NANYANG TECHNOLOGICAL UNIVERSITY

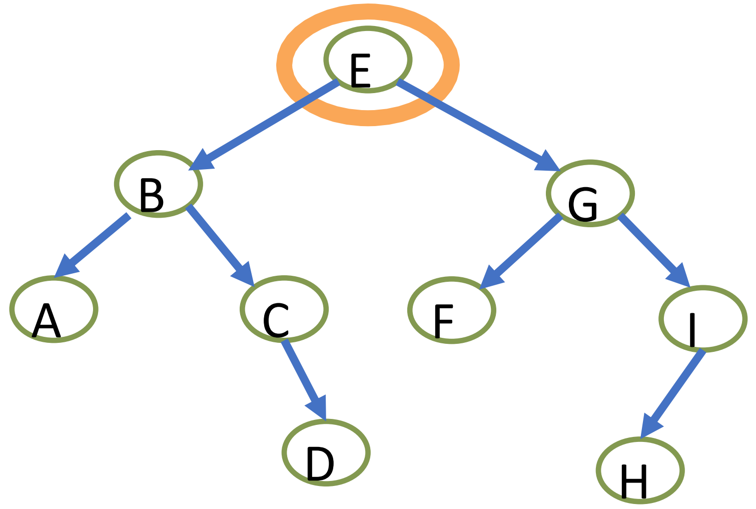
Data Structures & Algorithms in Python: Binary Search Trees

Dr. Owen Noel Newton Fernando

College of Computing and Data Science

Pre-order, In-order and Post-order Depth First Traversal

- **Pre-order**
  - **Process the current node's data**
  - Visit the left child subtree
  - Visit the right child subtree
- **In-order**
  - Visit the left child subtree
  - **Process the current node's data**
  - Visit the right child subtree
- **Post-order**
  - Visit the left child subtree
  - Visit the right child subtree
  - **Process the current node's data**



Summary of Depth First Traversal

Pre-Order Traversal

E B A C D G F I H

```
def pre_order_traversal(self, node):
        if node:
            print(node.data, end=" ")
            self.pre_order_traversal(node.left)
            self.pre_order_traversal(node.right)
```

In-Order Traversal

A B C D E F G H I

```
def in_order_traversal(self, node):
    if node:
        self.in_order_traversal(node.left)
        print(node.data, end=" ")
        self.in_order_traversal(node.right)
```

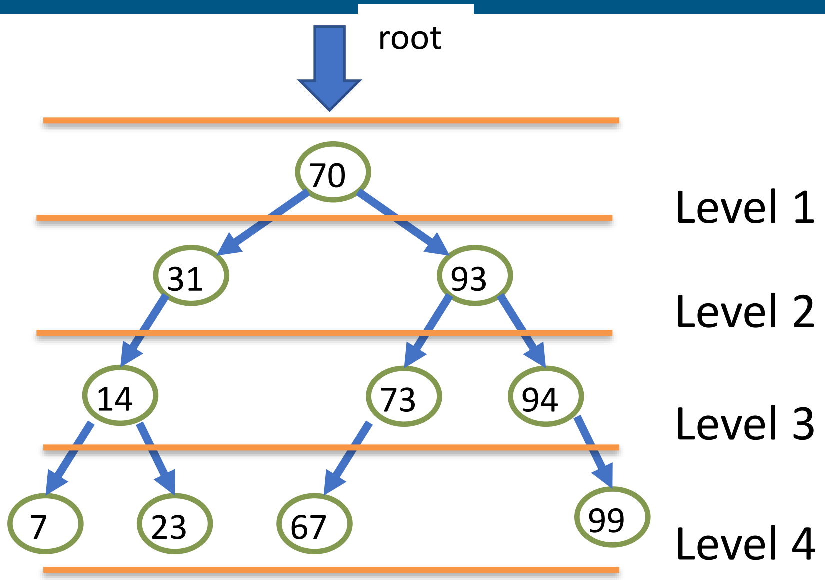
Post-Order Traversal

A D C B F H I G E

```
def post_order_traversal(self, node):
    if node:
        self.post_order_traversal(node.left)
        self.post_order_traversal(node.right)
        print(node.data, end=" ")
```

Breath-first Traversal: Level-by-level

1. **Initialize the queue:**
   - Create an instance of the Queue class.
   - Add the root node to the queue using **queue.enqueue(self.root).**
2. **Start the traversal loop:**
   - While the queue is not empty:
     - Dequeue the front node using **queue.dequeue()** and print its data.
     - Enqueue the left child if it exists.
     - Enqueue the right child if it exists.
3. **Finish traversal:**
   - Repeat the process until the queue is empty, ensuring level-by-level traversal.



```
def level_order_traversal(self):
    if not self.root:
        return

    queue = Queue()
    queue.enqueue(self.root)

    while not queue.is_empty():
        current_node = queue.dequeue()
        print(current_node.data, end=" ")

        if current_node.left:
                queue.enqueue(current_node.left)
        if current_node.right:
                queue.enqueue(current_node.right)
```

Count Nodes in a Binary Tree (SIZE)

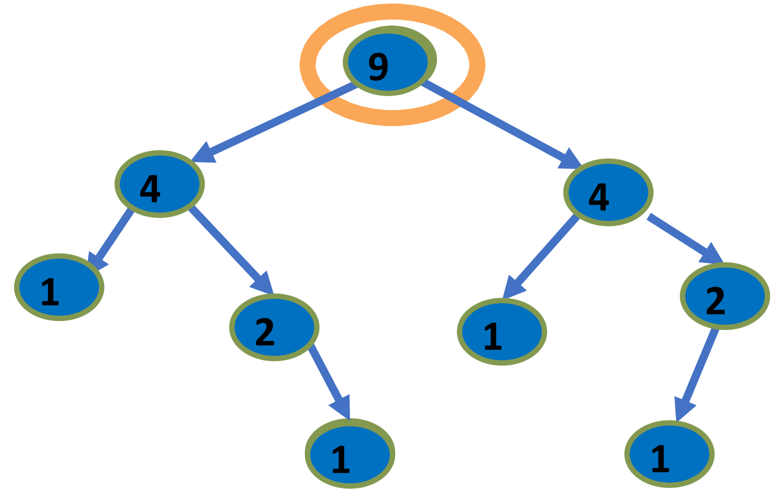
- Recursive definition:
  - Number of nodes in a tree

