

Trees

Author

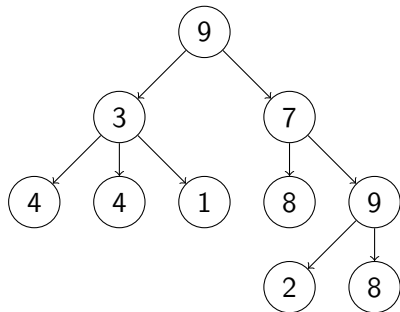
University of Illinois Urbana-Champaign

Date

Objectives

- Identify the core components of trees in general
- Cover the implementation of Binary Tree nodes and traversal algorithms.
- Identify the motivations for using Binary Search Trees (BST)
- Cover the implementation of the following BST methods:
 - Search
 - Adding a node
 - Removing a node

Unifying Structure

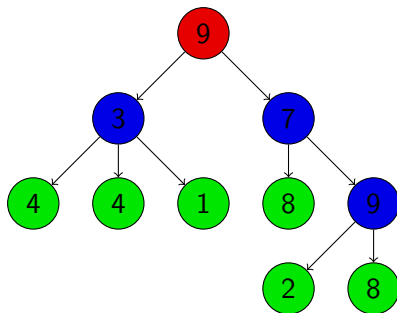


Trees have the following general, properties:

- *Directed*: Like a singly linked list a node does not have a reference to its parent.
- *Acyclic*: There are no “circles” of reference in the tree.
- A collection of nodes with:
 - Some data (generally)
 - The ability to reference child nodes

Tree Terminology: Structural

- *root* (red)
- *branch* (blue)
- *leaf* (green)

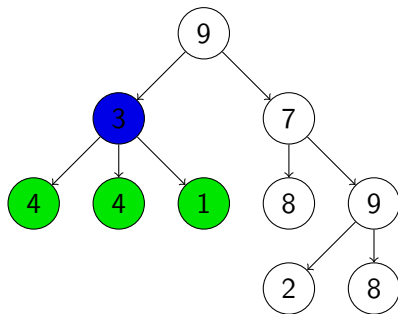


Tree Terminology: Relationship Based

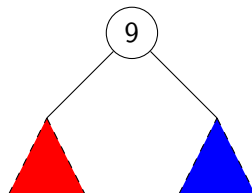
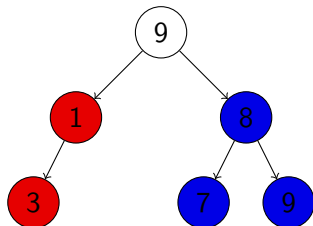
Terms based on relationships between nodes

- *parent* (blue)
- *child/sibling* (green)

The blue node has three *child* nodes (green) and, as a result, the green nodes are siblings.



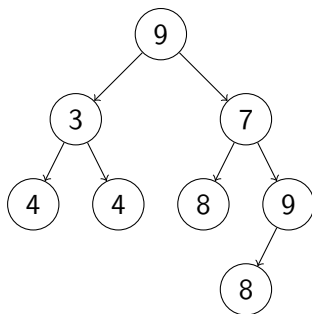
Tree Terminology: Subtrees



Binary Tree Structure

Binary Tree: Just a tree where each node has *at most two children*.

```
class IntTree{  
    static class IntTreeNode{  
        int data;  
        IntTreeNode left;  
        IntTreeNode right;  
  
        IntTreeNode(int data){  
            //..  
        }  
    }  
  
    IntTreeNode root;  
    //..  
}
```



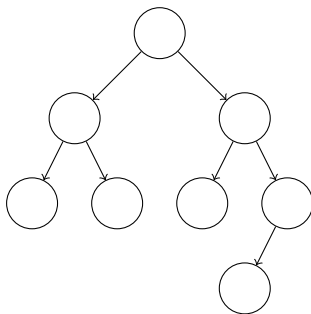
Binary Tree Structure

Binary Tree: Just a tree where each node has *at most two children*.

```
class GenericTree<E>
{
    class GenericTreeNode<E>{
        E data;
        GenericTreeNode<E> left;
        GenericTreeNode<E> right;

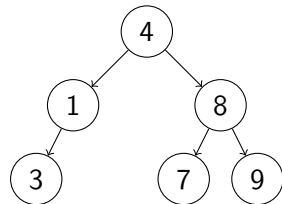
        GenericTreeNode(E data){
            // ..
        }
    }

    GenericTreeNode<E> root;
    // ..
}
```



Binary Tree: Manual Construction

```
class Tree{  
  
    IntTreeNode root;  
  
    Tree(){  
        root = new IntTreeNode(4);  
        root.left = new IntTreeNode(1);  
        root.left.left = new IntTreeNode(3);  
        root.right = new IntTreeNode(8);  
        root.right.left = new IntTreeNode(7);  
        root.right.right = new IntTreeNode(9);  
    }  
  
    public static void main(String[] args){  
        Tree myTree = new Tree();  
    }  
}
```



Recursive Traversal Algorithms

Function Inorder(*curr*)

if *curr is null* **then**

 return

end

 Inorder(node.left)

 print(node)

 Inorder(node.right)

return

Inorder: left, visit, right

Recursive Traversal Algorithms

Function Inorder(*curr*)

if *curr* is null **then**

 return

end

 Inorder(node.left)

 print(node)

 Inorder(node.right)

return

Inorder: left, visit, right

Function Preorder(*curr*)

if *curr* is null **then**

 return

end

 print(node)

 Preorder(node.left)

 Preorder(node.right)

return

Preorder: visit, left, right

Recursive Traversal Algorithms

Function Inorder(*curr*)

```
if curr is null then
    return
end
Inorder(node.left)
print(node)
Inorder(node.right)
return
```

Inorder: left, visit, right

Function Preorder(*curr*)

```
if curr is null then
    return
end
print(node)
Preorder(node.left)
Preorder(node.right)
return
```

