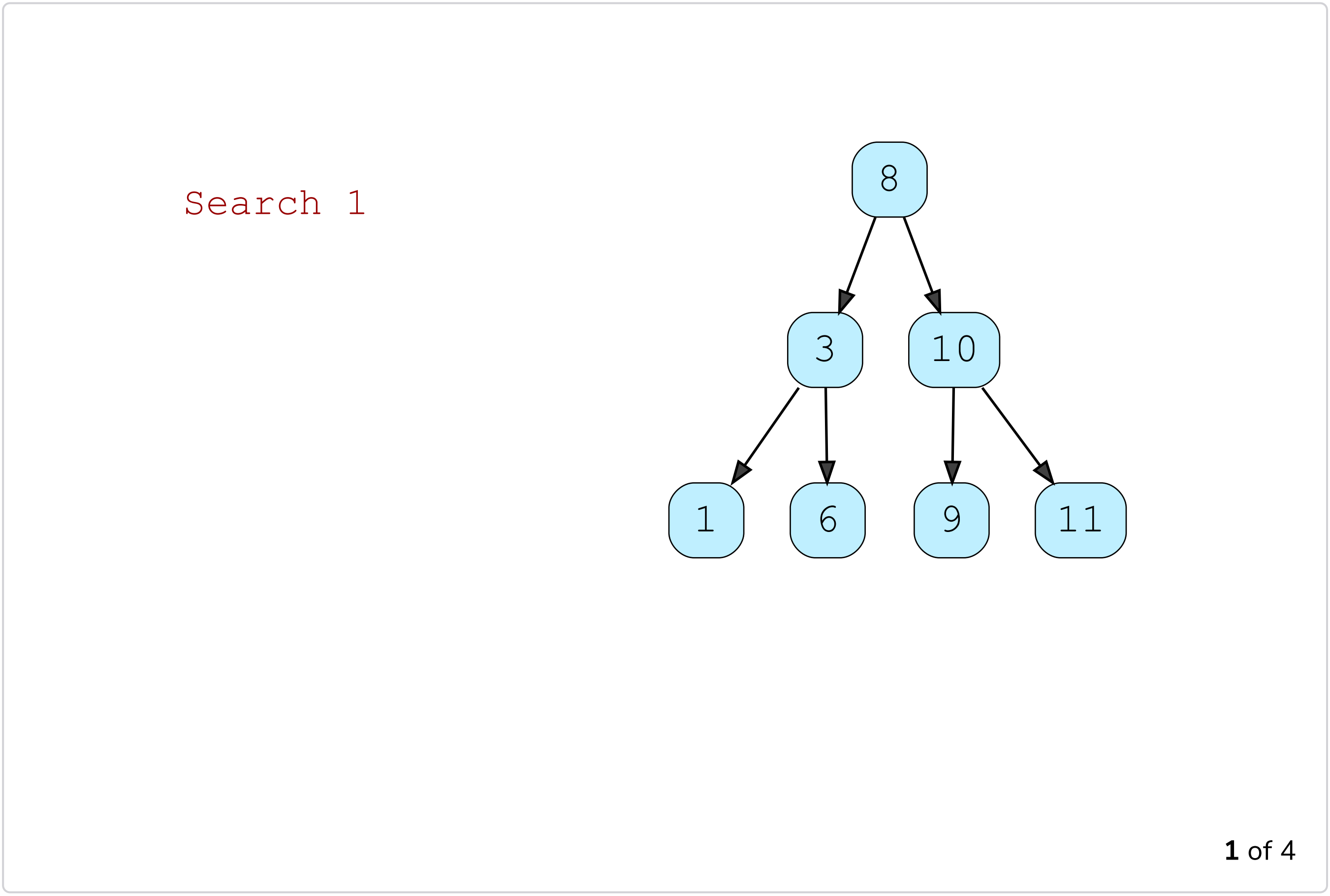
As 1 < 10, we traverse to the left subtree of 10.
As 1 < 8, we traverse to the left subtree of 8.
As 1 < 6, we traverse to the left subtree of 6.
As 1 < 3, it is inserted in the left subtree of 3.