    = 1

      + number of nodes in left subtree

      + number of nodes in right subtree

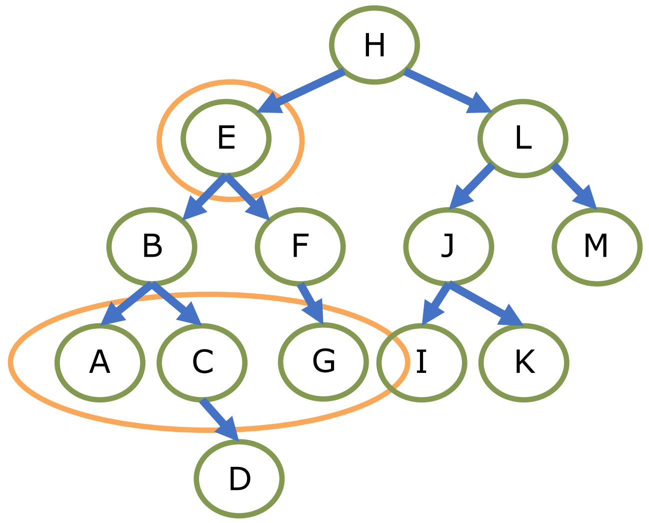
- Each node returns the number of nodes in its subtree



```
    def count_nodes(self, node):
        if not node:
            return 0
        return 1 + self.count_nodes(node.left) + self.count_nodes(node.right)
```

Find the Kth-level Descendant Nodes

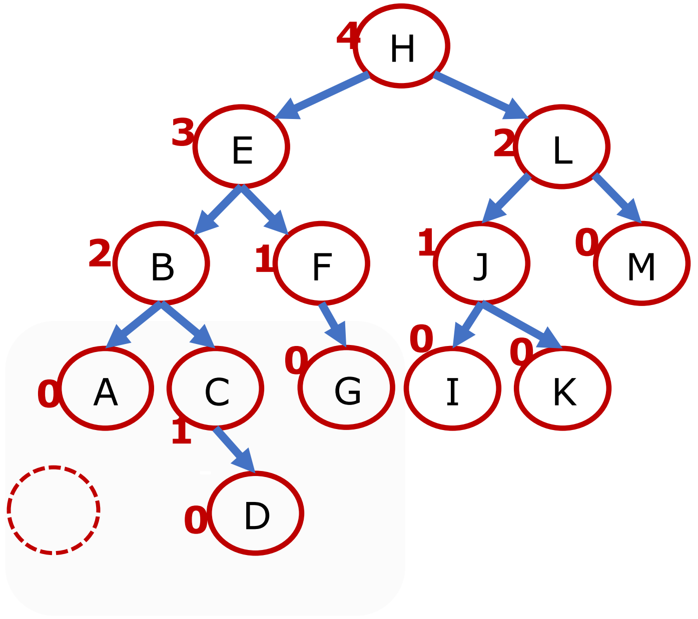
- Given a node X, find all the descendants of X
- Given node E and K=2, we should return its grandchild nodes A, C, and G
- What if we want to find Kth-level descendants?
  - **Need a way to keep track of how many levels down we've gone**



```
    def kth_level_descendants(self, node, k):
        if node is None:
            return []

        if k == 0:
            return [node.data]

        left_descendants = self.kth_level_descendants(node.left, k - 1)
        right_descendants = self.kth_level_descendants(node.right, k - 1)

        return left_descendants + right_descendants
…

print(bt.kth_level_descendants(bt.root, 2))
```

Height of A Node in A Binary Tree

- The height of a tree: The number of edges on the longest path from the root to a leaf
  - Leaf node returns 0
  - **None node (empty) returns -1**



```
def calculate_height(self, node):
        if node is None:
            return -1

        left_height = self.calculate_height(node.left)
        right_height = self.calculate_height(node.right)

        return 1 + max(left_height, right_height)
```

Other traversals

- The other traversals are the reverse of these three standard ones
  - That is, the right subtree is traversed before the left subtree is traversed
- Reverse preorder: root, right subtree, left subtree
- Reverse inorder: right subtree, root, left subtree
- Reverse postorder: right subtree, left subtree, root
- These are rarely used or talked about (although the reverse inorder traversal is convenient for printing trees)

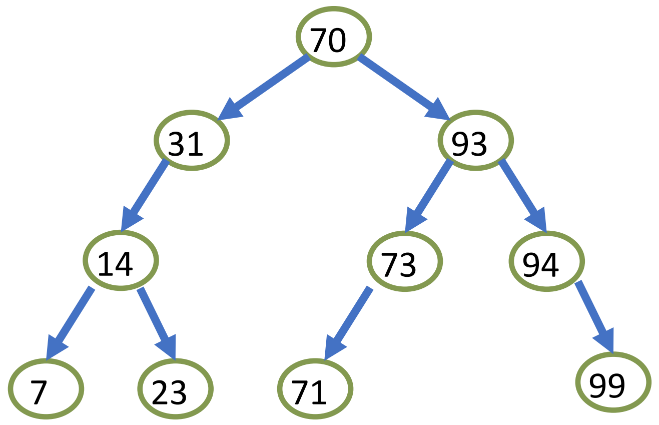
Node Insertion on A Binary Tree

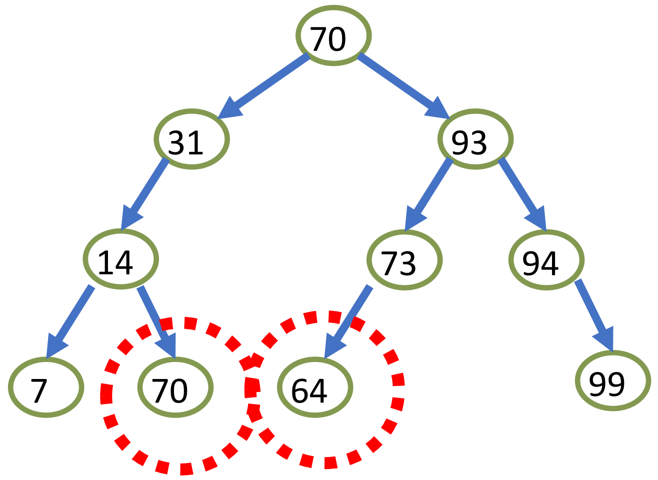
- The first node is inserted as a root node
- There is no rule to insert the subsequence nodes into a binary tree
- We will insert nodes to structure the tree for use as a binary search tree.