Preorder: visit, left, right

Function Postorder(*curr*)

```
if curr is null then
    return
end
Postorder(node.left)
Postorder(node.right)
print(node)
return
```

Postorder: left, right, visit

Recursive Traversal Algorithms

Function Inorder(*curr*)

```
if curr is null then
    return
end
Inorder(node.left)
print(node)
Inorder(node.right)
return
```

Inorder: left, visit, right

Function Preorder(*curr*)

```
if curr is null then
    return
end
print(node)
Preorder(node.left)
Preorder(node.right)
return
```

Preorder: visit, left, right

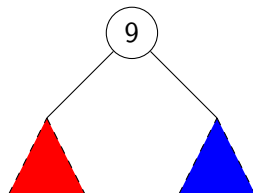
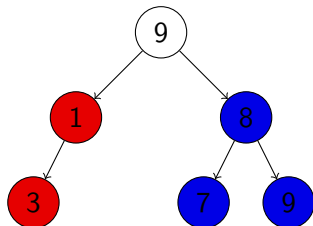
Function Postorder(*curr*)

```
if curr is null then
    return
end
Postorder(node.left)
Postorder(node.right)
print(node)
return
```

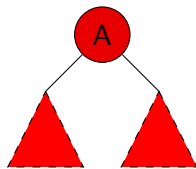
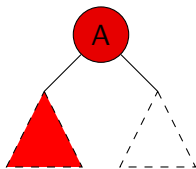
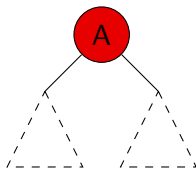
Postorder: left, right, visit

Left, right, visit!? What does it mean!?

Traversals: Recall Subtrees



Traversals: Preorder Intuition



For a given node A visit that node immediately upon encountering it and then traverse to its left and right subtrees

Traversals

Function Preorder(*curr*)

if *curr* is null **then**

 return

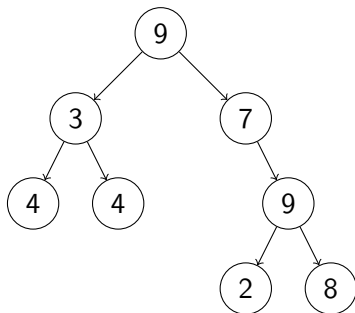
end

 print(*curr*)

 Preorder(*curr*.left)

 Preorder(*curr*.right)

return



Traversals

Function Preorder(*curr*)

if *curr* is null **then**

 return

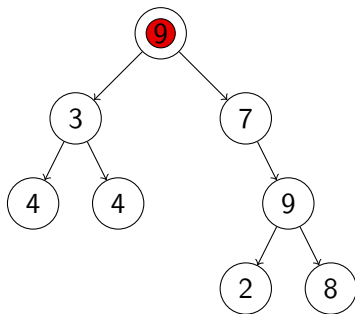
end

 print(*curr*)

 Preorder(*curr*.left)

 Preorder(*curr*.right)

return



Traversals

Function Preorder(*curr*)

if *curr* is null **then**

 return

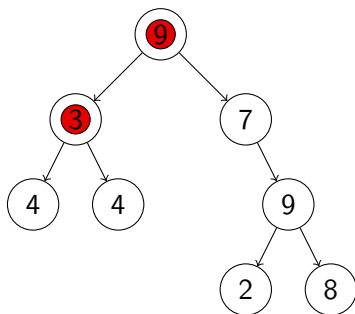
end

 print(*curr*)

 Preorder(*curr*.left)

 Preorder(*curr*.right)

return



Traversals

Function Preorder(*curr*)

if *curr* is null **then**

 return

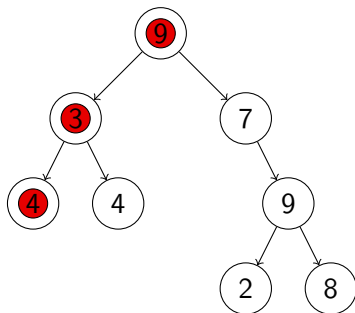
end

 print(*curr*)

 Preorder(*curr*.left)

 Preorder(*curr*.right)

return



Traversals

Function Preorder(*curr*)

if *curr* is null **then**

 return

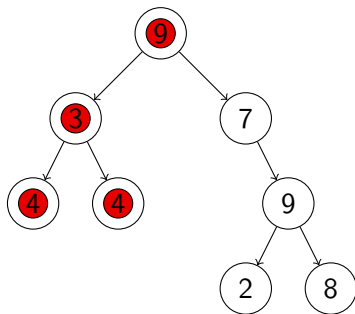
end

 print(*curr*)

 Preorder(*curr*.left)

 Preorder(*curr*.right)

return



Traversals

Function Preorder(*curr*)

if *curr* is null **then**

 return

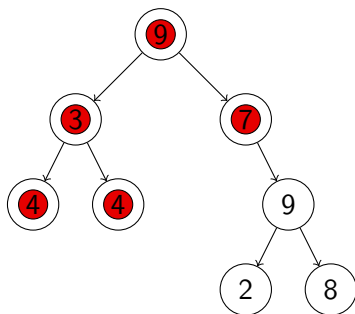
end

 print(*curr*)

 Preorder(*curr*.left)

 Preorder(*curr*.right)

return



Traversals

Function Preorder(*curr*)

if *curr* is null **then**

 return

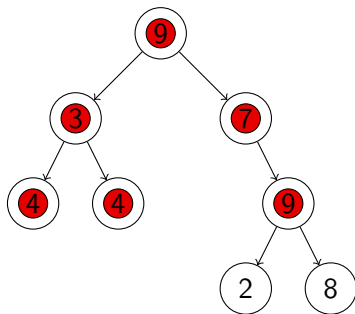
end

 print(*curr*)

