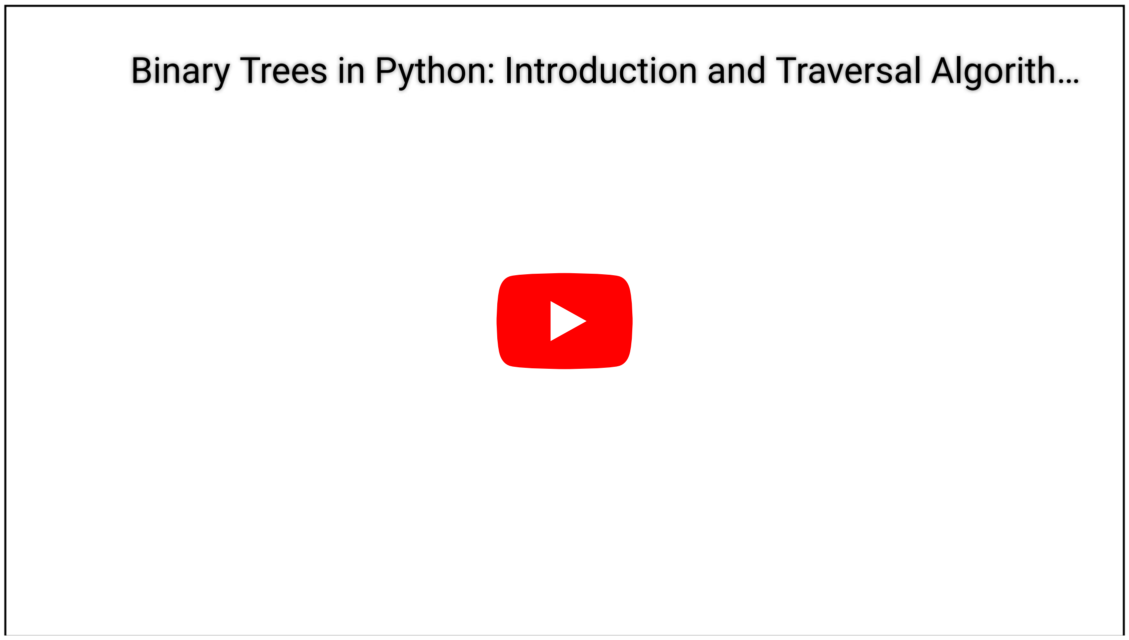
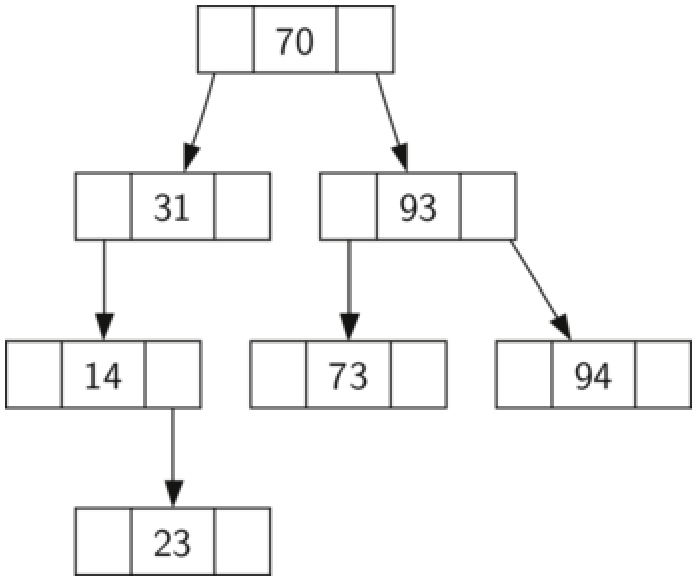
Binary Search Trees


Binary Trees in Python: Introduction and Traversal Algorith...

A binary search tree relies on the property that keys that are less than the parent are found in the left subtree, and keys that are greater than the parent are found in the right subtree. We will call this the **bst property**. As we implement the Map interface as described above, the bst property will guide our implementation. The figure below illustrates this property of a binary search tree, showing the keys without any associated values. Notice that the property holds for each parent and child. All of the keys in the left subtree are less than the key in the root. All of the keys in the right subtree are greater than the root.



Now that you know what a binary search tree is, we will look at how a binary search tree is constructed. The search tree above represents the nodes that exist after we have inserted the following keys in the order shown: $70, 31, 93, 94, 14, 23, 73$. Since 70 was the first key inserted into the tree, it is the root. Next, 31 is less than 70, so it becomes the left child of 70. Next, 93 is greater than 70, so it becomes the right child of 70. Now we have two levels of the tree filled, so the next key is going to be the left or right child of either 31 or 93. Since 94 is greater than 70 and 93, it becomes the right child of 93. Similarly 14 is less than 70 and 31, so it becomes the left child of 31. 23 is also less than 31, so it must be in the left subtree of 31. However, it is greater than 14, so it becomes the right child of 14.

To implement the binary search tree, we will use the nodes and references approach similar to the one we used to implement the linked list, and the expression tree. However, because we must be able create and work with a binary search tree that is empty, our implementation will use two classes. The first class we will call `BinarySearchTree`, and the second class we will call `TreeNode`.

The `BinarySearchTree` Class

The `BinarySearchTree` class has a reference to the `TreeNode` that is the root of the binary search tree. In most cases the external methods defined in the outer class simply check to see if the tree is empty. If there are nodes in the tree, the request is just passed on to a private method defined in the `BinarySearchTree` class that takes the root as a parameter. In the case where the tree is empty or we want to delete the key at the root of the tree, we must take special action. The code for the `BinarySearchTree` class constructor along with a few other miscellaneous functions is shown in Listing 1.