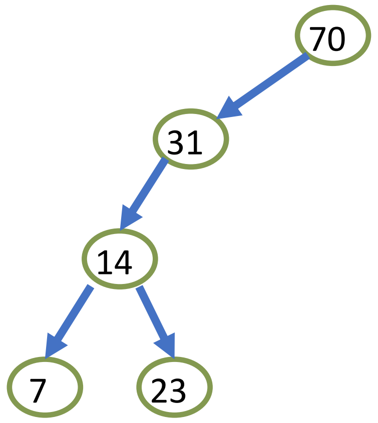
```
def insert(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert(data, self.root)
```

Binary Search Tree

- If the given binary tree is a binary search tree (BST), then each node in the tree must satisfy the following properties:
  - Node's value is greater than all values in its left subtree.
  - Node's value is less than all values in its right subtree.
  - Both subtrees of the node are also binary search trees.



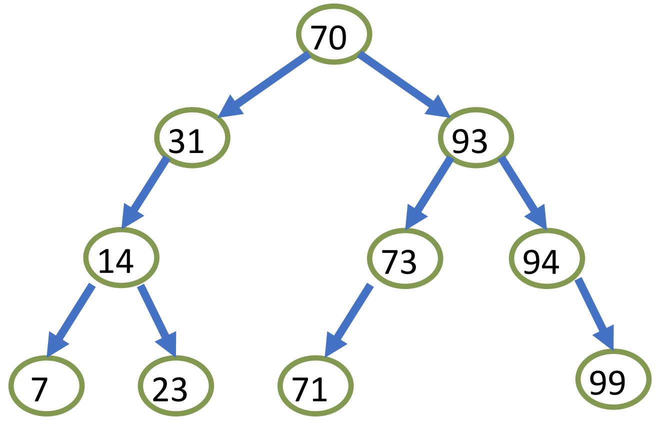




Binary Search Tree: Searching

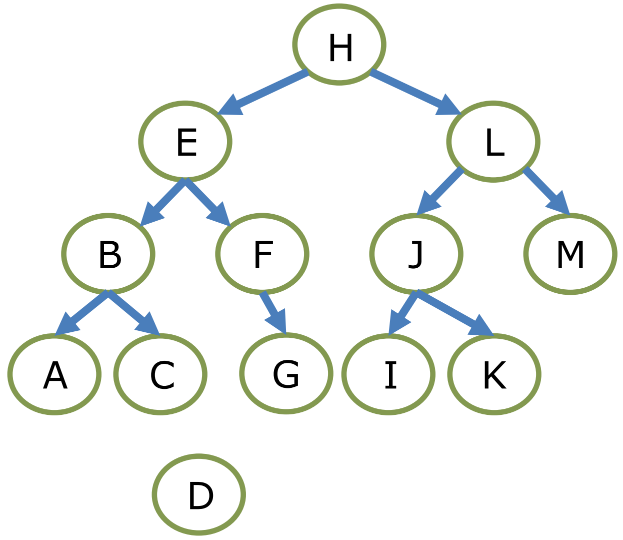
- The approach is a decrease-and-conquer approach
- A problem is divided into two smaller and similar sub-problems, one of which does not even have to be solved
- This approach uses the information of the order to reduce the search space.

```
def _search(self, data, current_node):
        if current_node is None:
            return False
        elif data == current_node.data:
            return True
        elif data < current_node.data:
            return self._search(data, current_node.left)
        else:
            return self._search(data, current_node.right)
```



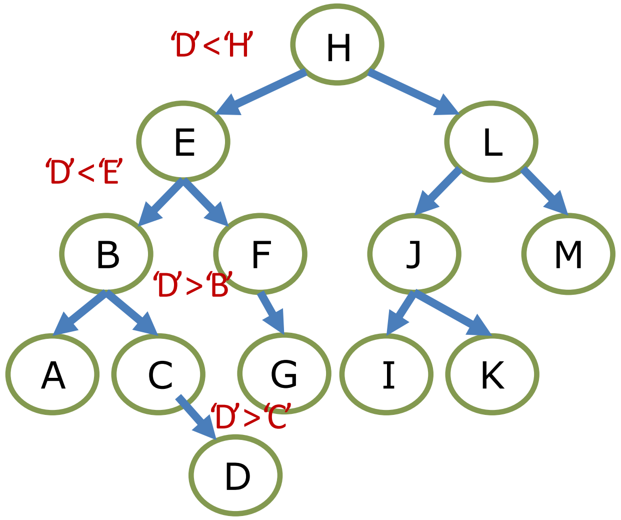
Binary Search Tree: Insertion

- After insert a node, the BST must remain as a BST
- A duplicate node is not allowed for insertion
- A unique position of the given BST will be given to the node