 Preorder(*curr*.left)

 Preorder(*curr*.right)

return



Traversals

Function Preorder(*curr*)

if *curr* is null **then**

 return

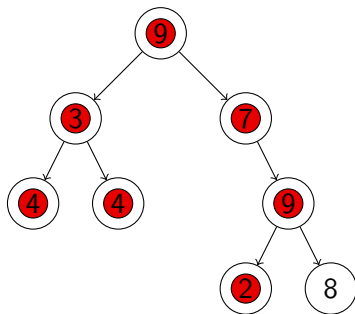
end

 print(*curr*)

 Preorder(*curr*.left)

 Preorder(*curr*.right)

return



Traversals

Function Preorder(*curr*)

if *curr* is null **then**

 return

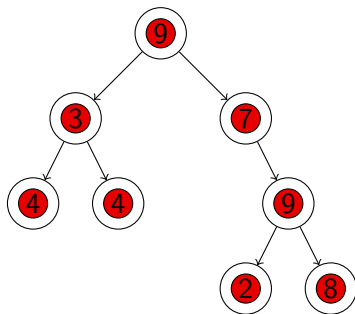
end

 print(*curr*)

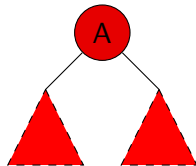
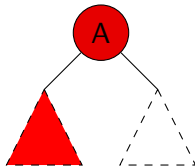
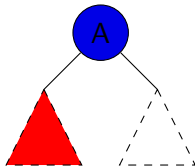
 Preorder(*curr*.left)

 Preorder(*curr*.right)

return



Traversals: Inorder Intuition



For a given node A, all nodes in A's left subtree must be visited before we visit A.

Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

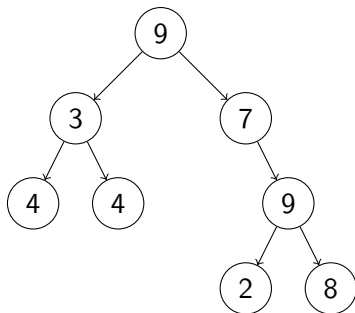
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

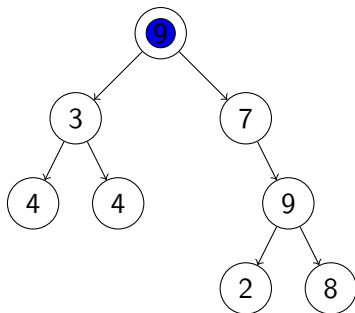
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

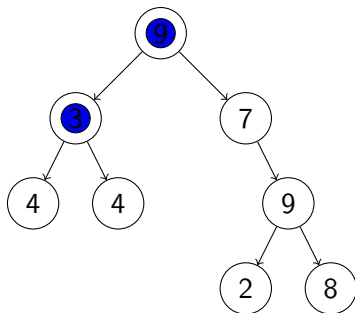
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

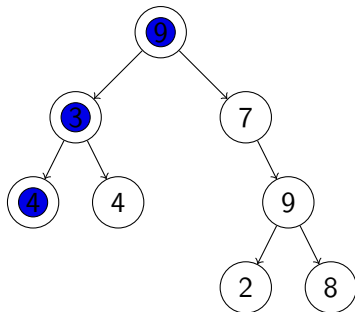
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

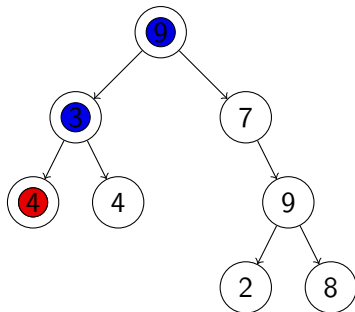
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

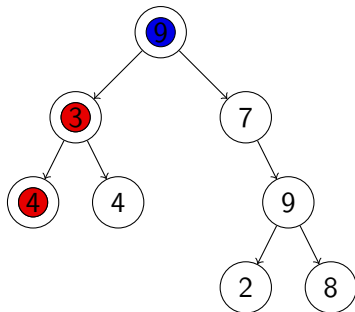
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

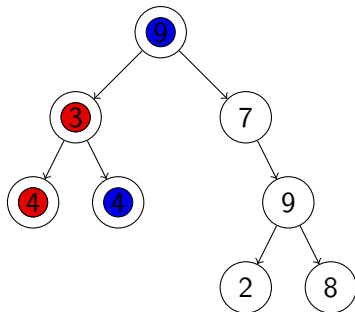
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

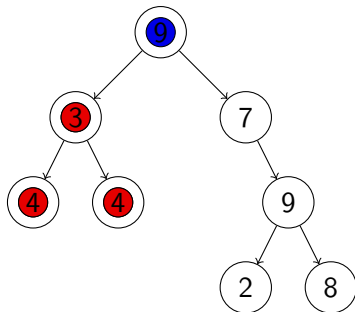
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

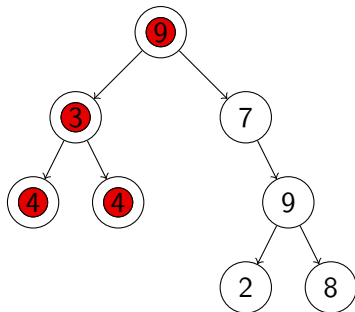
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

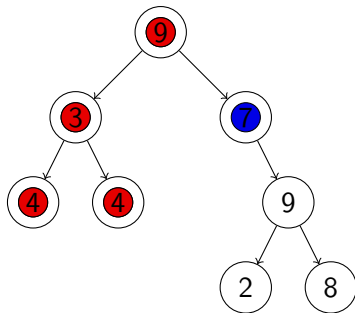
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

