IMPLEMENTATION OF BINARY SEARCH TREES

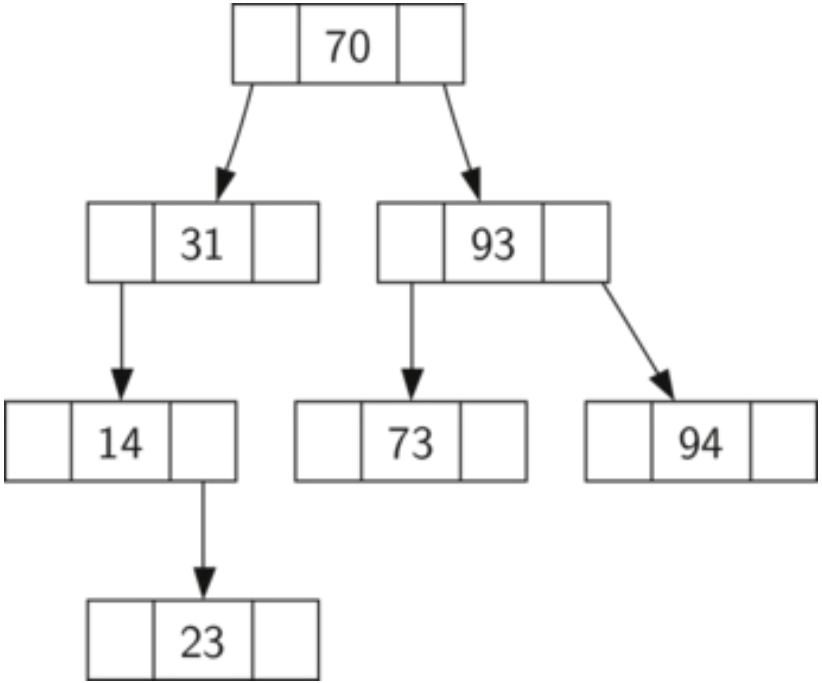
Implementation of Binary Search Trees

- 2 – Part Lecture
- Refer to Jupyter Notebook for Code
- Refer to Wikipedia article on BST for more information
- Overview of subject

Binary Search Trees

- 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.

Binary Search Trees

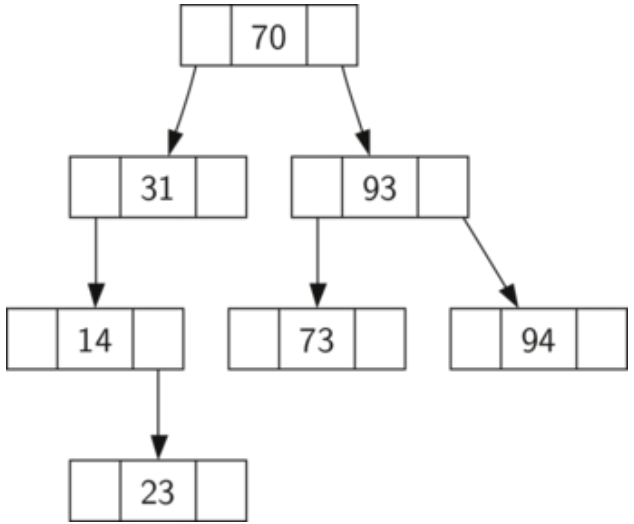


Binary Search Trees

- 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.

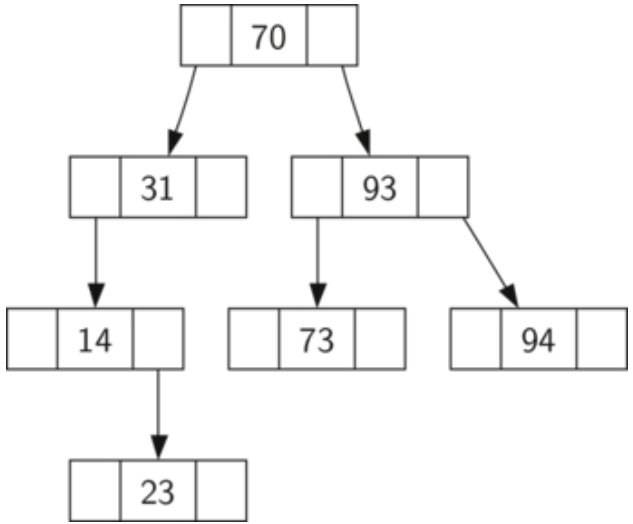
Binary Search Trees

- Now that you know what a binary search tree is, we will look at how a binary search tree is constructed.
- The search tree in the figure represents the nodes that exist after we have inserted the following keys in the order shown: 70,31,93,94,14,23,73