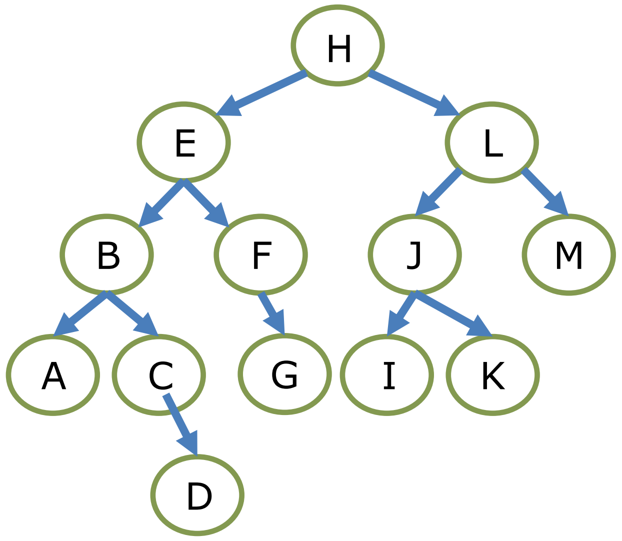
Binary Search Tree: Insertion

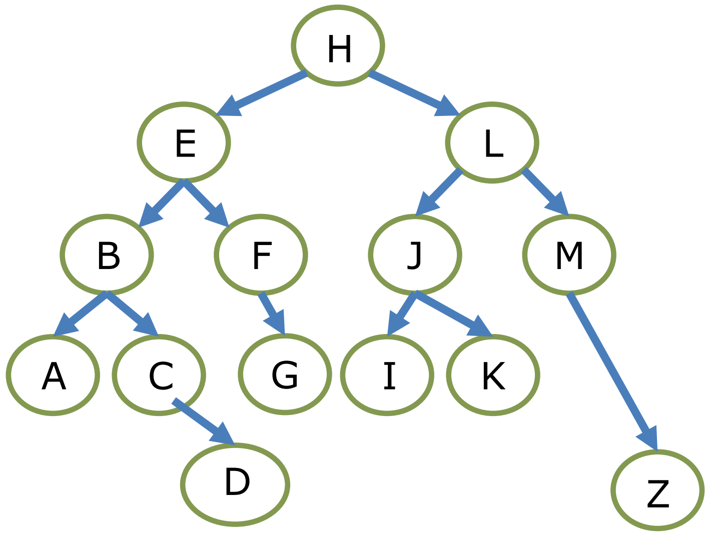
- After insert a node, the BST must remain as a BST
- A duplicate node is not allowed for insertion
- A unique position of the given BST will be given to the node



Binary Search Tree: Insertion

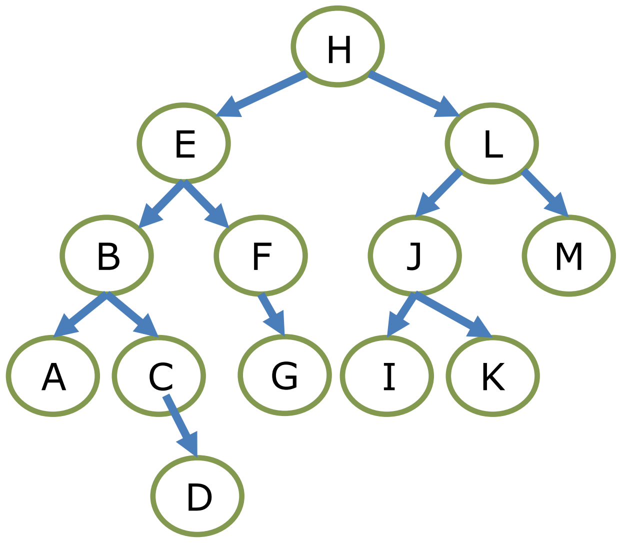
- After insert a node, the BST must remain as a BST
- A duplicate node is not allowed for insertion
- A unique position of the given BST will be given to the node

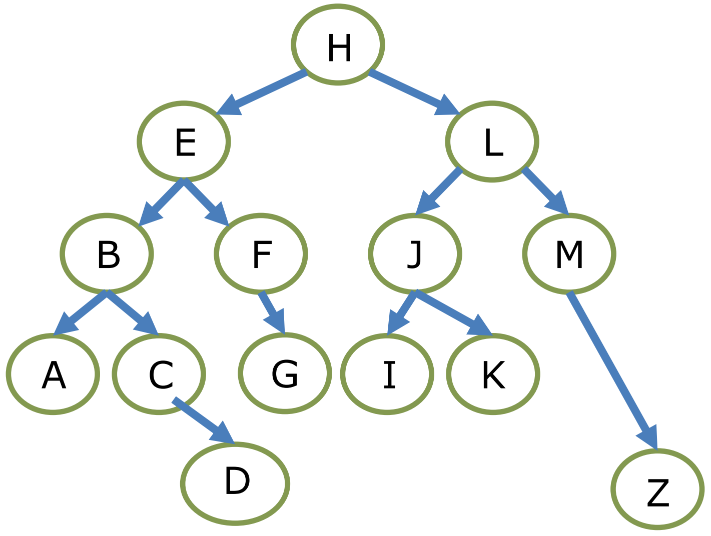


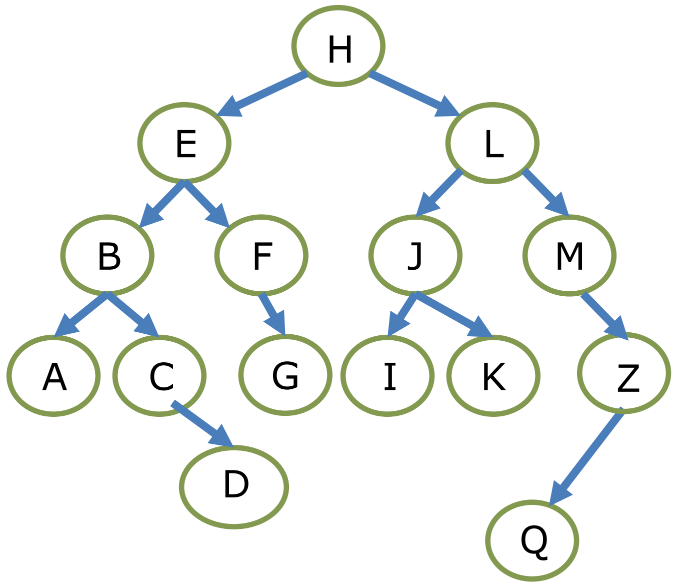


Binary Search Tree: Insertion

- After insert a node, the BST must remain as a BST
- A duplicate node is not allowed for insertion
- A unique position of the given BST will be given to the node



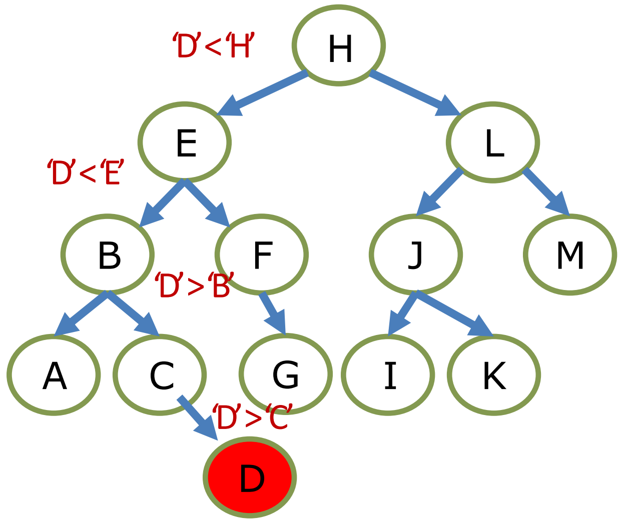




Binary Search Tree: Insertion

- After insert a node, the BST must remain as a BST
- A duplicate node is not allowed for insertion
- A unique position of the given BST will be given to the node

```
def _insert(self, data, current_node):
        if data < current_node.data:
            if current_node.left is None:
                current_node.left = Node(data)
            else:
                self._insert(data, current_node.left)
        else:
            if current_node.right is None:
                current_node.right = Node(data)
            else:
                self._insert(data, current_node.right)
```