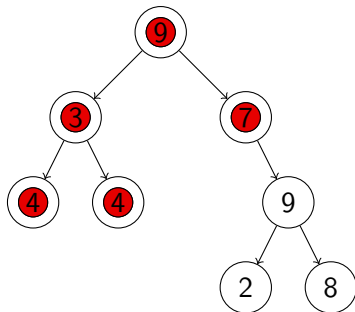
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

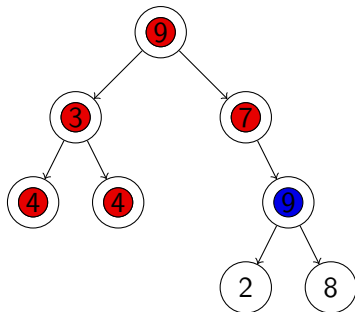
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

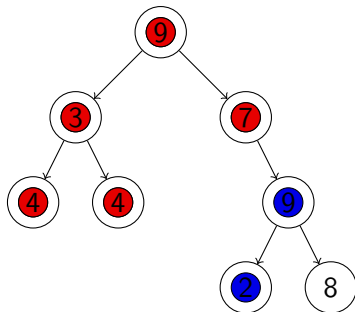
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

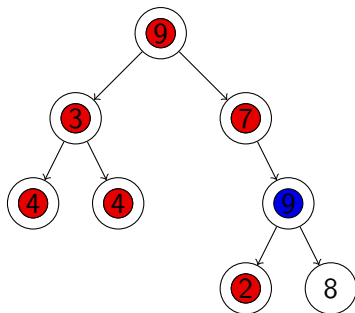
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

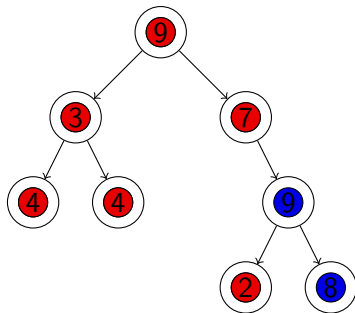
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

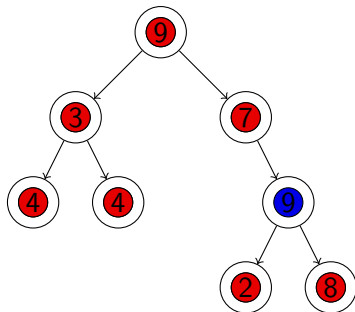
end

 Inorder(*curr*.left)

 print(*curr*)

 Inorder(*curr*.right)

return



Traversals

Function Inorder(*curr*)

if *curr* is null **then**

 return

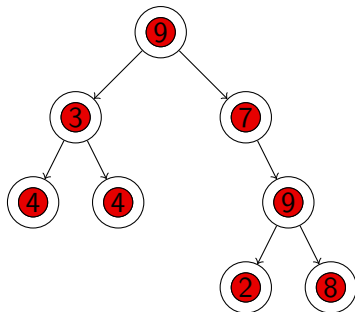
end

 Inorder(*curr*.left)

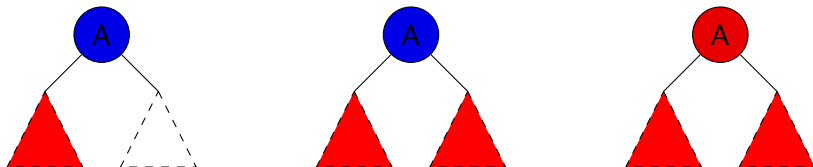
 print(*curr*)

 Inorder(*curr*.right)

return



Traversals: Postorder Intuition



For a given node A, all of the nodes in A's left and right subtrees must be visited before A can be visited.

Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

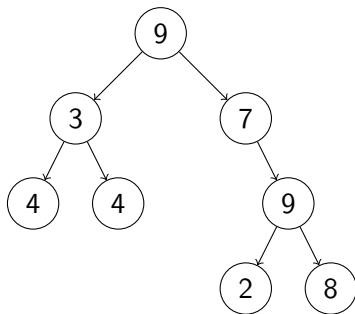
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

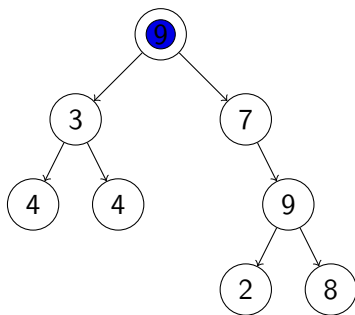
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

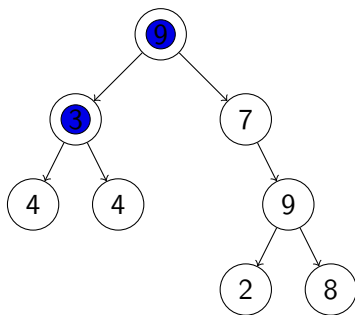
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

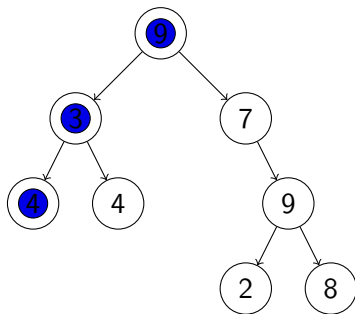
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

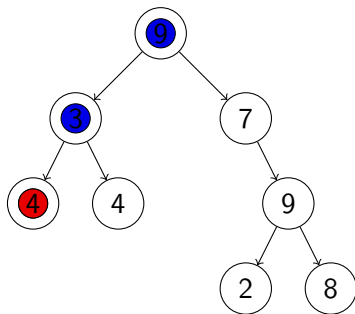
end

Postorder(*curr*.left)

Postorder(*curr*.right)

print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

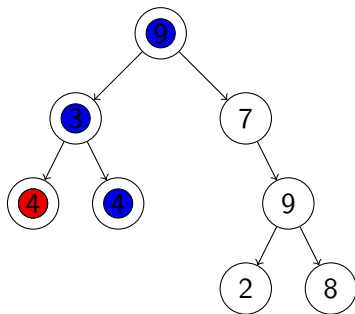
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