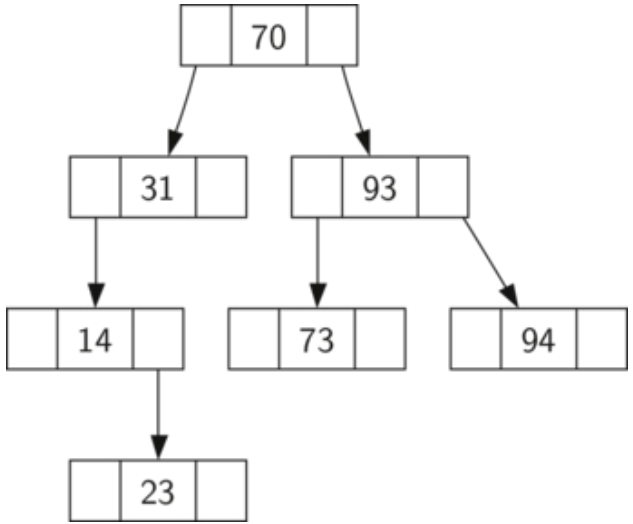
Binary Search Trees

- 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.



Binary Search Trees

- 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.



Binary Search Trees

- 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**.

Binary Search Trees

- 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

Binary Search Trees

```
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__()
```

Binary Search Trees

- Let’s jump to the Notebook to check out the full **TreeNode** class!

Binary Search Trees

- 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.

Binary Search Trees

- 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.

Binary Search Trees

- 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…

Binary Search Trees

- 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.

Binary Search Trees

- 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.

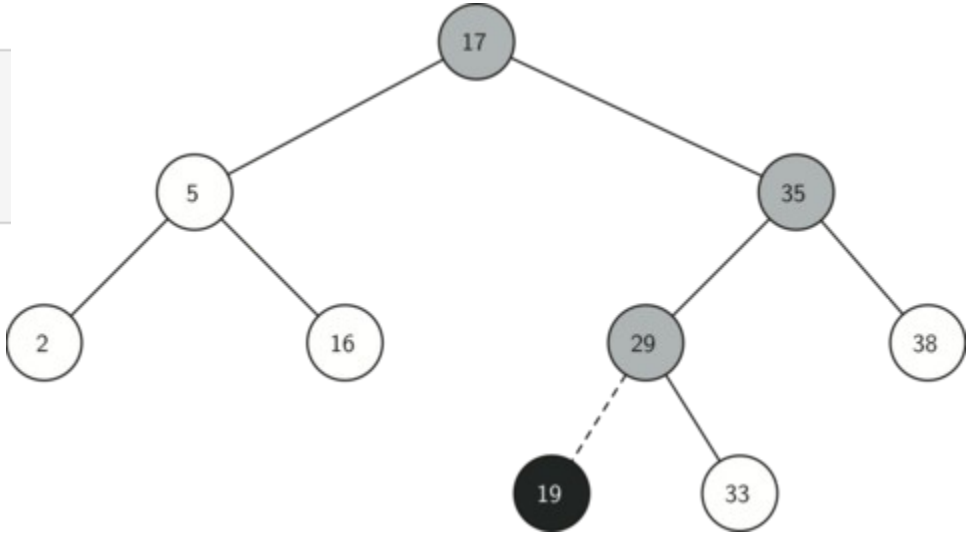
Binary Search Trees

```
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)
```

Binary Search Trees

- Inserting a Node with key= 19

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



Binary Search Trees

- Once the tree is constructed, the next task is to implement the retrieval of a value for a given key.
- The get method is even easier than the put method because it simply searches the tree recursively until it gets to a non-matching leaf node or finds a matching key.
- When a matching key is found, the value stored in the payload of the node is returned.

Binary Search Trees

```
def get(self,key):
    if self.root:
        res = self._get(key,self.root)
        if res:
                return res.payload
        else:
                return None
    else:
        return None

def _get(self,key,currentNode):
    if not currentNode:
        return None
    elif currentNode.key == key:
        return currentNode
    elif key < currentNode.key:
        return self._get(key,currentNode.leftChild)
    else:
        return self._get(key,currentNode.rightChild)

def __getitem__(self,key):
    return self.get(key)
```

Binary Search Trees

- Using get, we can implement the in operation by writing a **__contains__** method for the **BinarySearchTree**.
- The **__contains__** method will simply call get and return True if get returns a value, or False if it

```
def __contains__(self,key):
    if self._get(key,self.root):
        return True
    else:
        return False
```

Deleting Nodes

- Finally, we turn our attention to the most challenging method in the binary search tree, the deletion of a key.
- The first task is to find the node to delete by searching the tree.
- If the tree has more than one node we search using the **_get** method to find the **TreeNode** that needs to be removed.