As you can see, we end up with a linear sort of a binary search tree. In such a case, if you want to find **node 1** in the above binary search tree, you might have to traverse down. However, if you have a structure other than the linear one, then you can substantially cut down the number of operations that you would have to do to find **node 1**. We will look at that sort of a structure when we cover searching. The linear binary search tree is a structure we want to avoid when we perform operations on binary search trees because it kills the purpose of having a binary search tree. If you're curious about how you can prevent getting the structure from reading data and forming a binary search tree out of it, then you can look into something called an AVL tree. An AVL tree rebalances the nodes so that you get a hierarchically spread-out structure instead of a linear one. However, that is beyond the scope of this lesson.

Search

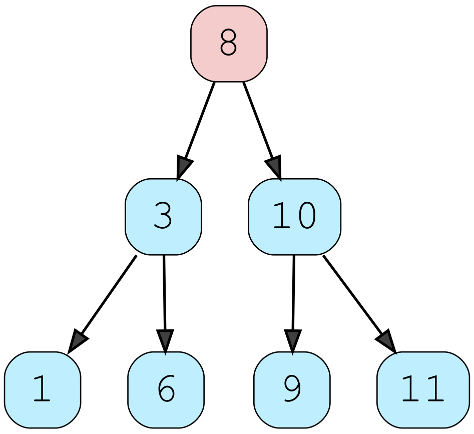
We search using a similar approach as we used in the insertion in a binary search tree. Starting from the root node, we decide which subtree to traverse by comparing the value to be searched with the current node. Then we traverse to the appropriate subtree and discard the other subtree that does not contain the element we are searching for due to the BST property.

Have a look at the illustrations to get a fair idea:



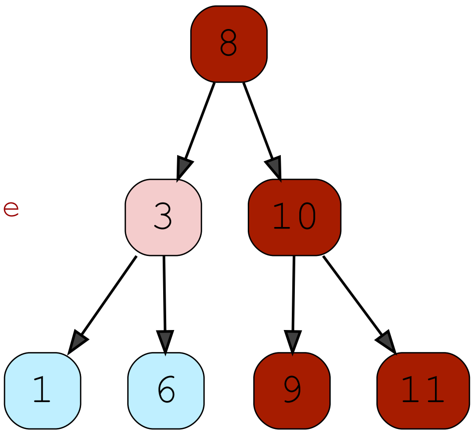
Search 1

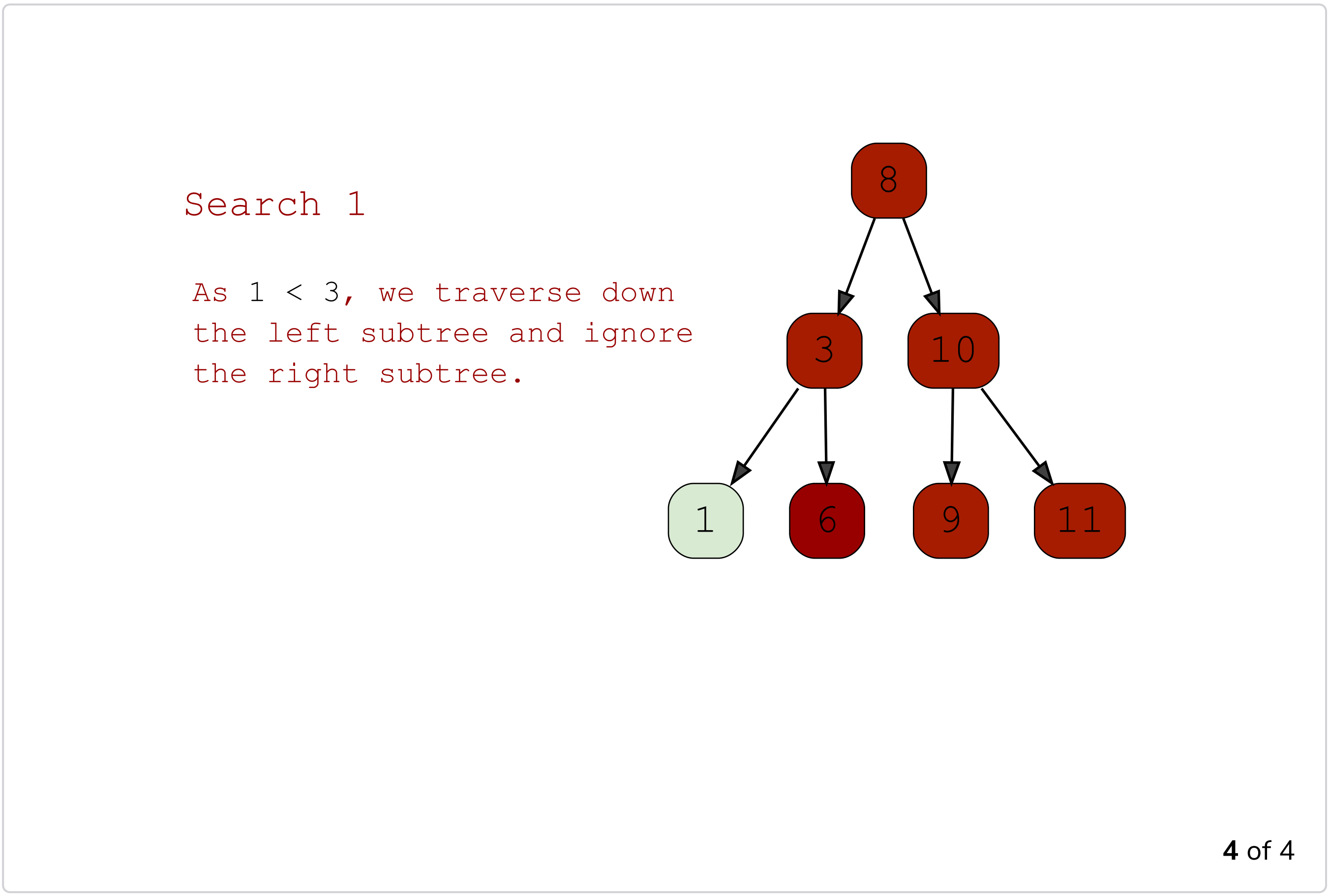
Start from the root node



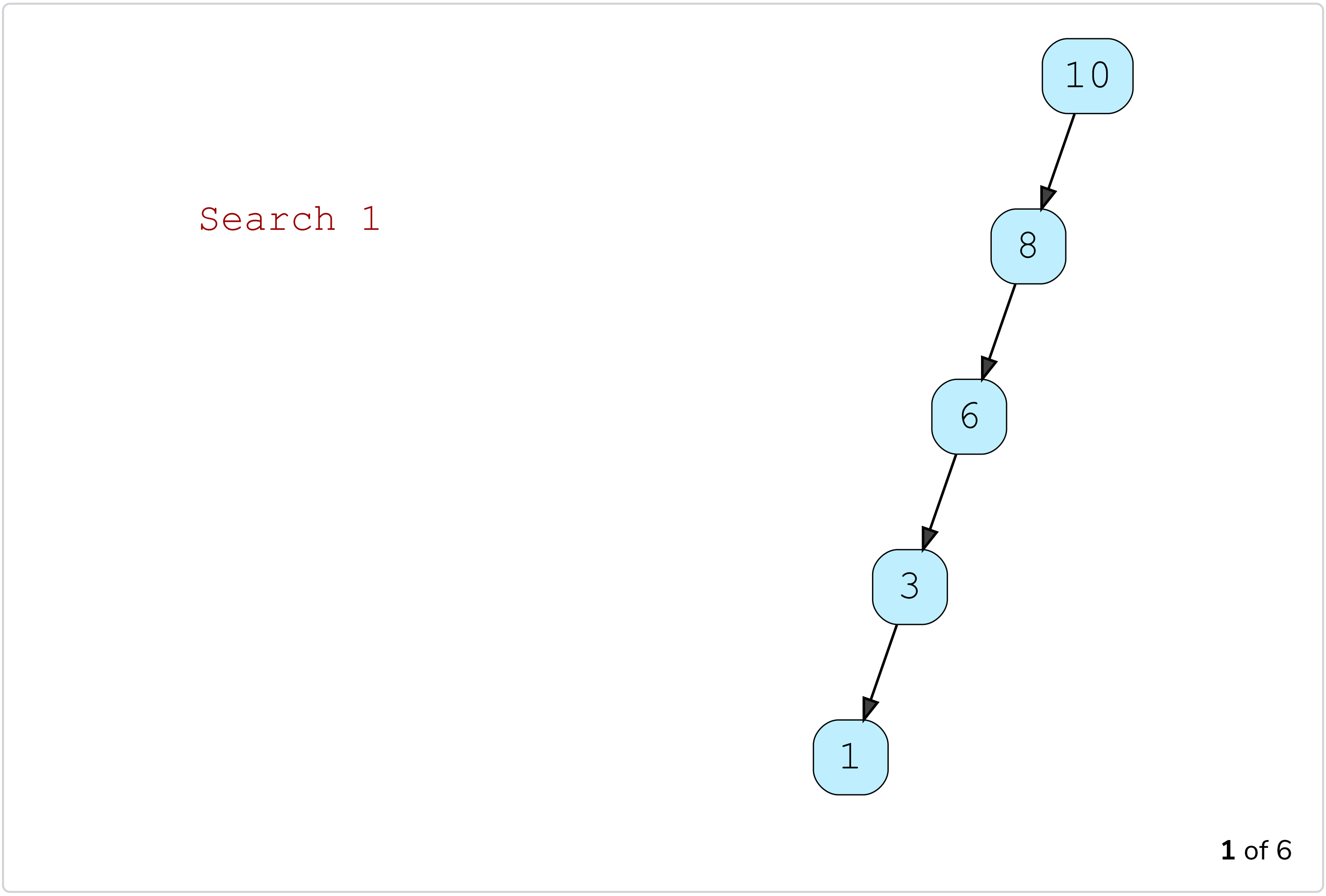
Search 1

As 1 < 8, we traverse down the left subtree and ignore the right subtree.



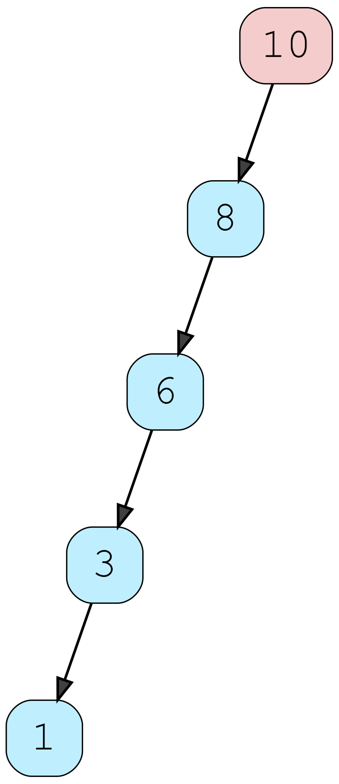

Search 1
As 1 < 3, we traverse down the left subtree and ignore the right subtree.
8
3
10
1
6
9
11
4 of 4

Now let’s also illustrate an example to search on a linear binary tree.