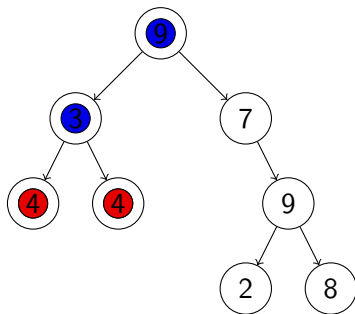
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

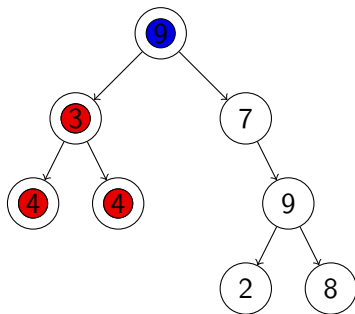
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

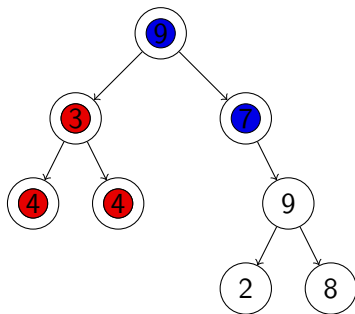
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

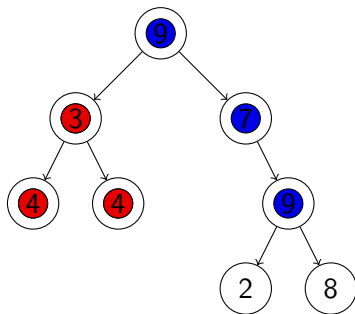
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

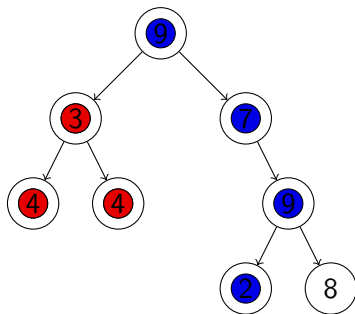
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

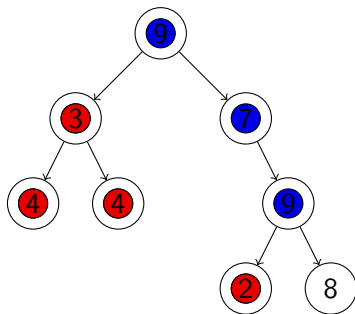
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

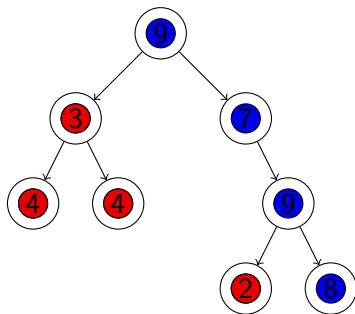
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

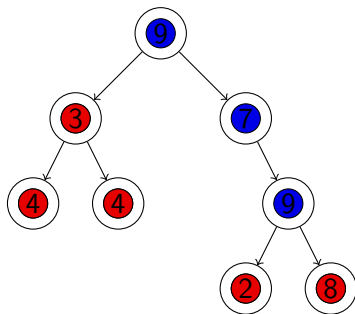
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

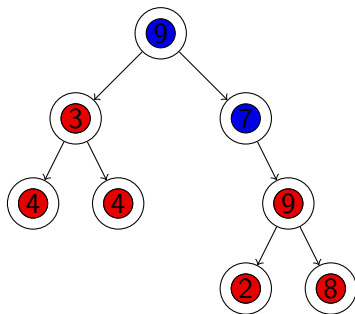
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

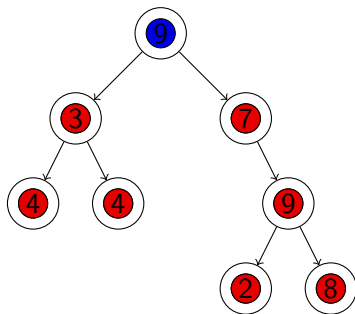
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

return



Traversals: Postorder

Function Postorder(*curr*)

if *curr* is null **then**

 return

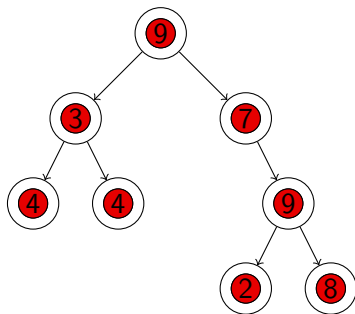
end

 Postorder(*curr*.left)

 Postorder(*curr*.right)

 print(*node*)

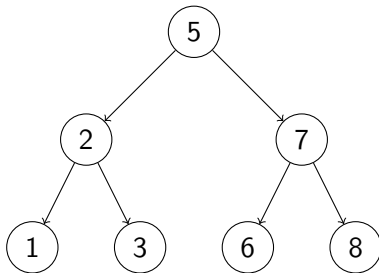
return



Binary Search Tree: Structure

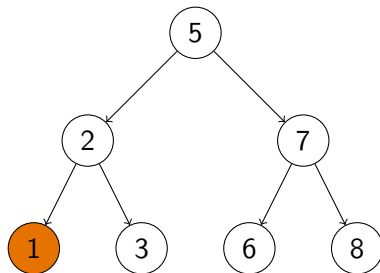
Binary Search Tree: A tree where:

- ① each node has *at most two children*.
- ② the value of a given node is greater than the value of every node in its left subtree.
- ③ the value of a given node is less than the value of every node in its right subtree.