Deleting Nodes

- If the tree only has a single node, that means we are removing the root of the tree, but we still must check to make sure the key of the root matches the key that is to be deleted.
- In either case if the key is not found the del operator raises an error.

Deleting Nodes

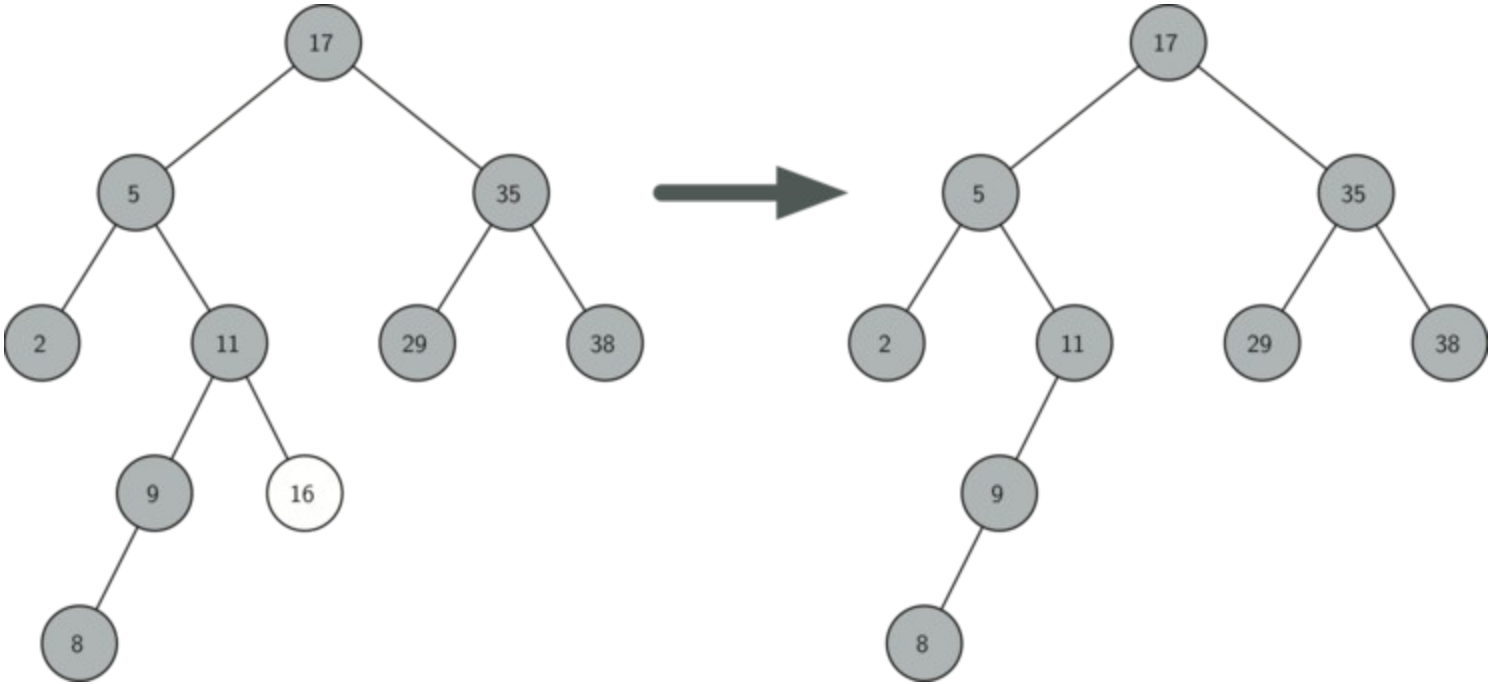
```
def delete(self,key):
   if self.size > 1:
      nodeToRemove = self._get(key,self.root)
      if nodeToRemove:
          self.remove(nodeToRemove)
          self.size = self.size-1
      else:
          raise KeyError('Error, key not in tree')
   elif self.size == 1 and self.root.key == key:
      self.root = None
      self.size = self.size - 1
   else:
      raise KeyError('Error, key not in tree')

def __delitem__(self,key):
    self.delete(key)
```

Deleting Nodes

- Once we've found the node containing the key we want to delete, there are three cases that we must consider:

Deleting Nodes Case 1

- The node to be deleted has no children



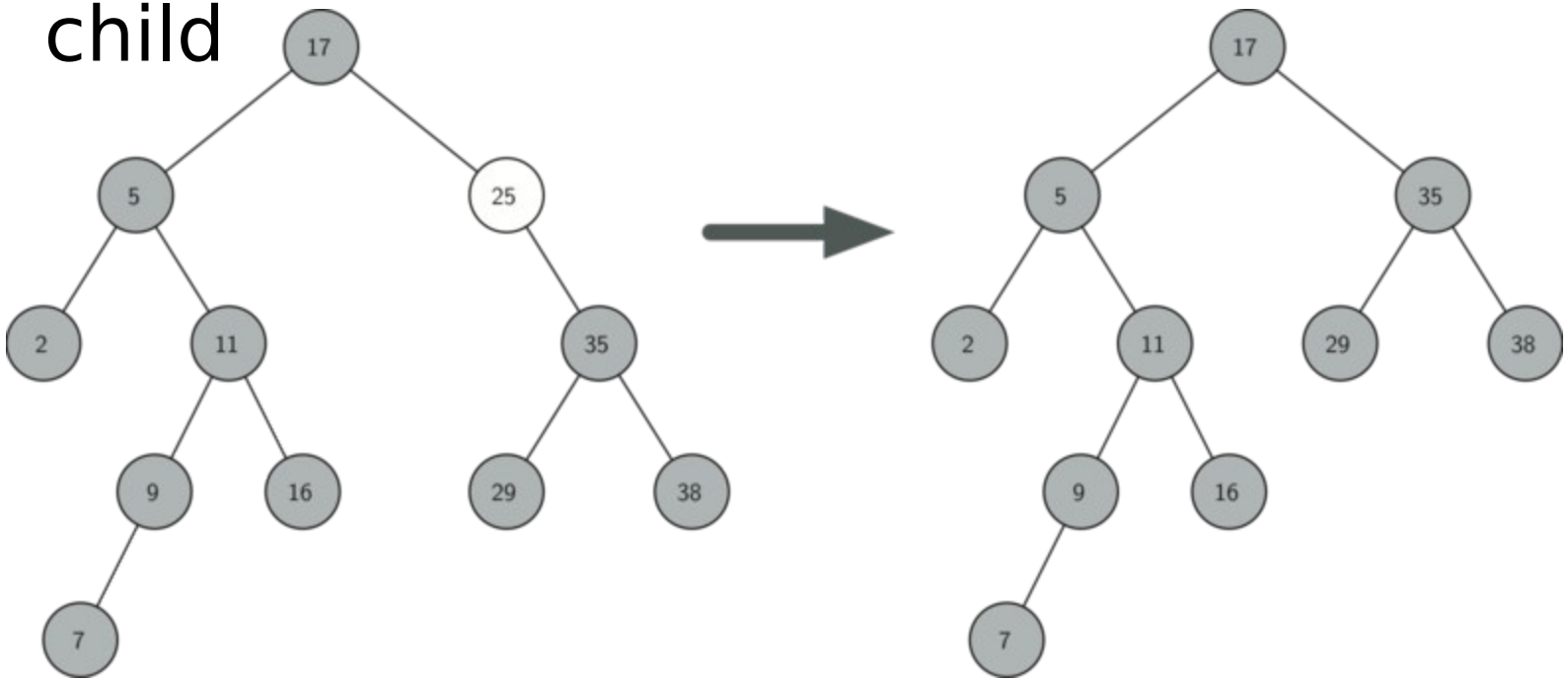
Deleting Nodes Case 1

- The first case is straightforward
- If the current node has no children all we need to do is delete the node and remove the reference to this node in the p

```
if currentNode.isLeaf():
    if currentNode == currentNode.parent.leftChild:
        currentNode.parent.leftChild = None
    else:
        currentNode.parent.rightChild = None
```