Listing 1

```
class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()
```

The `TreeNode` Class

The `TreeNode` class provides many helper functions that make the work done in the `BinarySearchTree` class methods much easier. The constructor for a `TreeNode`, along with these helper functions, is shown in Listing 2. As you can see in the listing many of these helper functions help to classify a node according to its own position as a child, (left or right) and the kind of children the node has. The `TreeNode` class will also explicitly keep track of the parent as an attribute of each node. You will see why this is important when we discuss the implementation for the `del` operator.

Another interesting aspect of the implementation of `TreeNode` in Listing 2 is that we use Python's optional parameters. Optional parameters make it easy for us to create a `TreeNode` under several different circumstances. Sometimes we will want to construct a new `TreeNode` that already has both a `parent` and a `child`. With an existing parent and child, we can pass parent and child as parameters. At other times we will just create a `TreeNode` with the key value pair, and we will not pass any parameters for `parent` or `child`. In this case, the default values of the optional parameters are used.

Listing 2

```
class TreeNode:
   def __init__(self,key,val,left=None,right=None,
                                       parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and self.parent.rightChild == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
        return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self,key,value,lc,rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self
```

The `put` Method

Now that we have the `BinarySearchTree` shell and the `TreeNode` it is time to write the `put` method that will allow us to build our binary search tree. The `put` method is a method of the `BinarySearchTree` class. This method will check to see if the tree already has a root. If there is not a root then `put` will create a new `TreeNode` and install it as the root of the tree. If a root node is already in place then `put` calls the private, recursive, helper function `_put` to search the tree according to the following algorithm:

- Starting at the root of the tree, search the binary tree comparing the new key to the key in the current node. If the new key is less than the current node, search the left subtree. If the new key is greater than the current node, search the right subtree.
- When there is no left (or right) child to search, we have found the position in the tree where the new node should be installed.
- To add a node to the tree, create a new `TreeNode` object and insert the object at the point discovered in the previous step.

Listing 3 shows the Python code for inserting a new node in the tree. The `_put` function is written recursively following the steps outlined above. Notice that when a new child is inserted into the tree, the `currentNode` is passed to the new tree as the parent.

One important problem with our implementation of insert is that duplicate keys are not handled properly. As our tree is implemented a duplicate key will create a new node with the same key value in the right subtree of the node having the original key. The result of this is that the node with the new key will never be found during a search. A better way to handle the insertion of a duplicate key is for the value associated with the new key to replace the old value. We leave fixing this bug as an exercise for you.

Listing 3

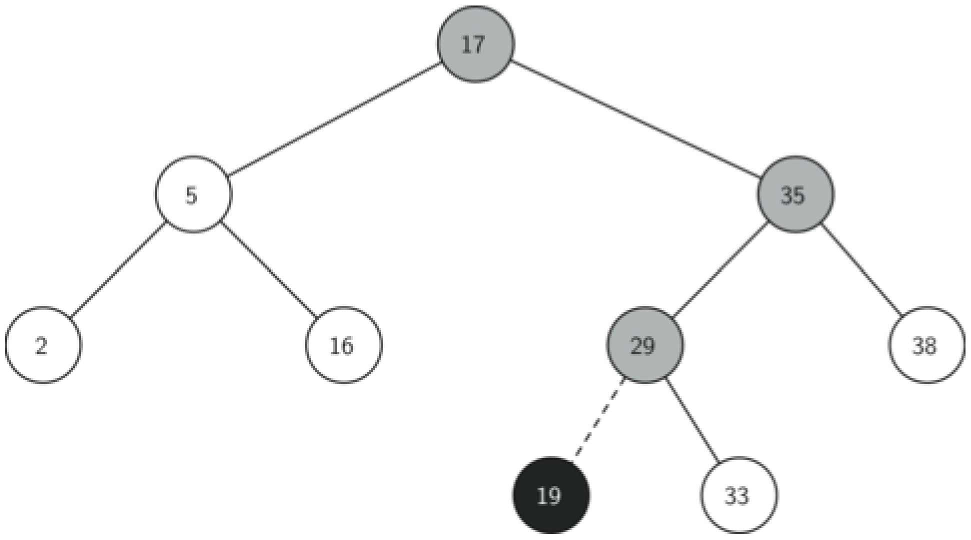
```
def put(self,key,val):
    if self.root:
        self._put(key,val,self.root)
    else:
        self.root = TreeNode(key,val)
    self.size = self.size + 1

def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
               self._put(key,val,currentNode.leftChild)
        else:
               currentNode.leftChild = TreeNode(key,val,parent=currentNode)
    else:
        if currentNode.hasRightChild():
               self._put(key,val,currentNode.rightChild)
        else:
               currentNode.rightChild = TreeNode(key,val,parent=currentNode)
```

With the `put` method defined, we can easily overload the `[]` operator for assignment by having the `__setitem__` method call (as seen blow) the put method. This allows us to write Python statements like `myZipTree['Plymouth'] = 55446`, just like a Python dictionary.

Listing 4

```
def __setitem__(self,k,v):
    self.put(k,v)
```

The figure below illustrates the process for inserting a new node into a binary search tree. The lightly shaded nodes indicate the nodes that were visited during the insertion process.



Source: Problem Solving and Algorithms in Python (http://interactivepython.org/runestone/static/pythonds/index.html#) from Bradley Miller on www.interactivepython.org (http://interactivepython.org/runestone/static/pythonds/index.html#).