Search 1

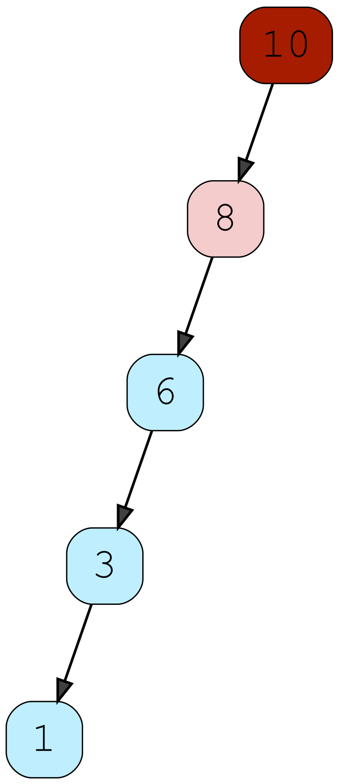
Start from the root node.



2 of 6

Search 1

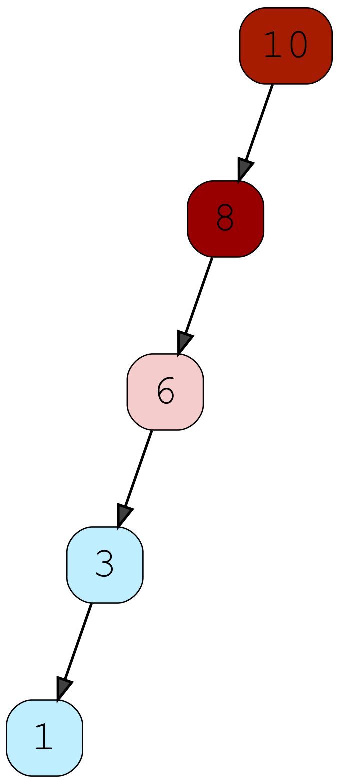
As 1 < 10, we traverse down the left subtree and ignore the right subtree.



3 of 6

Search 1

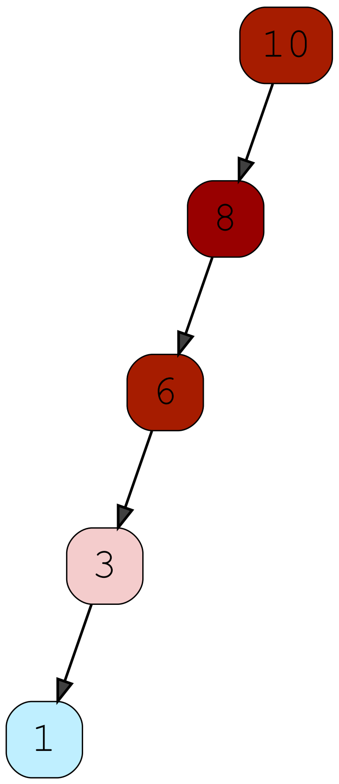
As 1 < 8, we traverse down the left subtree and ignore the right subtree.