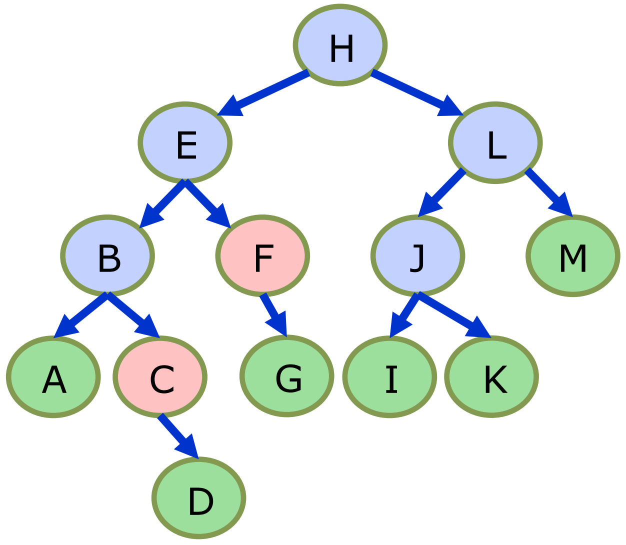


Deleting nodes in binary search trees

- Deleting nodes is the most complicated of the four basic operations
- After remove a node X, the BST must remain as a BST
- The procedure for deleting the node depends on its children:
  1. **X** has no children
  2. **X** has one child
  3. **X** has two children

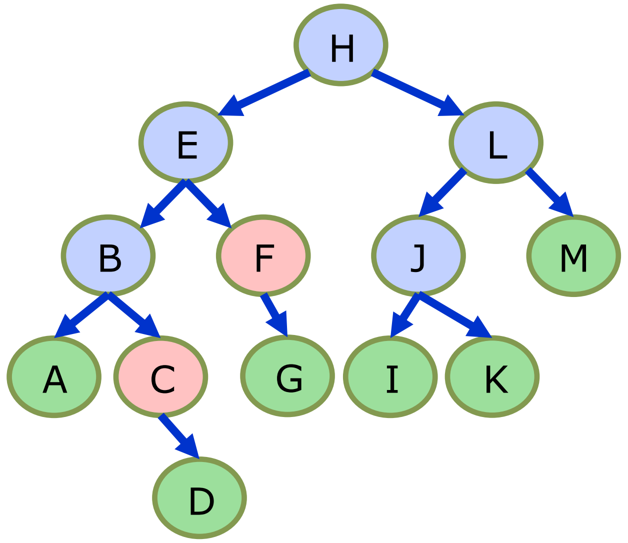
Deleting nodes in binary search trees

- Deleting nodes is the most complicated of the four basic operations
- After remove a node X, the BST must remain as a BST
- The procedure for deleting the node depends on its children:
    1. X has no children
    2. X has one child
    3. X has two children



Deleting nodes in binary search trees

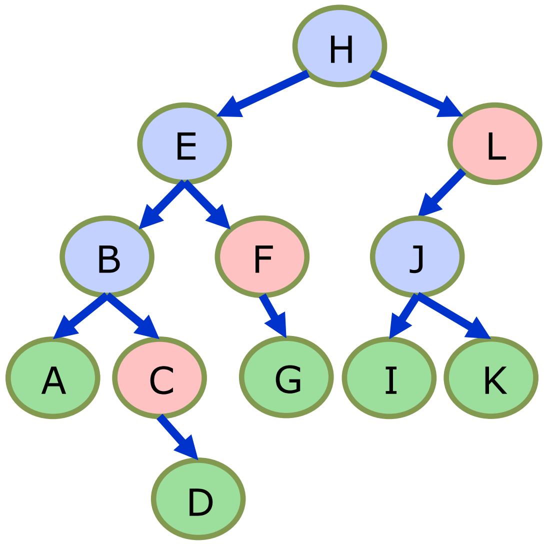
- Deleting nodes is the most complicated of the four basic operations
- After remove a node X, the BST must remain as a BST
- The procedure for deleting the node depends on its children:
  1. X has no children
     - Remove X



Binary Search Tree: Deletion

- Deleting nodes is the most complicated of the four basic operations
- After remove a node X, the BST must remain as a BST
- The procedure for deleting the node depends on its children:

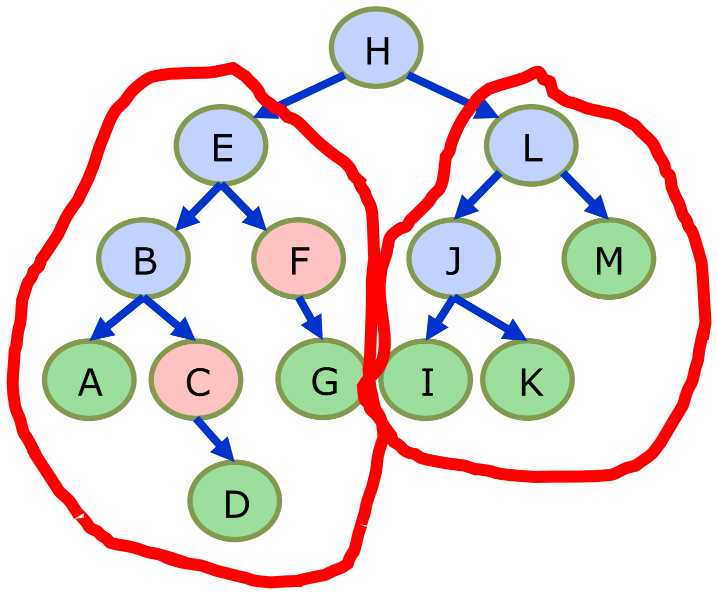
2. X has one child
    - Replace X with Y
    - Remove X



