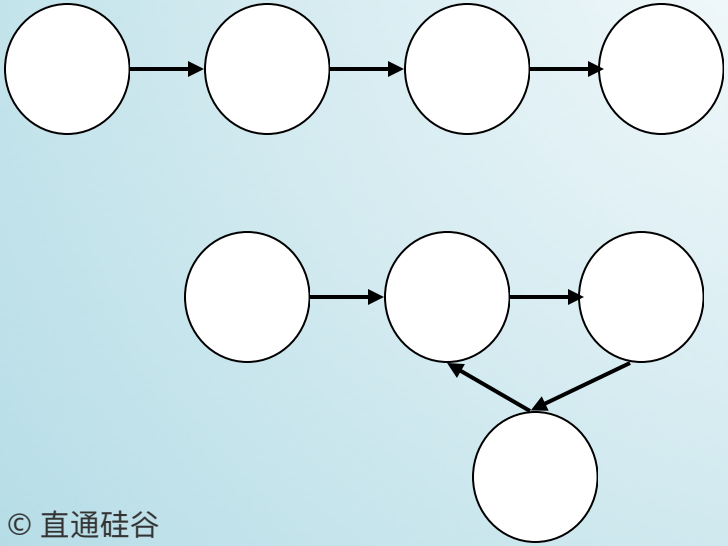
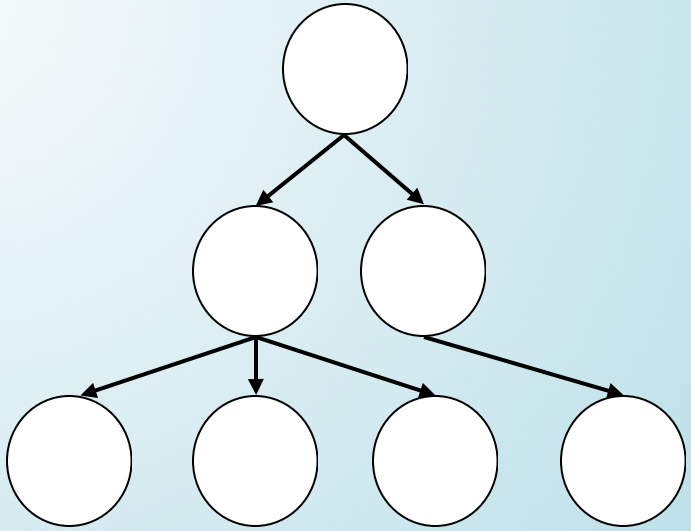
Tree

Definition

A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle.





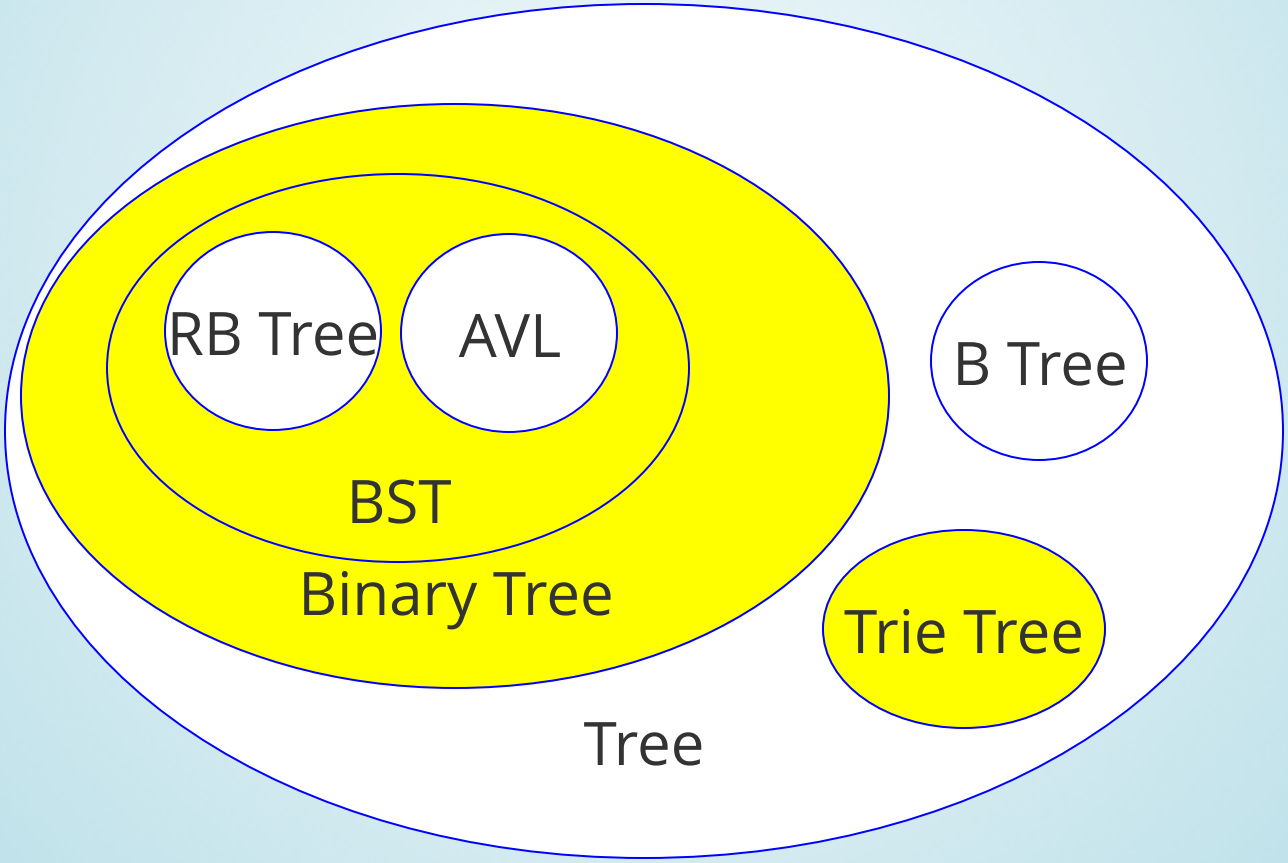
Tree

Tree is one of the most important data structure in algorithms

Like our current File System

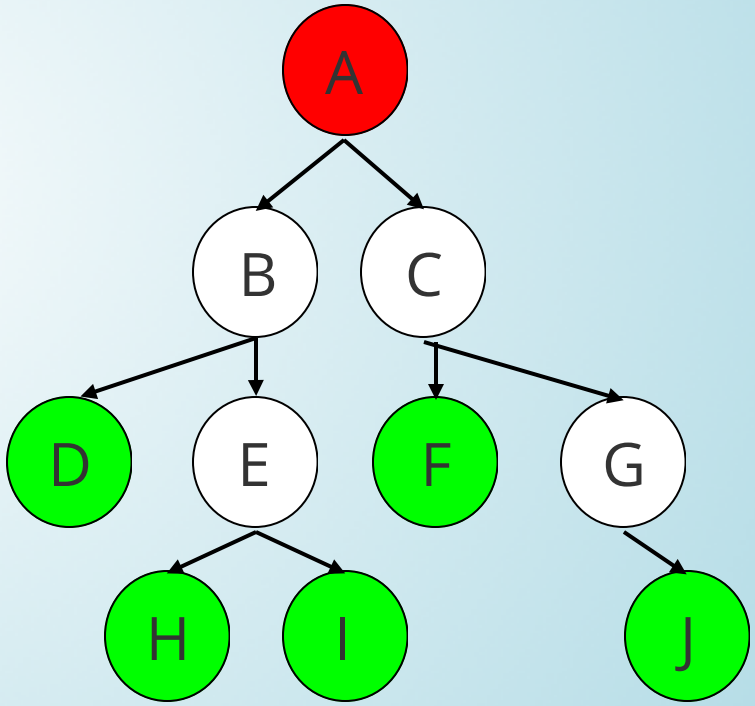
Tree has many different types

Tree



Terminology

- Node
  - Root
  - Leaf
  - Parent
  - Child
  - Siblings
  - Ancestor
  - Descendant
- Edge
- Height
- Depth
- Level
- Path