Deleting Nodes Case 2

- The node to be deleted has only one child



Deleting Nodes Case 2

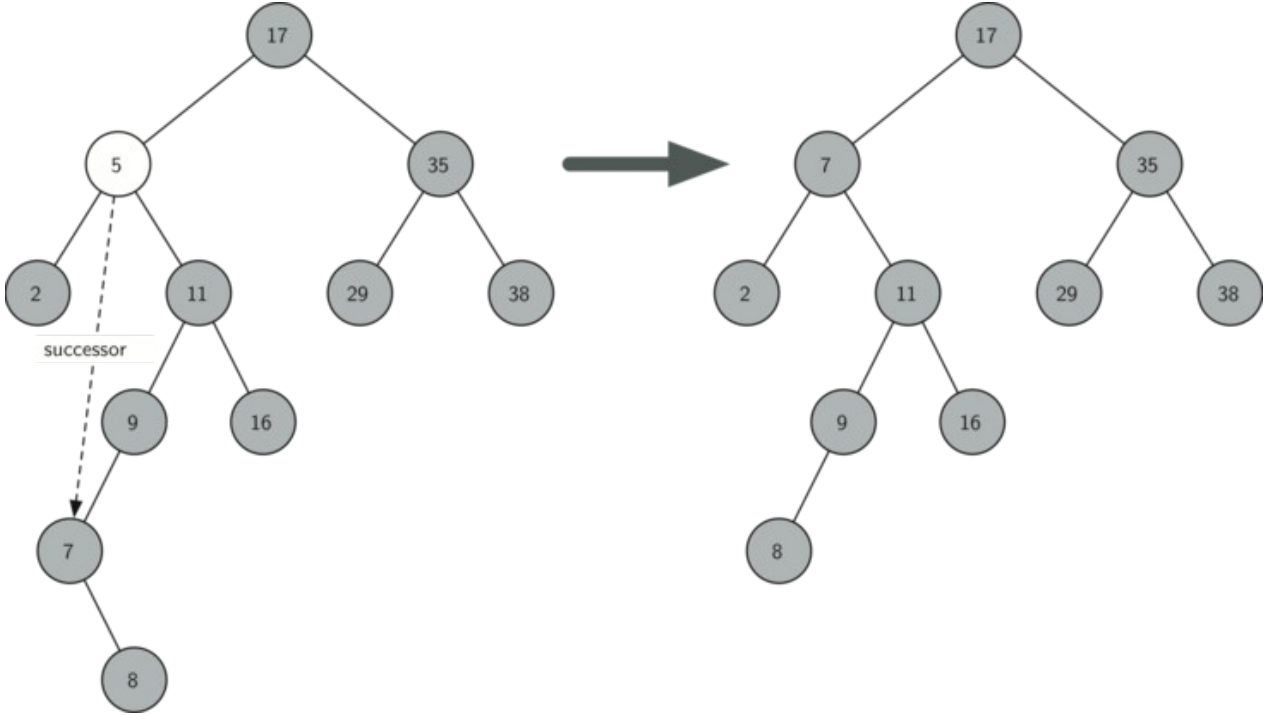
- The second case is only slightly more complicated.
- If a node has only a single child, then we can simply promote the child to take the place of its parent.

Deleting Nodes Case 2

- Let’s jump to the code to discuss this second case!

Deleting Nodes Case 3

- The node to be deleted has two children



Deleting Nodes Case 3

- The third case is the most difficult case to handle
- If a node has two children, then it is unlikely that we can simply promote one of them to take the node’s place.
- We can, however, search the tree for a node that can be used to replace the one scheduled for deletion.

Deleting Nodes Case 3

- What we need is a node that will preserve the binary search tree relationships for both of the existing left and right subtrees.
- The node that will do this is the node that has the next-largest key in the tree. We call this node the **successor**, and we will look at a way to find the successor shortly.

Deleting Nodes Case 3

- The successor is guaranteed to have no more than one child, so we know how to remove it using the two cases for deletion that we have already implemented.
- Once the successor has been removed, we simply put it in the tree in place of the node to be deleted.

Deleting Nodes Case 3

- Notice that we make use of the helper methods **findSuccessor** and **findMin** to find the successor.
- To remove the successor, we make use of the method **spliceOut**.
- The reason we use **spliceOut** is that it goes directly to the node we want to splice out and makes the right changes.

Deleting Nodes Case 3

```
elif currentNode.hasBothChildren(): #interior
        succ = currentNode.findSuccessor()
        succ.spliceOut()
        currentNode.key = succ.key
        currentNode.payload = succ.payload
```

findSuccessor

- Let’s jump to the notebook again to discuss the **findSuccessor** method.

One last method!

- We need to look at one last interface method for the binary search tree.
- Suppose that we would like to simply iterate over all the keys in the tree in order.
- This is definitely something we have done with dictionaries, so why not trees?

One last method!

- You already know how to traverse a binary tree in order, using the **inorder** traversal algorithm.
- However, writing an iterator requires a bit more work, since an iterator should return only one node each time the iterator is called.

One last method!

- The __iter__ method

```
def __iter__(self):
   if self:
      if self.hasLeftChild():
            for elem in self.leftChild:
               yield elem
      yield self.key
      if self.hasRightChild():
            for elem in self.rightChild:
               yield elem
```

Review

- We’ve covered a lot of ground! Review the Jupyter Notebook and review Wikipedia article on Binary Search Trees!
- Up next…Interview Problems!