Binary Tree: Terms and Basic Operations

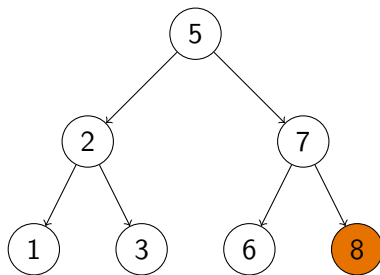
```
Function Minimum(curr)
|   while curr.left  $\neq$  null do
|   |   curr = curr.left
|   end
return curr
```



Minimum Node: Starting at a given node *curr*, the node with the least value under *curr* will be the leftmost node in it's left subtree.

Binary Search Tree: Maximum Node

```
Function Maximum(curr)
  while curr.right  $\neq$  null do
    | curr = curr.right
  end
return curr
```



Maximum Node: Starting at a given node *curr*, the node with the greatest value under *curr* will be the rightmost node in it's right subtree.

Binary Search Tree: Finding a Node

```

Function RecursiveSearch(Curr, Val)
  if Curr = NULL OR Val = Curr.Val then
    | return curr
  end
  if Val < Curr.Val then
    | return RecursiveSearch(Curr.Left, Val)
  end
  else
    | return RecursiveSearch(Curr.Right, Val)
  end
return
    
```

- Though this can be done iteratively or recursively, both methods follow the pattern of finding and returning a reference to a node with a given value.

Binary Search Tree: Finding a Node

```
Function RecursiveSearch(Curr, Val)
  if Curr = NULL OR Val = Curr.Val then
    | return curr
  end
  if Val < Curr.Val then
    | return RecursiveSearch(Curr.Left, Val)
  end
  else
    | return RecursiveSearch(Curr.Right, Val)
  end
return
```

- Though this can be done iteratively or recursively, both methods follow the pattern of finding and returning a reference to a node with a given value.
- Searching will also form the basis for adding and removing nodes.

Adding a Node

```
Function Add(Data)
    |   Root = Add(Root, Data);
return
```

```

Function Add(Curr, Data)
    if Curr = NULL then
        | return TreeNode(Data)
    end
    else if Data > Curr.Data then
        | Curr.Right ← Add(Curr.Right, Data)
    end
    else
        | Curr.Left ← Add(Curr.Left, Data)
    end
    return Curr
return

```

- **Case 1 (red):** Curr is null so we create and return the new node.
- **Case 2 (blue):** Curr isn't null so we need to keep traversing according to BST rules.

Binary Search Tree: Removing a Node

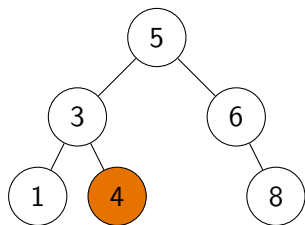


Figure 1: Leaf

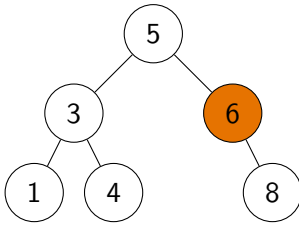


Figure 2: One Subtree

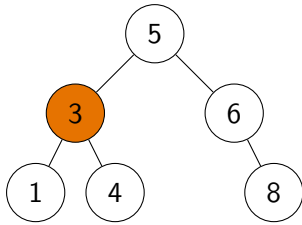


Figure 3: Two Subtrees

There are three cases for removing a node:

Binary Search Tree: Removing a Node

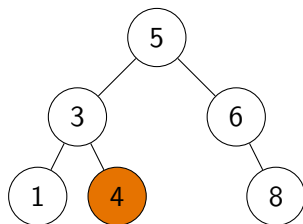


Figure 1: Leaf

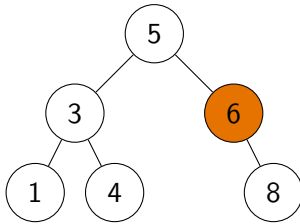


Figure 2: One Subtree

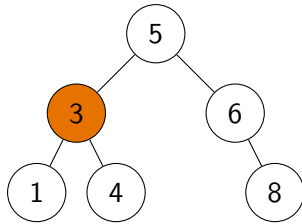


Figure 3: Two Subtrees

There are three cases for removing a node:

- 1 The node we are removing is a leaf (Figure 1).

Binary Search Tree: Removing a Node

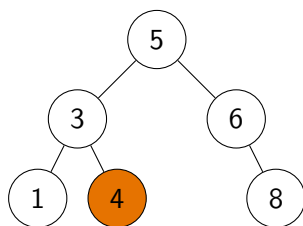


Figure 1: Leaf

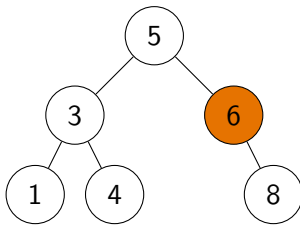


Figure 2: One Subtree

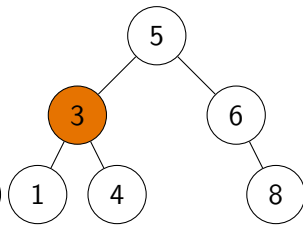


Figure 3: Two Subtrees

There are three cases for removing a node:

- ➊ The node we are removing is a leaf (Figure 1).
- ➋ The node we are removing has one subtree (Figure 2).