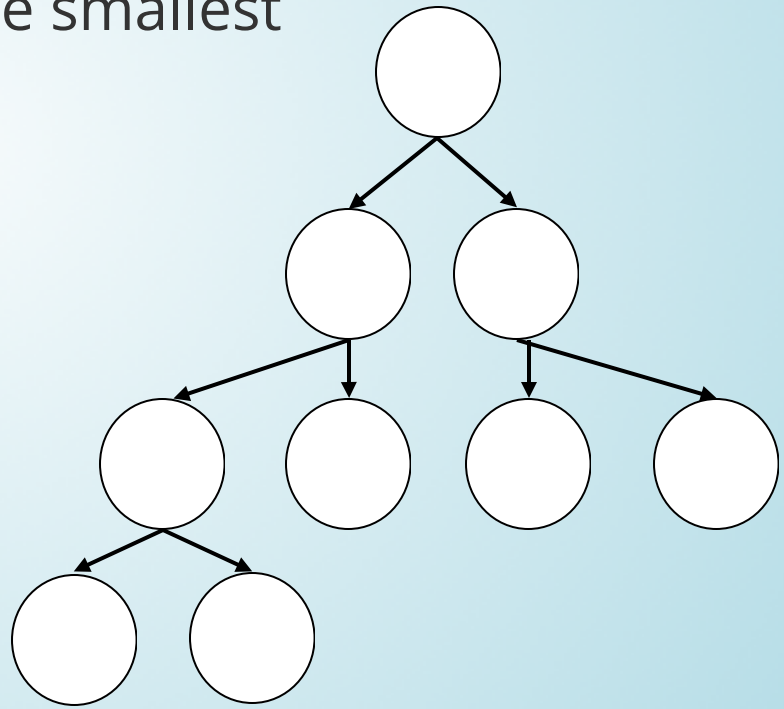


Binary Tree

- It is actually different from tree
- It has right sub-tree and left sub-tree
- Each node can have at most two children

Complete Binary Tree

- Very useful in algorithm problems
- Always assuming as a tree's property to compute the time complexity
- The sum of the paths are the smallest



Properties of Binary Tree

- at Level i, at most 2^i nodes
- a tree with height k, at most 2^k-1 nodes
- a complete binary tree with n nodes, the height will be $\lceil log_2(n+1) \rceil$
- If we number the node from root and base on the level, for a complete binary tree, we will have:
  - for node number k, the left child is 2k+1
  - for node number k, the right child is 2k+2
- How to store a Binary Tree? a Complete Binary Tree?

Basic Data Structure of a Binary Tree

```
public class BinaryTree<T> {
    private Node<T> root;

    public Tree(T rootData) {
        root = new Node<T>();
        root.data = rootData;
    }

    public static class Node<T> {
        private T data;
        private Node<T> leftNode;
        private Node<T> rightNode;
    }
}
```

Traversal of a Binary Tree

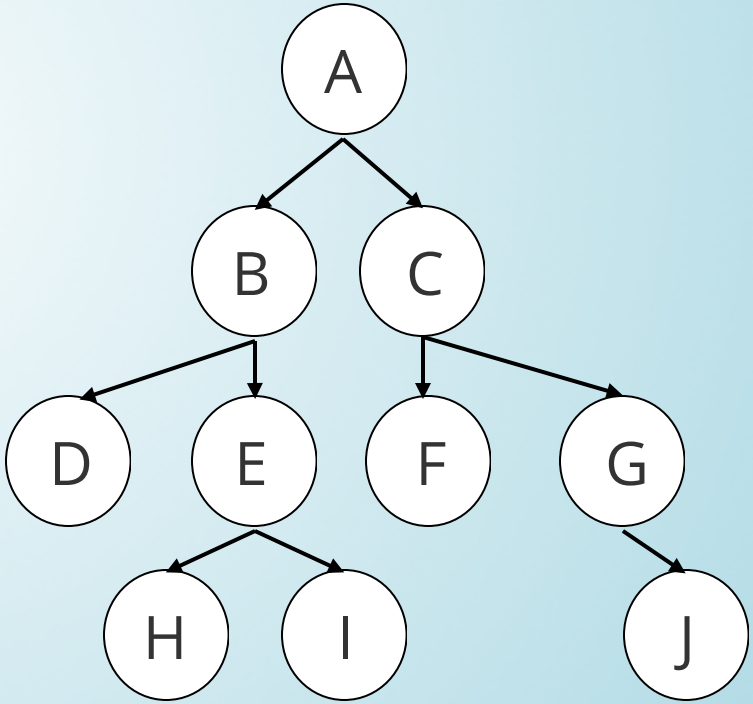
- PreOrder: Parent, left child, right child
- InOrder: left child, Parent, right child
- PostOrder: left child, right child, Parent

Traversal of a Binary Tree

PreOrder: A B D E H I C F G J

InOrder: D B H E I A F C G J

PostOrder: D H I E B F J G C A



Traversal of a Binary Tree

We need to use recursion to traverse a tree

```
public void preorder(TreeNode root) {
    if(root !=  null) {
   //Visit the node by Printing the node data
      System.out.printf("%c ",root.data);
      preorder(root.left);
      preorder(root.right);
    }
}
```

Traversal of a Binary Tree

```
public void inorder(TreeNode root) {
    if(root !=  null) {
      inorder(root.left);
      System.out.printf("%c ",root.data);
      inorder(root.right);
    }
}
```

```
public void postorder(TreeNode root) {
    if(root !=  null) {
      postorder(root.left);
      postorder(root.right);
      System.out.printf("%c ",root.data);
    }
}
```

Traversal of a Binary Tree

Can we use Stack to traverse the tree without recursion?

Traversal of a Binary Tree

PreOrder: visit, push right, push left

Because we need to visit left first

Traversal of a Binary Tree

```
public static void PreOrder(Node root) {
        Stack<Node> nodeStack = new Stack<Node>();
        nodeStack.push(root);
        while(!nodeStack.empty()) {
                Node node = nodeStack.pop();
                System.out.printf("%c ", node.data);
                if(node.rightNode != null) {
                        nodeStack.push(node.rightNode);
                }
                if(node.leftNode != null) {
                        nodeStack.push(node.leftNode);
                }
        }
}
```

Traversal of a Binary Tree

Do we have a better way?

Could we save some space on the stack?

Do we need to push both children to the stack?

Traversal of a Binary Tree

```
public static void PreOrder2(Node root) {
        Stack<Node> nodeStack = new Stack<Node>();
        nodeStack.push(root);
        Node node = root;
        while(!nodeStack.empty()) {
                System.out.printf("%c ", node.data);
                if(node.rightNode != null) {
                        nodeStack.push(node.rightNode);
                }
                if(node.leftNode != null) {
                        node = node.leftNode;
                }
                else {
                        node = nodeStack.pop();
                }
        }
}
```

Traversal of a Binary Tree

InOrder: What we should do?

InOrder: We push the node, go to left

If there is no left, we pop, print and push the right

Traversal of a Binary Tree

```
public static void InOrder(Node root) {
        Stack<Node> nodeStack = new Stack<Node>();
        Node node = root;
        while(!nodeStack.empty() || node != null) {
                if(node != null) {
                        nodeStack.push(node);
                        node = node.leftNode;
                }
                else {
                        node = nodeStack.pop();
                        System.out.printf("%c ", node.data);
                        node = node.rightNode;
                }
        }
}
```

Traversal of a Binary Tree

PostOrder: The most difficult one

PostOrder: go to left, when there is no left, visit it.
Then for the top of the stack, do we visit it now?

No. Because it is what InOrder does

We need to then go to the right until there
is nowhere to go, we visit it and then pop

Traversal of a Binary Tree

PostOrder: The most difficult one

Can we do the PostOrder just using the current structure?

No. We do not know the status of the top element in the stack. Has the right child been traversed or not?