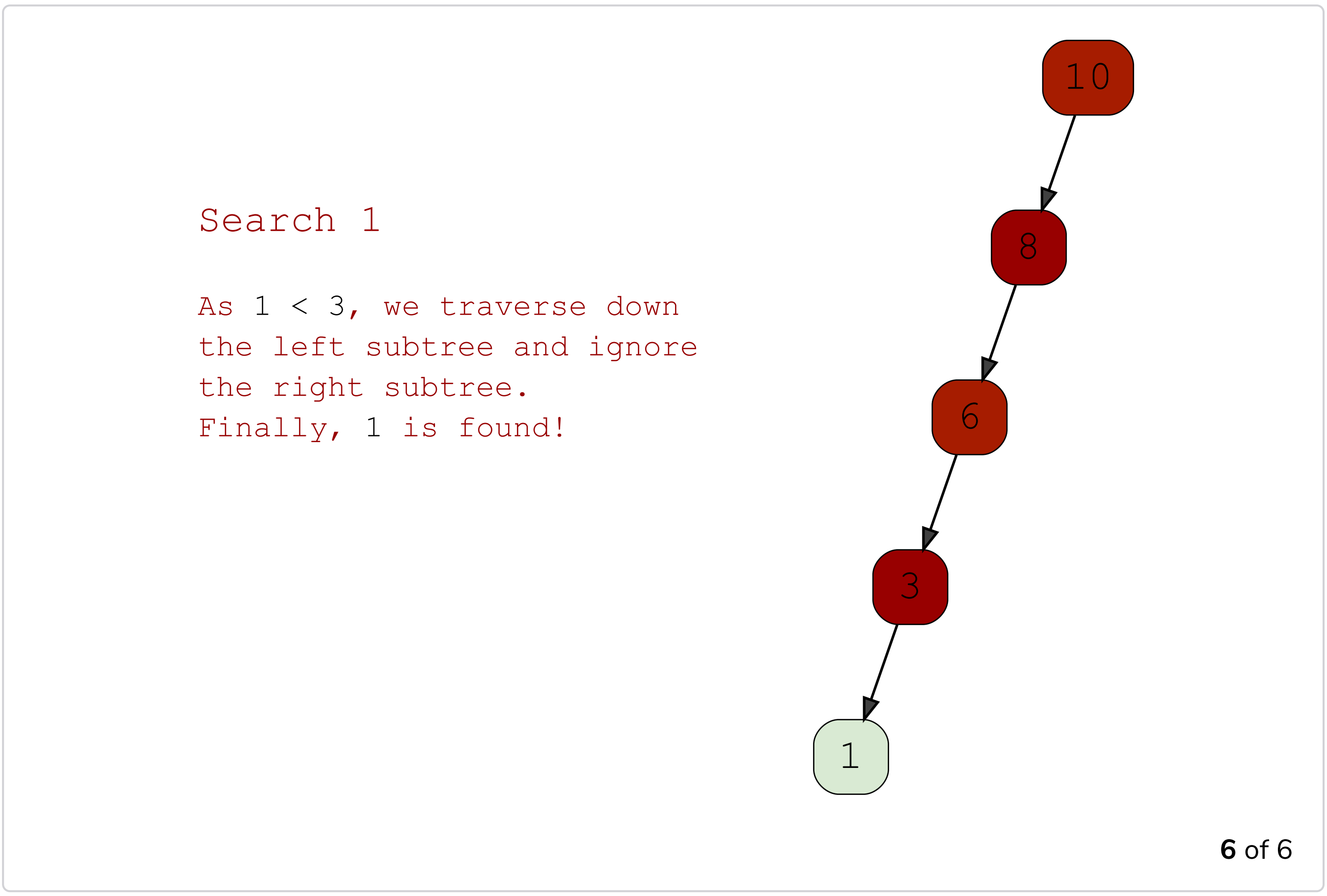
4 of 6

Search 1

As 1 < 6, we traverse down the left subtree and ignore the right subtree.



5 of 6


Search 1

As 1 < 3, we traverse down the left subtree and ignore the right subtree. Finally, 1 is found!

As you can see from the illustration above, we don't discard anything in a linear structure, so we have to traverse *all the nodes* to find the node we are looking for. This makes it an operation of $O(n)$ in the worst case. If we have a non-linear structure for a BST, the time complexity significantly improves to $O(logn)$.

This table summarizes the time complexities for a BST:

Algorithm	Average Case	Worst Case
Search	$O(logn)$	$O(n)$
Insert	$O(logn)$	$O(n)$
Delete	$O(logn)$	$O(n)$

Now let's jump to the implementations in Python!

Implementation

The implementation of the `BST` class is similar to the implementation of the `BinaryTree` class. The `Node` class is exactly the same as the `Node` class in the Binary Trees chapter.