Binary Search Tree: Removing a Node

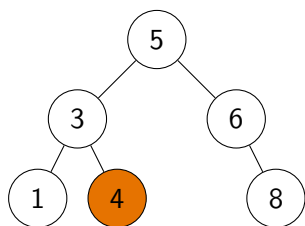


Figure 1: Leaf

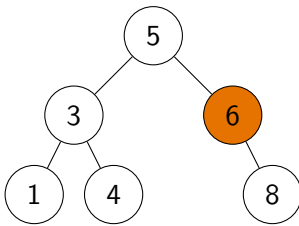


Figure 2: One Subtree

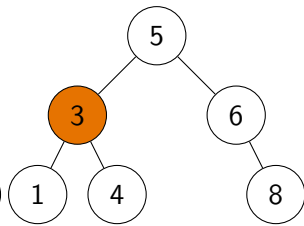


Figure 3: Two Subtrees

There are three cases for removing a node:

- ➊ The node we are removing is a leaf (Figure 1).
- ➋ The node we are removing has one subtree (Figure 2).
- ➌ The node we are removing has two subtrees (Figure 3).

Case 1: Removing a Leaf

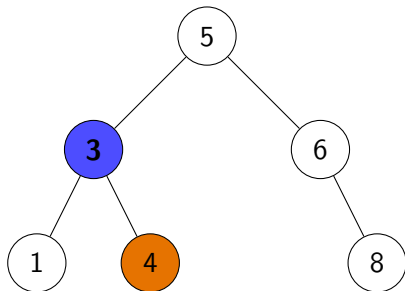


Figure 4: Find the node you want to remove (orange), in this case the one with value 4, and that node's parent (blue)

Case 1: Removing a Leaf

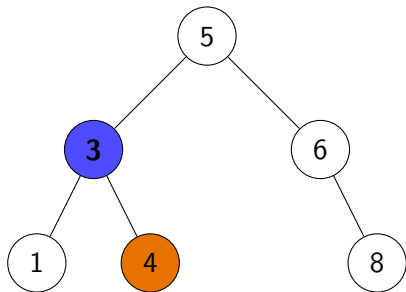


Figure 4: Find the node you want to remove (orange), in this case the one with value 4, and that node's parent (blue)

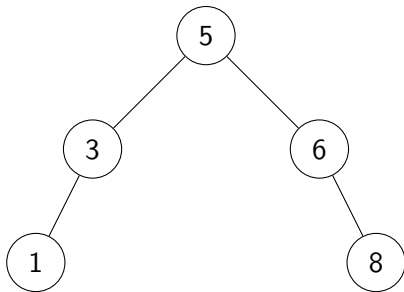


Figure 5: Set the parent's reference to that node, in this case it's parents right reference equal to `null`, to remove the node

Case 2: Removing a Node with One Subtree

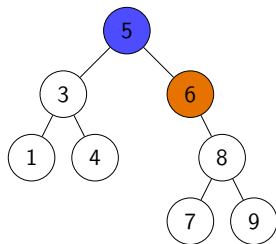


Figure 6: Find the node you want to remove (orange) and that node's parent (blue)

Case 2: Removing a Node with One Subtree

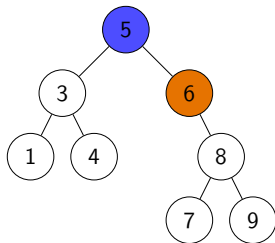


Figure 6: Find the node you want to remove (orange) and that node's parent (blue)

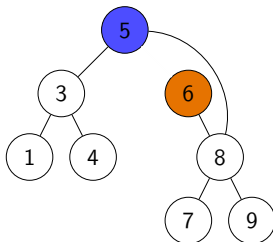


Figure 7: Set `parent.right` to the root of the node we want to remove's subtree.

Case 2: Removing a Node with One Subtree

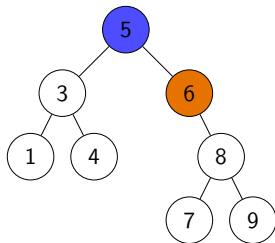


Figure 6: Find the node you want to remove (orange) and that node's parent (blue)

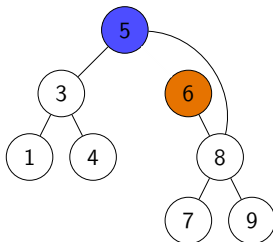


Figure 7: Set `parent.right` to the root of the node we want to remove's subtree.

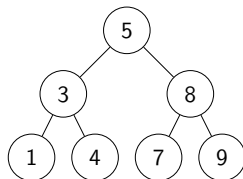


Figure 8: Set the node we want to remove's reference to it's subtree is **null**, thus removing it from the tree

Case 3: Removing a Node with Two Subtrees

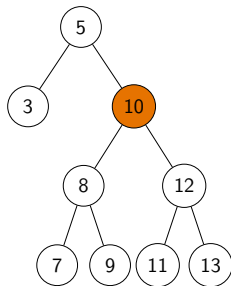


Figure 9: Find the node you want to remove (orange)

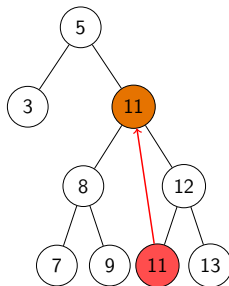


Figure 10: Find the successor node (red) and copy the successor's value to the node we want to remove.

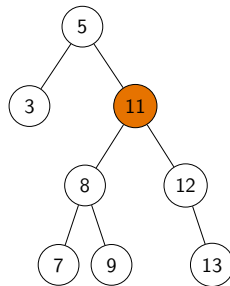


Figure 11: Call the removal method on the successor node.

Removing a Node

```
Function Remove(Val)
|   root = Remove(Root, Val);
```

```
Function Remove(Curr, Val)
|   if curr = null then
|       return null
|   end
|   if Curr.val < Val then
|       Curr.right ← Remove(Curr.right, Val)
|   end
|   else if Curr.val > Val then
|       Curr.left ← Remove(Curr.left, Val)
|   end
|   else
|       if Curr.Left is NULL AND Curr.Right is NULL then
|           return NULL
|       end
|       else if Curr.left is null then
|           return Curr.right
|       end
|       else if Curr.right is null then
|           return Curr.left
|       end
|       else
|           MinNode ← Minimum(Curr.right)
|           Curr.data ← MinNode.data
|           Curr.right ← Remove(Curr.right, MinNode.data)
|       end
|   end
end
```

- **Step 1 (red):** Here we are recurring down the tree to search for the node we want to remove.