Traversal of a Binary Tree

We need an additional flag to store the status of the node

```
static class NodeWithFlag {
        Node node;
        boolean flag;
        public NodeWithFlag(Node n, boolean value) {
                node = n;
                flag = value;
        }
}
```

After we visit the right child, we update the flag;

Traversal of a Binary Tree

```
public static void PostOrder(Node root) {
        Stack<NodeWithFlag> nodeStack = new Stack<NodeWithFlag>();
        Node curNode = root;
        NodeWithFlag newNode;
        while(!nodeStack.empty() || curNode != null) {
                while(curNode != null) {
                        newNode = new NodeWithFlag(curNode, false);
                        nodeStack.push(newNode);
                        curNode = curNode.leftNode;
                }
                newNode = nodeStack.pop();
                curNode = newNode.node;
                if(!newNode.flag) {
                        newNode.flag = true;
                        nodeStack.push(newNode);
                        curNode = curNode.rightNode;
                }
                else {
                        System.out.printf("%c ", curNode.data);
                        curNode = null;
                }
        }
}
```

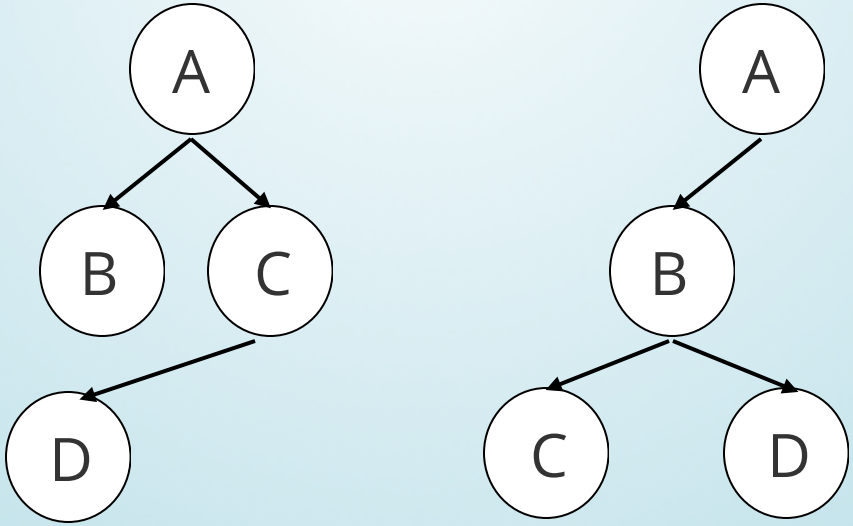
Traversal of a Binary Tree

Another way is to use a Hashmap to store which node has been visited

Construct a Binary Tree

If we have a traversal result of a tree, can we construct the tree?

PreOrder: A B C D



Construct a Binary Tree

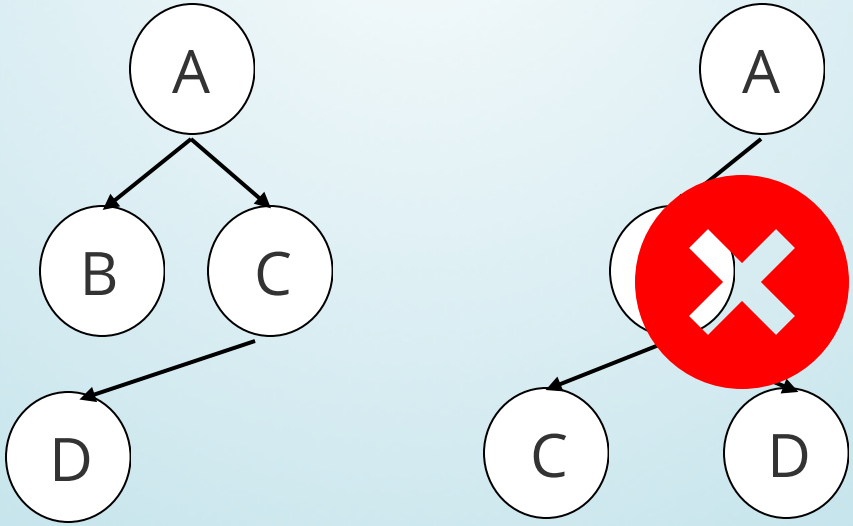
No matter if you give a PreOrder, InOrder or PostOrder, it can construct different trees

What if we give two traversals?

Construct a Binary Tree

PreOrder: A B C D

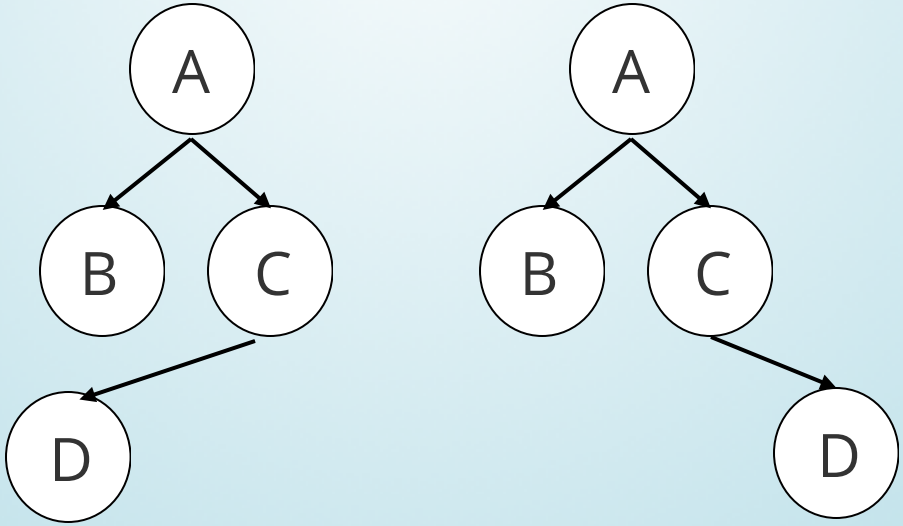
InOrder: B A D C



Construct a Binary Tree

PreOrder: A B C D

PostOrder: B D C A



Construct a Binary Tree

Conclusion: You need two traversals to construct a tree, and one of them must be a **InOrder** traversal

How to construct a tree with a PreOrder and an InOrder?

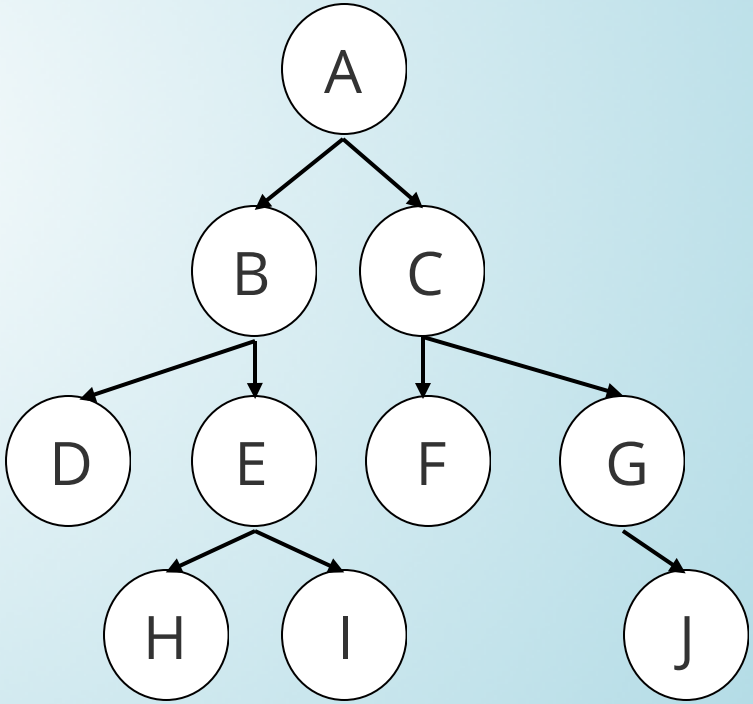
Construct a Binary Tree

PreOrder: A B D E H I C F G J

InOrder: D B H E I A F C G J

A is root so A is the first in PreOrder, InOrder can use A to seperate left sub-tree and right sub-tree