Have a look at the implementation below:

```
class Node(object):
  def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None


class BST(object):
  def __init__(self, root):
    self.root = Node(root)
```

class Node and class BST

`insert`

Now we'll look into the implementation of `insert` operation in Python.

```
def insert(self, new_val):
  self.insert_helper(self.root, new_val)

def insert_helper(self, current, new_val):
  if current.data < new_val:
    if current.right:
      self.insert_helper(current.right, new_val)
    else:
      current.right = Node(new_val)
  else:
      if current.left:
        self.insert_helper(current.left, new_val)
      else:
        current.left = Node(new_val)
```

insert(self, new_val) and insert_helper(self, current, new_val)

The `insert` method on **line 1** invokes the helper function (`insert_helper`) on **line 2**, while also passing `self.root` and `new_val` into that method. In the `insert_helper` method, we have `current` and `new_val` which refer to the current node and the new value to be inserted. In order to find the appropriate location to insert `new_value`, we compare `current.data` to `new_value`. If `current.data` is less than `new_val`, then `new_val` must be placed somewhere in the right subtree. Therefore, we check on **line 6** if `current.right` is `None`. If it's not, this implies that we have to traverse the tree further and we recursively call `insert_helper` on **line** 7 and pass `current.right` and `new_value` to it. On the other hand, if `current.right` is `None`, then `current` must be a leaf node, and we have found an appropriate position to insert the `new_val`. Then we initialize `current.right` to a new `Node` by passing `new_val` to the constructor on **line 9**.

Let's talk about when `current.data` is greater than or equal to `new_val`. If that's the case, we repeat what we discussed above about the code in **lines 6-9**, but replace `current.right` with `current.left` as `new_val` must be placed in the left subtree of the current node.

`search`

Let's discuss the `search` method:

```
def search(self, find_val):
  return self.search_helper(self.root, find_val)

def search_helper(self, current, find_val):
  if current:
    if current.data == find_val:
        return True
    elif current.data < find_val:
        return self.search_helper(current.right, find_val)
    else:
        return self.search_helper(current.left, find_val)
```

search(self, find_val) and search_helper(self, current, find_val)

If you get the hang of the insert operation, then the `search` method is relatively simple to understand. Like the `insert` method, the `search` method invokes a helper function (`search_helper`) and passes the root node and the value to search (`find_val`) to the helper method. It also returns whatever is returned from the helper method.

In `search_helper`, we check on **line 5** if `current` is `None`. If it's not, then the execution jumps to **line 6** where the condition acts as a sort of base case to the recursive method. If the value of the current node (`current.data`) is equal to `find_val`, then we have found the node that we are supposed to search. `True` is returned on **line 7** to send a confirmation back to the caller method. However, if the condition on **line 6** does not evaluate to `True`, we check on

line 8 to see if `current.data` is less than `find_val`. If this evaluates to `True`, `find_val` must be in the right subtree of the current node to which we make a recursive call on **line 9**. Otherwise, if `current.data` is greater than `find_val`, we make a recursive call to the left subtree on **line 11**.

The entire code that we have implemented can be found below. Make a BST and insert elements to it to verify the `insert` and the `search` methods.

```
class Node(object):
  def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None


class BST(object):
  def __init__(self, root):
    self.root = Node(root)

  def insert(self, new_val):
    self.insert_helper(self.root, new_val)

  def insert_helper(self, current, new_val):
    if current.data < new_val:
      if current.right:
        self.insert_helper(current.right, new_val)
      else:
        current.right = Node(new_val)
    else:
        if current.left:
          self.insert_helper(current.left, new_val)
        else:
          current.left = Node(new_val)

  def search(self, find_val):
    return self.search_helper(self.root, find_val)

  def search_helper(self, current, find_val):
    if current:
      if current.data == find_val:
          return True
      elif current.data < find_val:
          return self.search_helper(current.right, find_val)
      else:
          return self.search_helper(current.left, find_val)

bst = BST(10)
bst.insert(3)
bst.insert(1)
bst.insert(25)
bst.insert(9)
bst.insert(13)

print(bst.search(9))
print(bst.search(9))
print(bst.search(14))
```

Brace yourself for a challenge in the next lesson!