Binary Search Tree: Deletion

- Deleting nodes is the most complicated of the four basic operations
- After remove a node X, the BST must remain as a BST
- The procedure for deleting the node depends on its children:

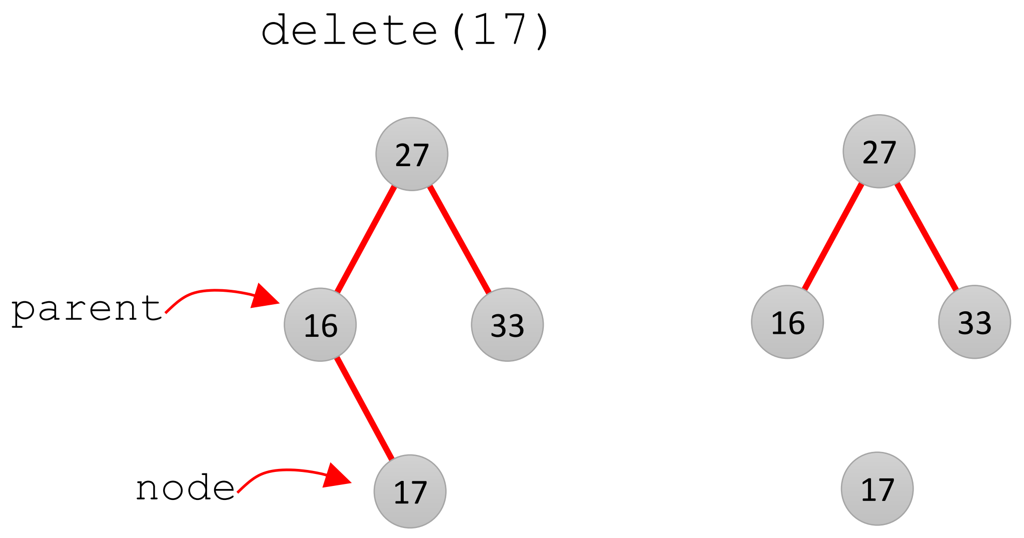
3. X has two children
   - Swap x with successor
     - the (largest) rightmost node in left subtree
     - the (smallest) leftmost node in right subtree
   - Perform case 1 or 2 to remove it



Deletion Case 0: Node to Delete is a Leaf

- When the node to delete is a leaf, simply setting its parent's child link to `None` takes it out of the tree

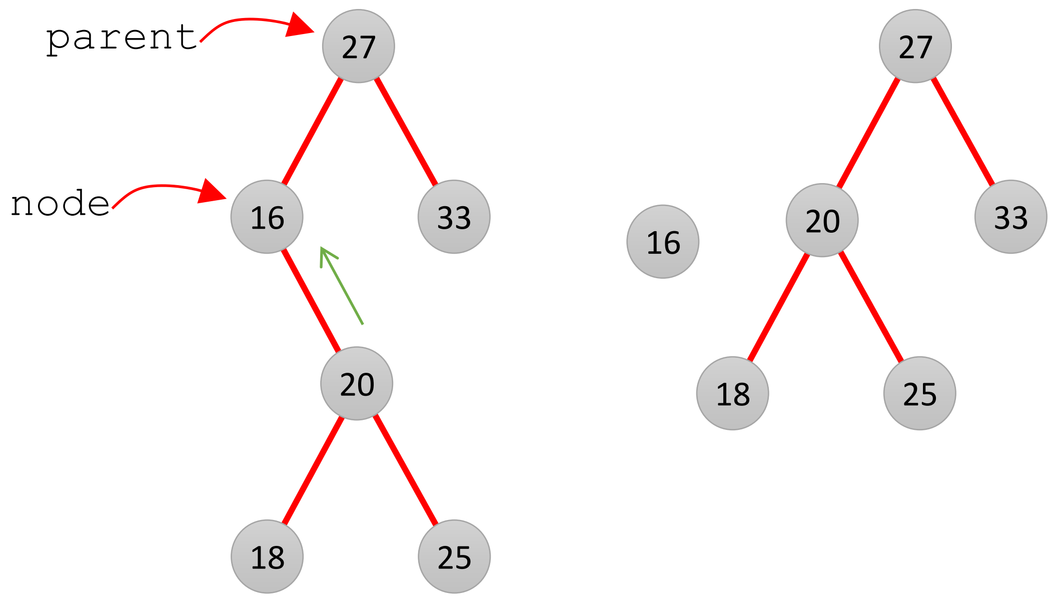
- Removing the node doesn't disturb the order of the node before or after it



Deletion Case 1: Node to Delete has 1 Child

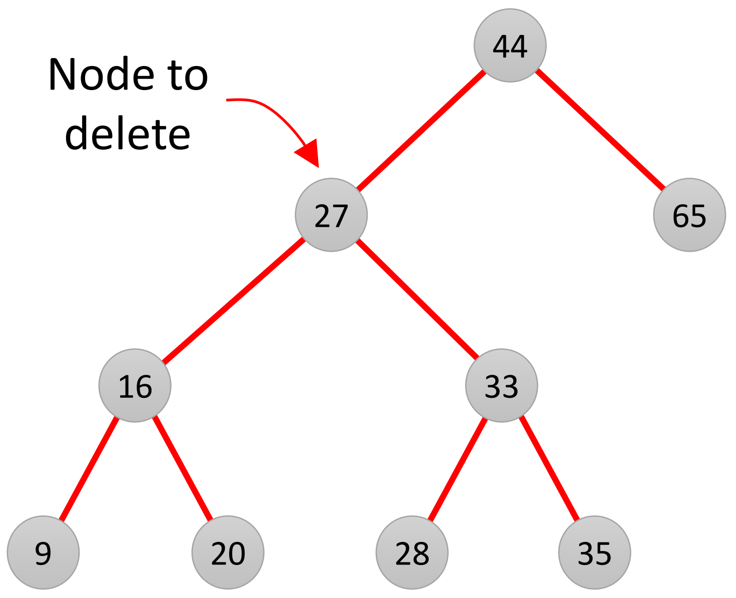
- When the node to delete, N, has a single child, the situation is similar to removing a node from a linked list
- Replacing N's parent's child link with the single child link of N, moves the subtree rooted at N's child into the correct position
- All the nodes remain in proper order for a binary search tree

```
delete(16)
```