Then we can do recursion



PreOrder + InOrder

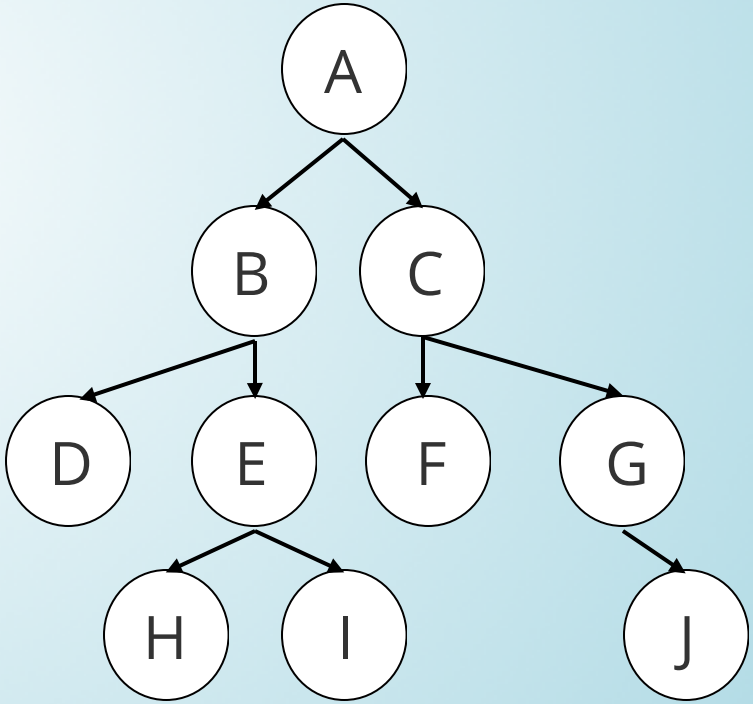
```
public TreeNode buildTree(int[] preorder, int[] inorder) {
    int preStart = 0;
    int preEnd = preorder.length-1;
    int inStart = 0;
    int inEnd = inorder.length-1;

    return construct(preorder, preStart, preEnd, inorder, inStart, inEnd);
}

public TreeNode construct(int[] preorder, int preStart, int preEnd, int[] inorder, int inStart, int inEnd){
    if(preStart>preEnd||inStart>inEnd){
        return null;
    }

    int val = preorder[preStart];
    TreeNode p = new TreeNode(val);
    int k=0;
    for(int i=inStart; i<=inEnd; i++){
        if(val == inorder[i]){
            k=i;
            break;
        }
    }

    p.left = construct(preorder, preStart+1, preStart+(k-inStart), inorder, inStart, k-1);
    p.right= construct(preorder, preStart+(k-inStart)+1, preEnd, inorder, k+1 , inEnd);
    return p;
}
```

Construct a Binary Tree

PostOrder: D H I E B F J G C A

InOrder: D B H E I A F C G J

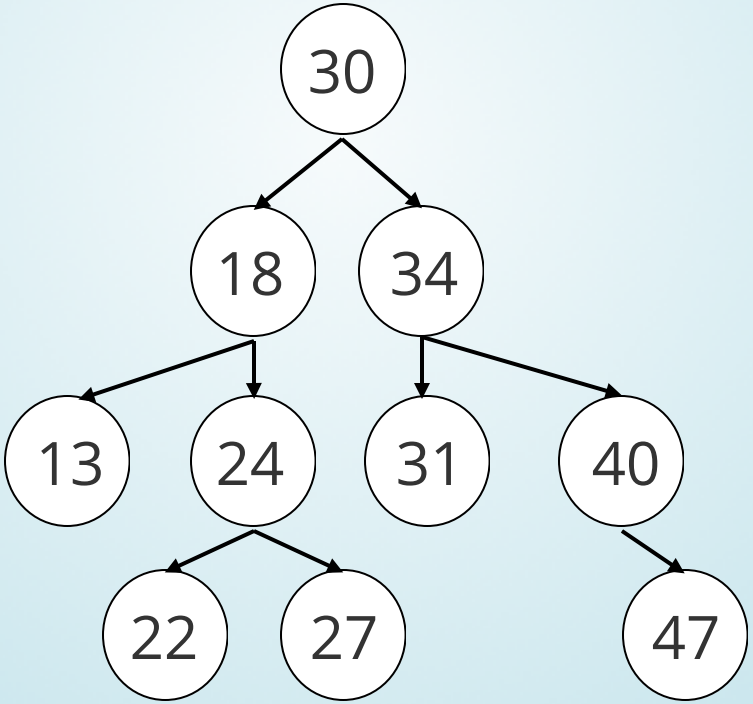
It is similar as PreOrder + InOrder, the only difference is the root now is the last one in PostOrder



PostOrder + InOrder

```
public TreeNode buildTree(int[] inorder, int[] postorder) {
        int inStart = 0;
        int inEnd = inorder.length - 1;
        int postStart = 0;
        int postEnd = postorder.length - 1;
        return build(inorder, inStart, inEnd, postorder, postStart, postEnd);
}
public TreeNode build(int[] inorder, int inStart, int inEnd, int[] postorder, int postStart, int postEnd) {
        if (inStart > inEnd || postStart > postEnd)
                return null;

        int rootValue = postorder[postEnd];
        TreeNode root = new TreeNode(rootValue);

        int k = 0;
        for (int i = inStart; i <=inEnd; i++) {
                if (inorder[i] == rootValue) {
                        k = i;
                        break;
                }
        }

        root.left = build(inorder, inStart, k - 1, postorder, postStart,
                        postStart + k - (inStart + 1));
        root.right = build(inorder, k + 1, inEnd, postorder, postStart + k- inStart, postEnd - 1);
        return root;
}
```

Binary Search Tree



Binary Search Tree

- All the sub-tree are BST
- All elements in left sub-tree is small than root
- All elements in Right sub-tree is larger than root
- If we do InOrder traversal, the result of BST is a sorted array

Binary Search Tree

- Find
- Add
- Remove

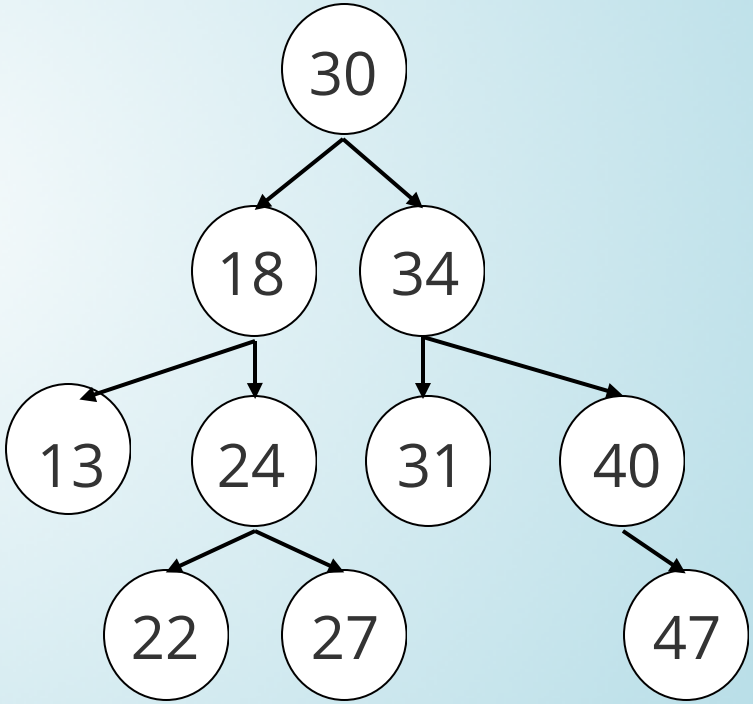
Binary Search Tree

Find 24

- 30 > 24 -> go to left
- 18 < 24 -> go to right
- find 24, return true

Find 42

- 30 < 42 -> go to right
- 34 < 42 -> go to right
- 40 < 42 -> go to right
- 47 > 42 -> go to left
- nothing on left, return false