Removing a Node

```
Function Remove(Val)
|   root = Remove(Root, Val);
```

```
Function Remove(Curr, Val)
|   if curr = null then
|       return null
|   end
|   if Curr.val < Val then
|       Curr.right ← Remove(Curr.right, Val)
|   end
|   else if Curr.val > Val then
|       Curr.left ← Remove(Curr.left, Val)
|   end
|   else
|       if Curr.Left is NULL AND Curr.Right is NULL then
|           return NULL
|       end
|       else if Curr.left is null then
|           return Curr.right
|       end
|       else if Curr.right is null then
|           return Curr.left
|       end
|       else
|           MinNode ← Minimum(Curr.right)
|           Curr.data ← MinNode.data
|           Curr.right ← Remove(Curr.right, MinNode.data)
|       end
|   end
end
```

- **Step 1 (red):** Here we are recurring down the tree to search for the node we want to remove.
- **Step 2:** Remove using cases

Removing a Node

Function Remove(*Val*)

 | root = Remove(Root, *Val*);

Function Remove(*Curr*, *Val*)

 | if *curr* = null then

 | return null

 | end

 | if *Curr.val* < *Val* then

 | *Curr.right* ← Remove(*Curr.right*, *Val*)

 | end

 | else if *Curr.val* > *Val* then

 | *Curr.left* ← Remove(*Curr.left*, *Val*)

 | end

 | else

 | if *Curr.Left* is NULL AND *Curr.Right* is NULL then

 | return NULL

 | end

 | else if *Curr.left* is null then

 | return *Curr.right*

 | end

 | else if *Curr.right* is null then

 | return *Curr.left*

 | end

 | else

 | MinNode ← Minimum(*Curr.right*)

 | *Curr.data* ← MinNode.data

 | *Curr.right* ← Remove(*Curr.right*, MinNode.data)

 | return *Curr*

 | end

 | end

- **Step 1 (red):** Here we are recurring down the tree to search for the node we want to remove.
- **Step 2:** Remove using cases
 - **Case 1 (green):** No subtree.

Removing a Node

```
Function Remove(Val)
|   root = Remove(Root, Val);
```

```
Function Remove(Curr, Val)
|   if curr = null then
|       |   return null
|   end
|   if Curr.val < Val then
|       |   Curr.right ← Remove(Curr.right, Val)
|   end
|   else if Curr.val > Val then
|       |   Curr.left ← Remove(Curr.left, Val)
|   end
|   else
|       if Curr.Left is NULL AND Curr.Right is NULL then
|           |   return NULL
|       end
|       else if Curr.left is null then
|           |   return Curr.right
|       end
|       else if Curr.right is null then
|           |   return Curr.left
|       end
|       else
|           |   MinNode ← Minimum(Curr.right)
|           |   Curr.data ← MinNode.data
|           |   Curr.right ← Remove(Curr.right, MinNode.data)
|           |   return Curr
|       end
|   end
end
```

- **Step 1 (red):** Here we are recurring down the tree to search for the node we want to remove.
- **Step 2:** Remove using cases
 - **Case 1 (green):** No subtree.
 - **Case 2 (blue):** One subtree.

Removing a Node

```
Function Remove(Val)
|   root = Remove(Root, Val);
```

```
Function Remove(Curr, Val)
|   if curr = null then
|       return null
|   end
|   if Curr.val < Val then
|       Curr.right ← Remove(Curr.right, Val)
|   end
|   else if Curr.val > Val then
|       Curr.left ← Remove(Curr.left, Val)
|   end
|   else
|       if Curr.Left is NULL AND Curr.Right is NULL then
|           return NULL
|       end
|       else if Curr.left is null then
|           return Curr.right
|       end
|       else if Curr.right is null then
|           return Curr.left
|       end
|       else
|           MinNode ← Minimum(Curr.right)
|           Curr.data ← MinNode.data
|           Curr.right ← Remove(Curr.right, MinNode.data)
|           return Curr
|       end
|   end
end
```

- **Step 1 (red):** Here we are recurring down the tree to search for the node we want to remove.
- **Step 2:** Remove using cases
 - **Case 1 (green):** No subtree.
 - **Case 2 (blue):** One subtree.
 - **Case 3 (purple):** Both subtrees.

Searching Unsorted Linear Data Structures vs Searching Binary Search Tree

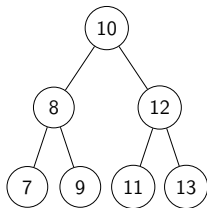


Figure 12: Balanced BST

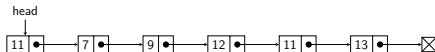


Figure 13: Linked List

- Searching a *balanced* BST, search is $O(\log_2(n))$ where n is the number of nodes in the tree.
- Recall, searching a linked list is $(\log_2(n))$ where n is the number of nodes in the list.

The Issue of Unbalanced Binary Search Trees

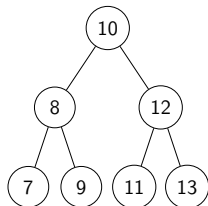


Figure 14: Balanced BST

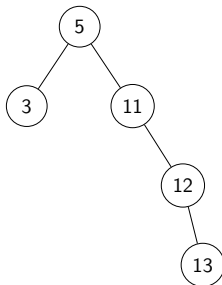


Figure 15: Unbalanced BST

- For *balanced* BST, search is $O(\log_2(n))$ where n is the number of nodes in the tree.
- For *unbalanced* BST, search is $O(h)$ where h is the height of the tree.
- We want to make sure our tree remains balanced when we insert/delete but this can't be done with traditional BST.
- **Next week:** AVL self balancing trees