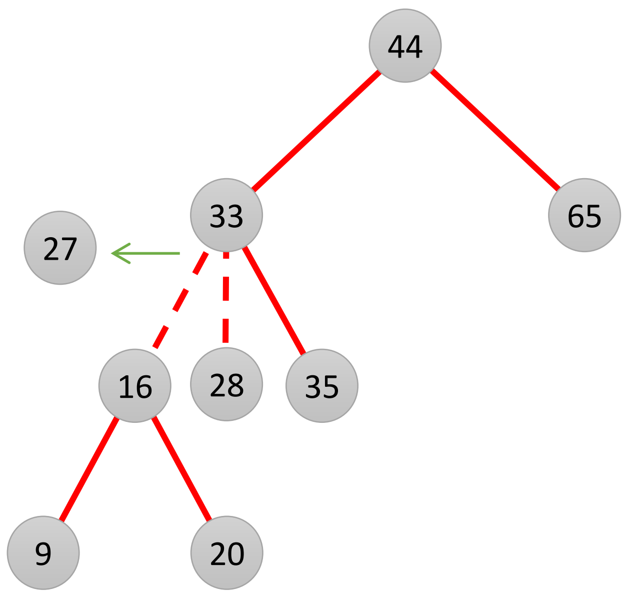


Deletion Case 2: Node to Delete has 2 Children

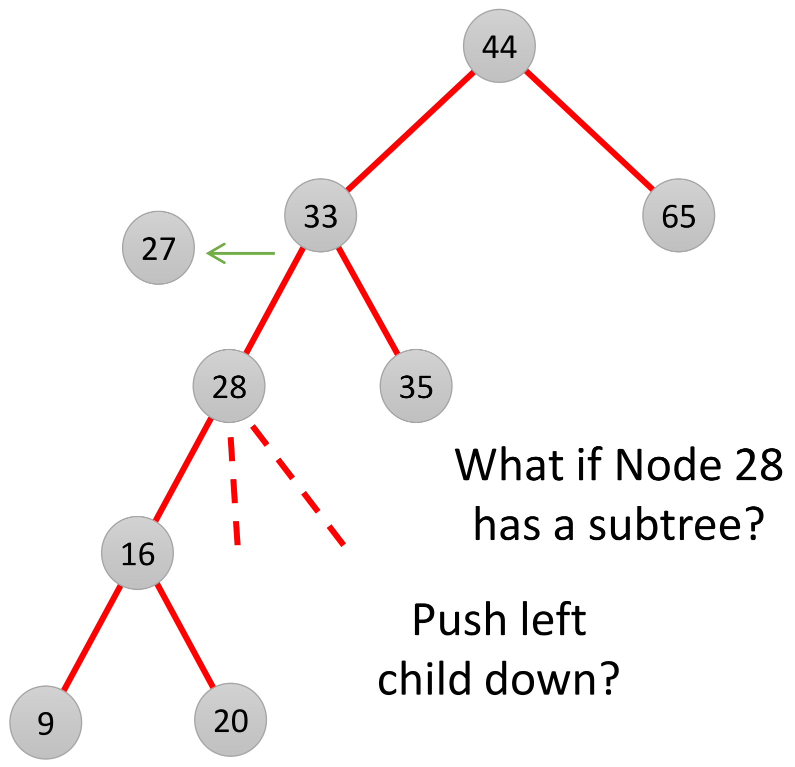
- The most complicated deletion case occurs when the node to be deleted has two children
- Once the node is deleted, how do we rearrange its children to preserve values order?



Original tree



Allow more than 2 children?

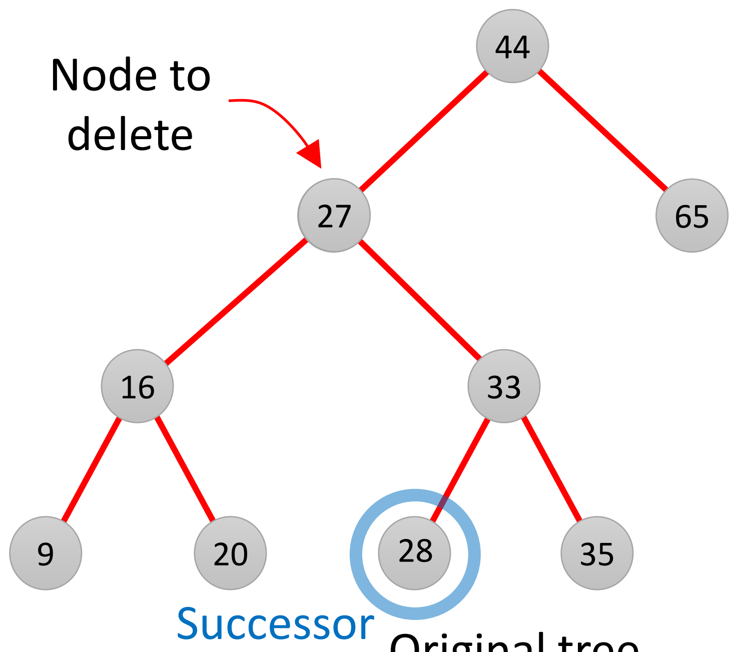


What if Node 28 has a subtree?

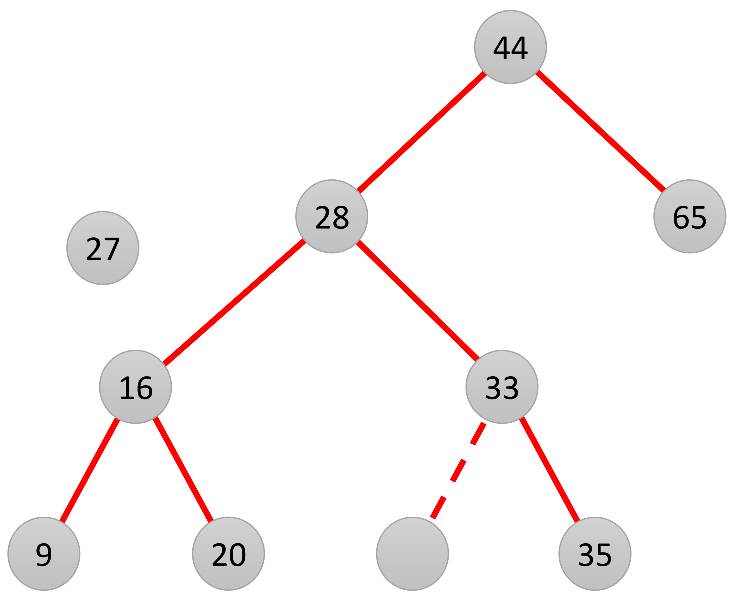
Push left child down?