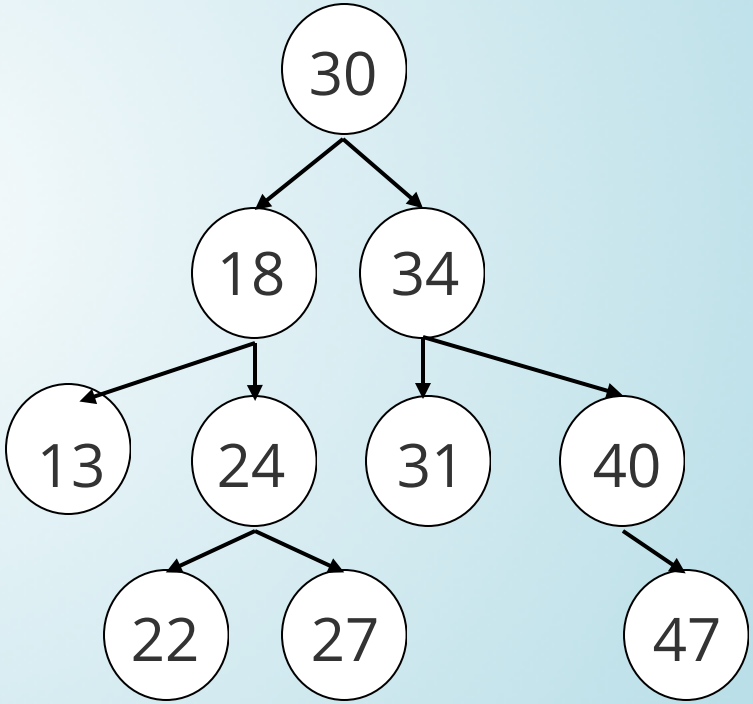
Binary Search Tree

```
public static boolean find(int value, Node root) {
        Node node = root;
        while(node != null) {
                if(node.data > value) {
                        node = node.leftNode;
                }
                else if(node.data < value) {
                        node = node.rightNode;
                }
                else return true;
        }
        return false;
}
```

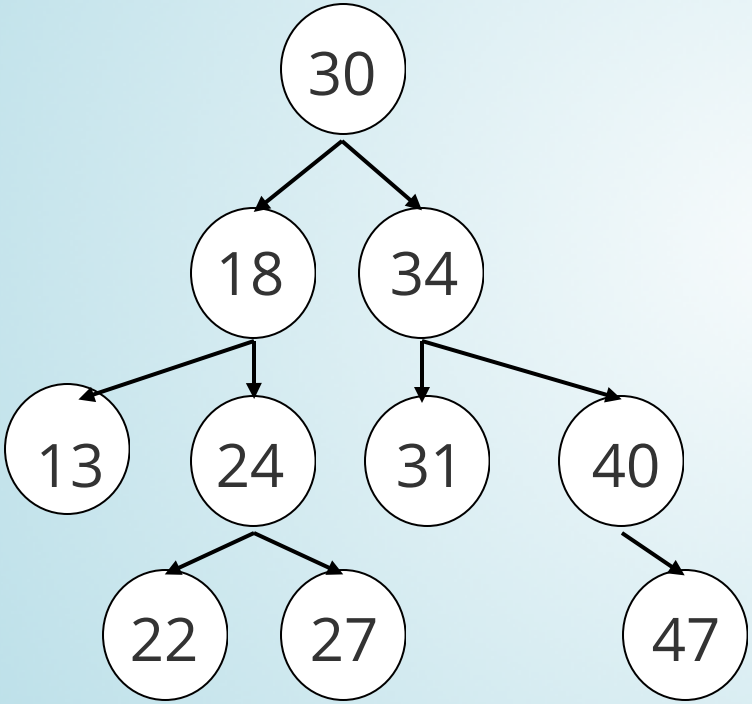
Binary Search Tree

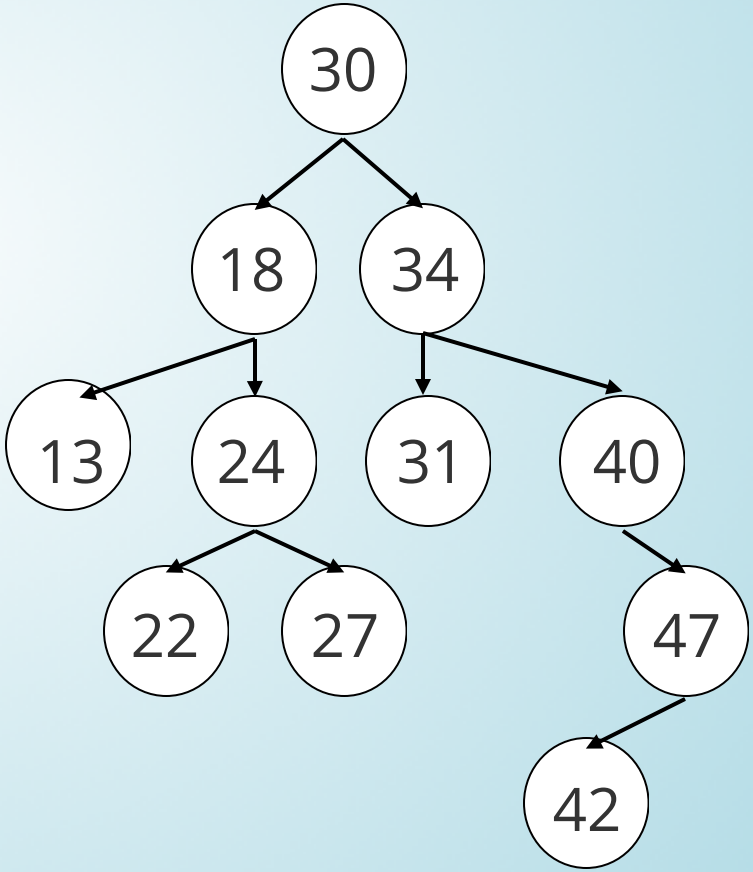
Add 42

- First we need to make sure if there is 42 in the tree already
- Find the place we put 42



Binary Search Tree





Binary Search Tree

```
public static boolean add(int value, Node root) {
        if(root == null) {
                root = new Node(value);
                return true;
        }
        Node node = root;
        while(node != null) {
                if(node.data > value) {
                        if(node.leftNode != null) {
                                node = node.leftNode;
                        }
                        else {
                                node.leftNode = new Node(value);
                                return true;
                        }
                }
                else if(node.data < value) {
                        if(node.rightNode != null) {
                                node = node.rightNode;
                        }
                        else {
                                node.rightNode = new Node(value);
                                return true;
                        }
                }
                else return false;
        }
        return false;
}
```

Binary Search Tree

What is the time complexity for finding and adding some element into the tree?

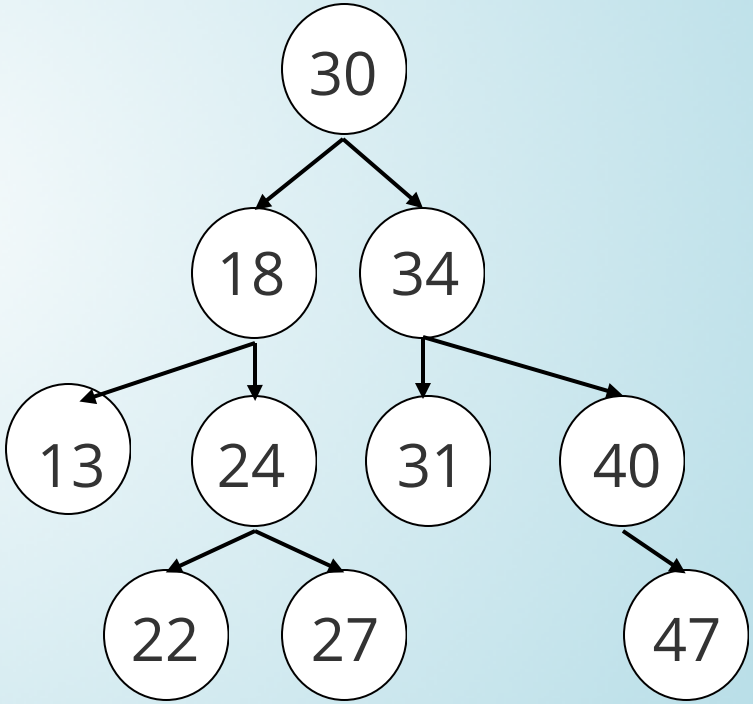
Binary Search Tree

Remove 27

- First we need to make sure if there is 27 in the tree already
- We need to remove 27

Remove 30

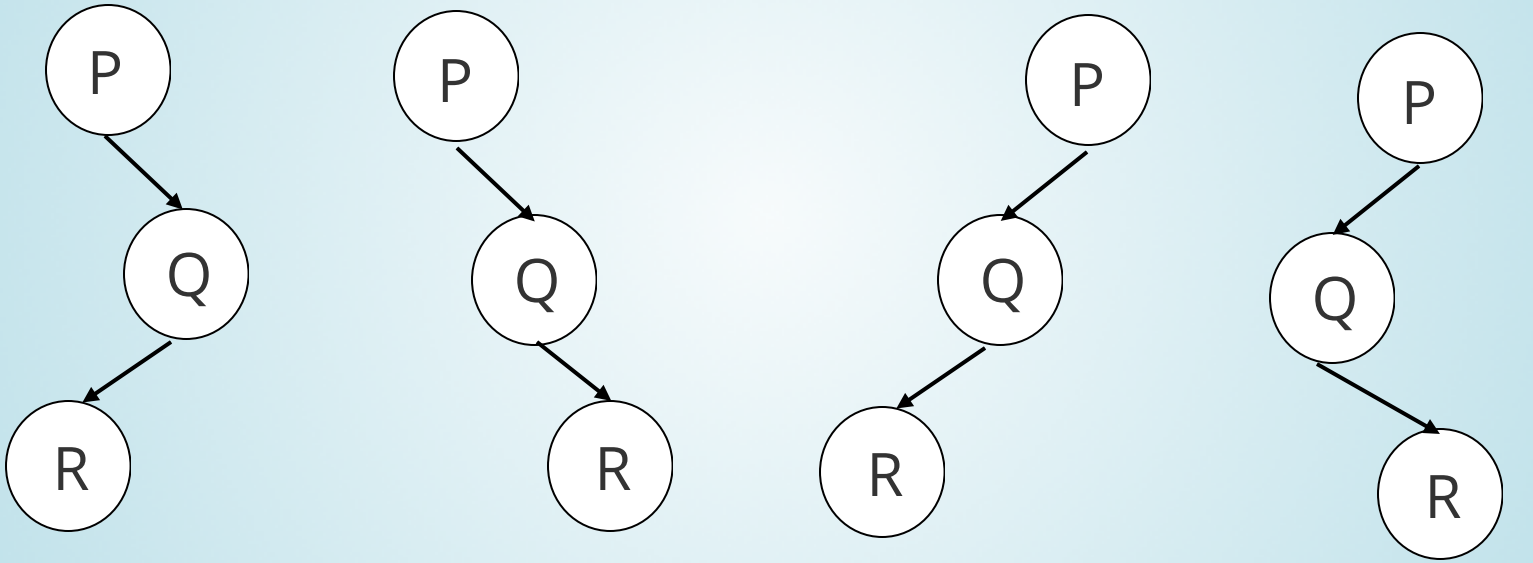
- How do we remove something that is not a leaf?



Binary Search Tree

- If Q is a leaf
- If Q has one child
  - if the child R is the right child
  - if the child R is the left child
- if Q has two children