Deletion Case 2: Find the Successor

- When the node to delete has 2 children, there must be a **successor node**, the next node that would be visited by an **in-order traversal**
- What happens if we place the successor in the node to delete?



Original tree



After moving successor

The values are all in correct order

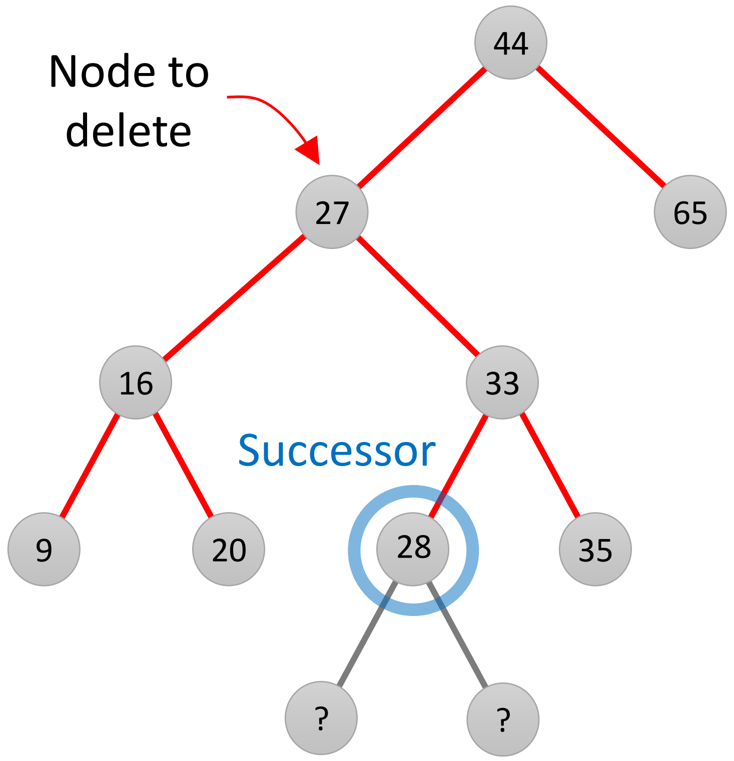
What should be done where the successor was?

Just dropping it seems to work

What if there was a tree rooted at node 28?

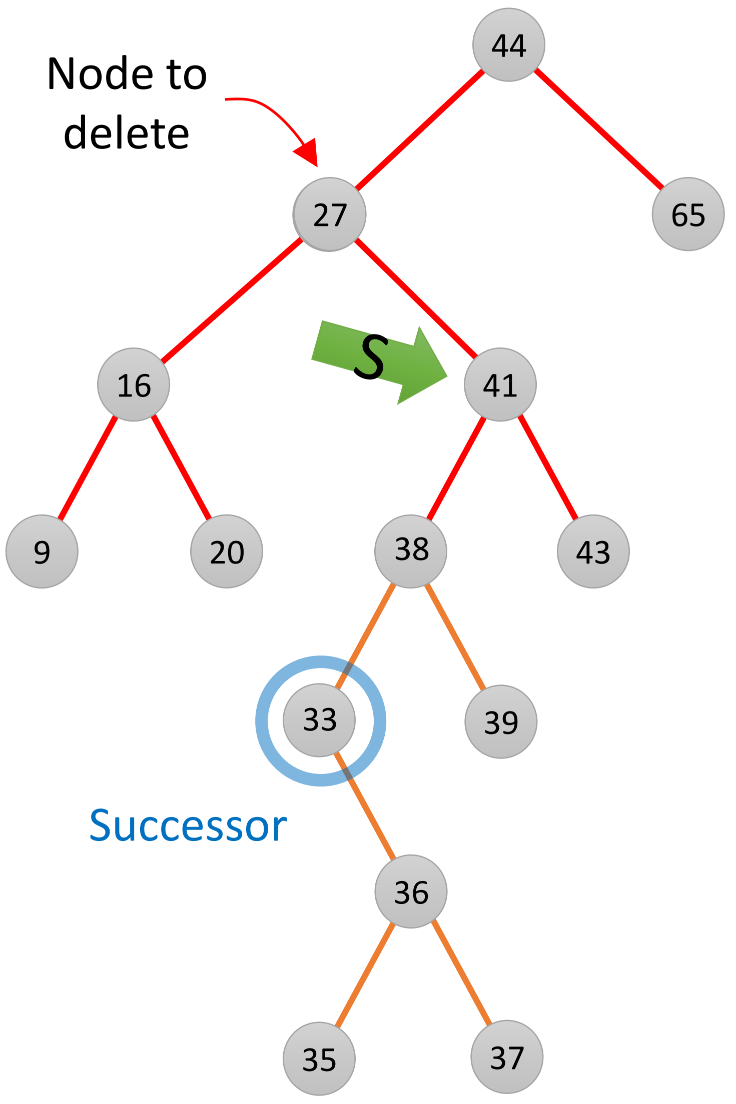
Deletion Case 2: More About the Successor

- If the successor is a leaf, that node can just be dropped to complete the delete operation
- What if the successor is a subtree?
- Well, the successor can have a right child, but it cannot have a left child
- If it had a left child, that subtree would contain the successor!
- So the successor node can always be deleted using the rules for Cases 0 or 1



Algorithm to Find the Successor

- Once we find the node to delete, how do we find the successor?
- Start by setting a pointer, S, to the right child of the node to delete
- While node S has a left child
  - Move S to its left child
- S points to the successor
- Deleting Node 27 causes the subtree at Node 33 to move up



Deletion Algorithm

- Find the `node` to delete (and its `parent`) and save the item to return later
- Branch based on number of children:
  - Case 0: the node to delete is a leaf and has no children
    - Set the child link from `parent` to this `node` to `None`
  - Case 1: the node to delete has a single child
    - Set the child link from `parent` to this `node` to the `node`'s child
  - Case 2: the node to delete has two children
    - Find the successor node
    - Copy the item from the successor to `node`
    - Recursively call the delete routine on the successor node
- Return the deleted item, if found