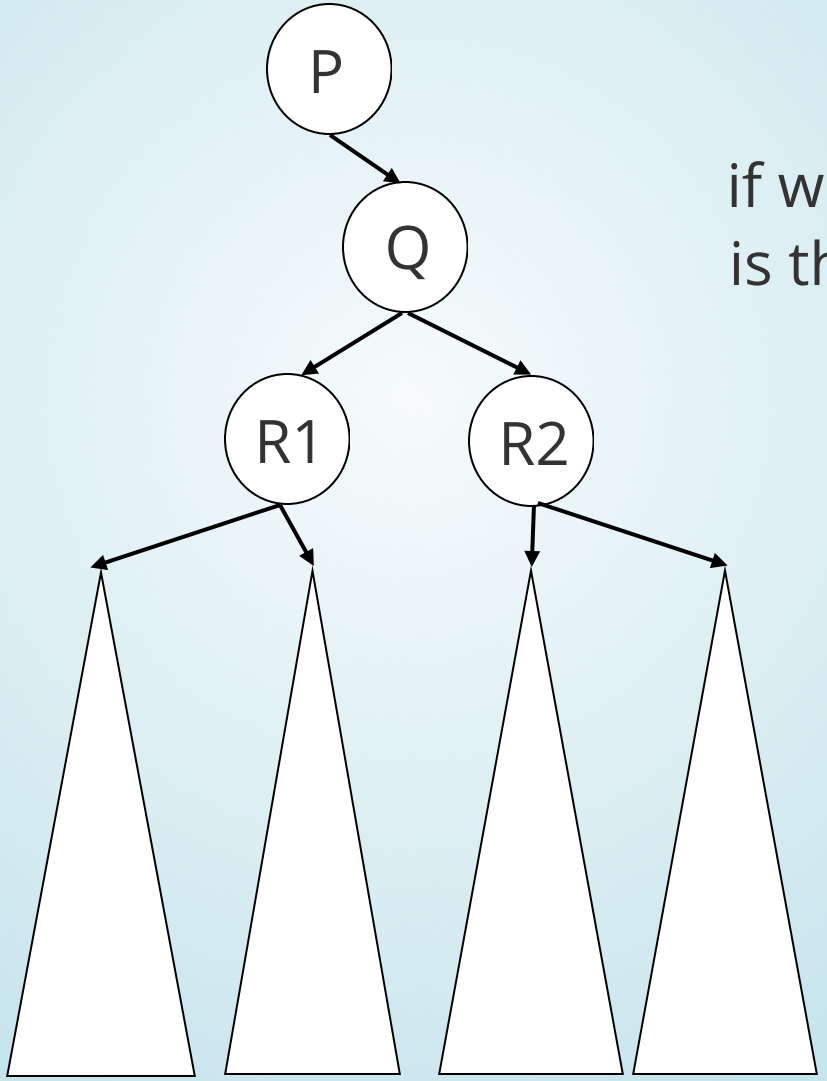
Binary Search Tree



P<R<Q P<Q<R R<Q<P Q<R<P

Just use R to replace Q

Binary Search Tree

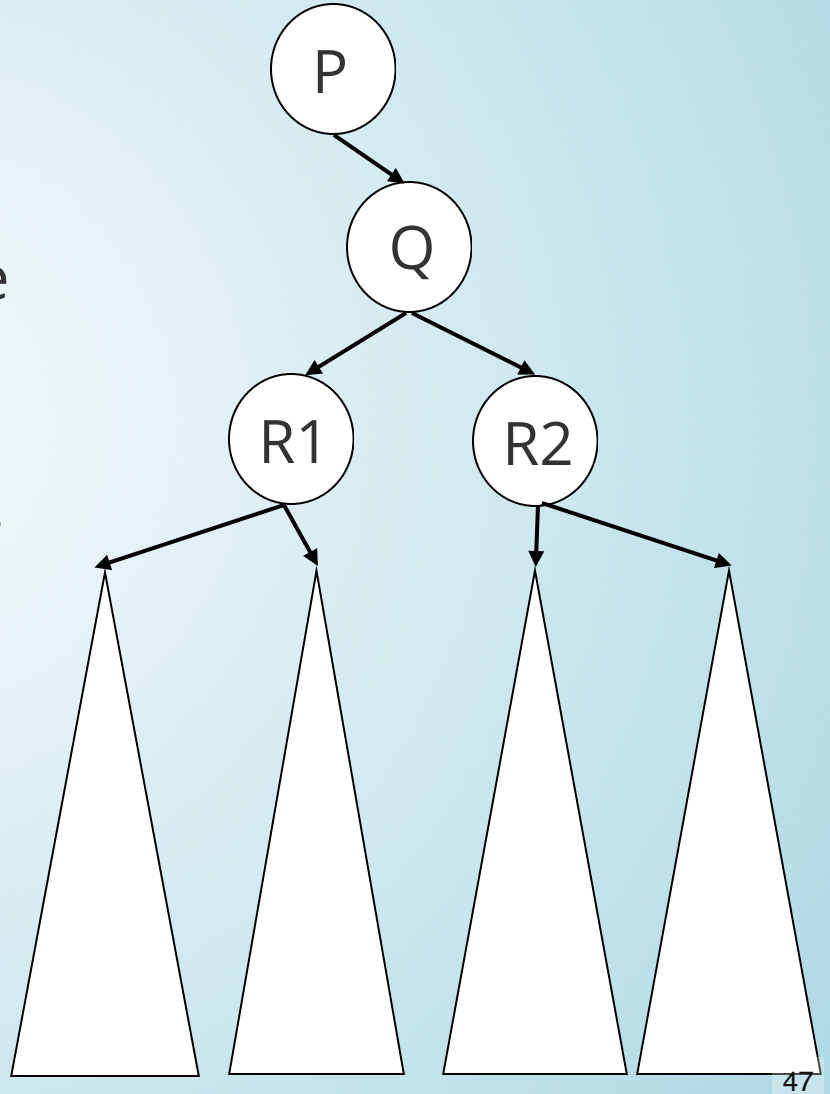


if we remove Q, which is the best substitute?

Binary Search Tree

Consider the inOrder traversal, the one just before Q and the one just after Q are the best candidates, since if they replace Q, when we do the inOrder again, the output is still an sorted array, which means the tree is still a BST.

So where are these two nodes?



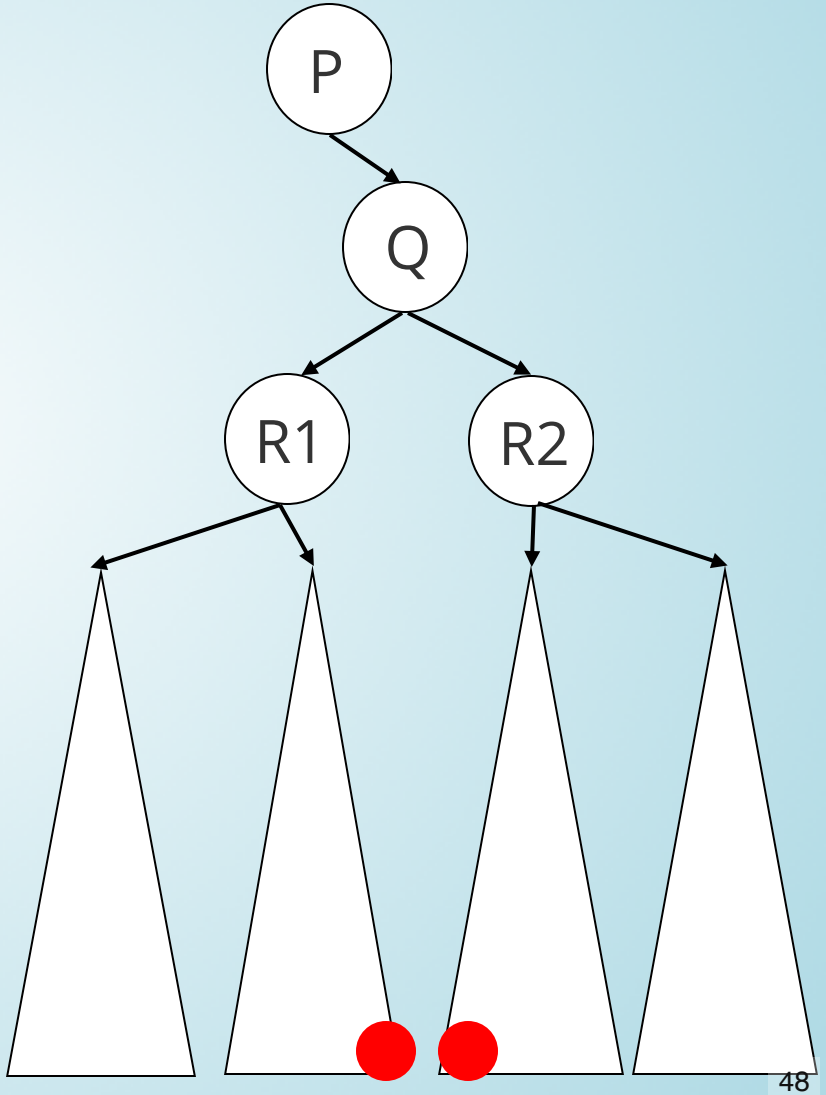
Binary Search Tree

So for the element before Q, it is the largest node in sub-tree R1

The element after Q, it is the smallest node in sub-tree R2

Does that node have child?
Canwe remove them directly

Maybe, but at most one, so we go back to condition 2

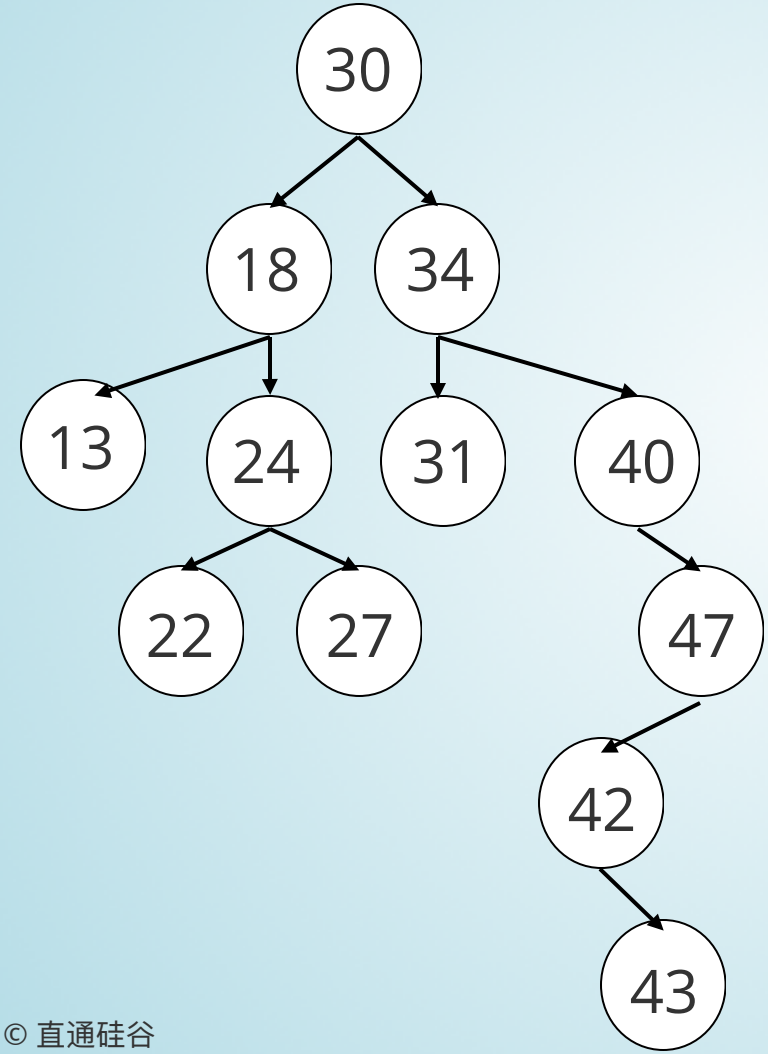


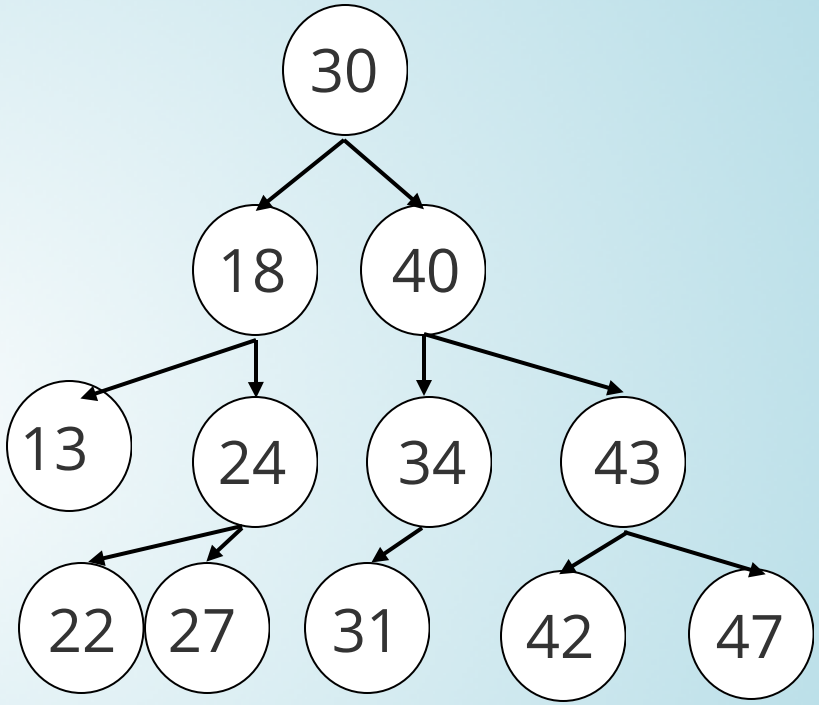
```
public static boolean remove(int value, Node root) {
        if(root == null) return false;
        if(root.data == value) {
                root = removeNode(root);
                return true;
        }
        Node node = root;
        while(node != null) {
                if(node.data > value) {
                        if(node.leftNode != null && node.leftNode.data != value) {
                                node = node.leftNode;
                        }
                        else if(node.leftNode == null) return false;
                        else {
                                node.leftNode = removeNode(node.leftNode);
                                return true;
                        }
                }
                else if(node.data < value) {
                        if(node.rightNode != null && node.rightNode.data != value) {
                                node = node.rightNode;
                        }
                        else if(node.rightNode == null) return false;
                        else {
                                node.rightNode = removeNode(node.rightNode);
                                return true;
                        }
                }
                else return false;
        }
        return false;
}
```

Binary Search Tree(cont)

```
public static Node removeNode(Node node) {
        if(node.leftNode == null && node.rightNode == null) {
                return null;
        }
        else if(node.leftNode == null) {
                return node.rightNode;
        }
        else if(node.rightNode == null) {
                return node.leftNode;
        }
        else {
                node.data = findAndRemove(node);
                return node;
        }
}

public static int findAndRemove(Node node) {
        int result;
        if(node.leftNode.rightNode == null) {
                result = node.leftNode.data;
                node.leftNode = node.leftNode.leftNode;
                return result;
        }
        node = node.leftNode;
        while(node.rightNode.rightNode != null) {
                node = node.rightNode;
        }
        result = node.rightNode.data;
        node.rightNode = node.rightNode.leftNode;
        return result;
}
```

Binary Search Tree





Binary Search Tree

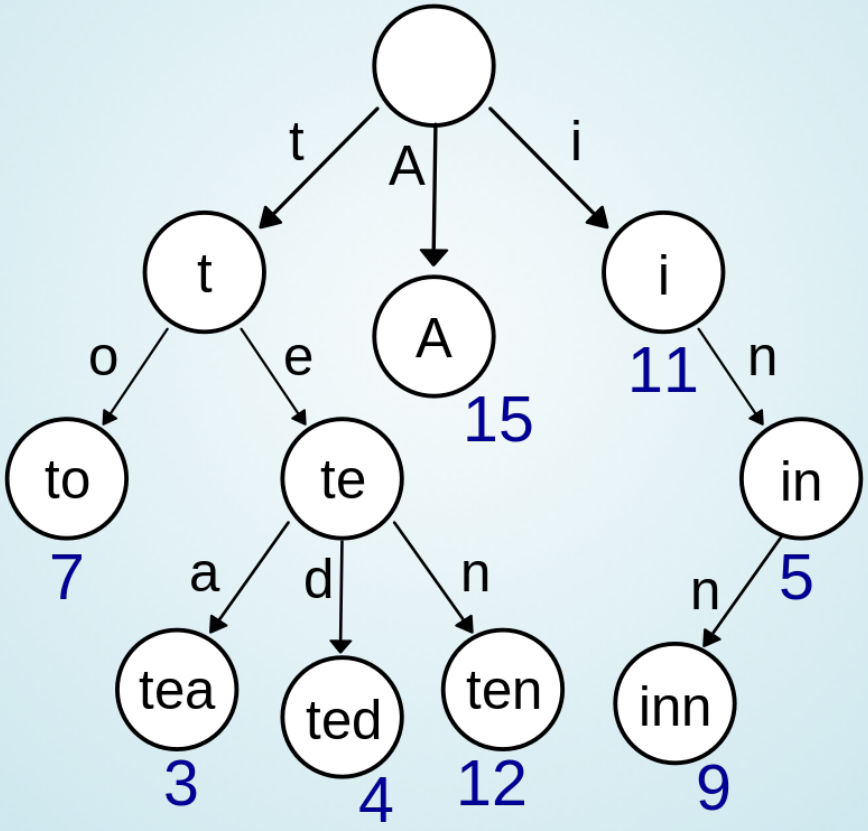
We know the search time is highly related to the height of the tree

If we keep add and remove elements in the tree, the tree will become unbalanced

So we have **Red-black tree** and **AVL tree**, they could use rotation and reconstruct to make the tree balance.

Trie Tree

Trie Tree



Trie Tree

In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.

What Trie Tree Can Help?

- It can store a dictionary in a good way without consuming much additional space
- It can easily find common prefix for two strings (this is very useful in typeahead)
- It can use easily find if a word is in the dictionary (maintain the order)

Implement Trie (Prefix Tree)

Implement a trie with insert, search, and startsWith methods

We only use lowercase 'a' to 'z'

How many children does a node have?

Implement Trie (Prefix Tree)

```
class TrieNode {
    // Initialize your data structure here.
        boolean isWord;
        char var;
        TrieNode[] children;
    public TrieNode() {
        children = new TrieNode[26];
        var = 0;
        isWord = false;
    }
}
```

Implement Trie (Prefix Tree)

```
public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Inserts a word into the trie.
    public void insert(String word) {
        if(word == null || word.length() == 0) return;
        TrieNode pNode = root;
        for(int i = 0; i < word.length(); i ++) {
                char c = word.charAt(i);
                int index = c - 'a';
                if(pNode.children[index] == null) {
                        TrieNode newNode = new TrieNode();
                        pNode.children[index] = newNode;
                }
                pNode = pNode.children[index];
        }
        pNode.isWord = true;
    }
}
```

Implement Trie (Prefix Tree)

```
// Returns if the word is in the trie.
public boolean search(String word) {
    TrieNode pNode = root;
    if(word == null || word.length() == 0) return true;
    for(int i = 0; i < word.length(); i ++) {
        int index = word.charAt(i) - 'a';
        pNode = pNode.children[index];
        if(pNode == null) return false;
    }
    return pNode.isWord;
}

// Returns if there is any word in the trie that starts with the given prefix.
public boolean startsWith(String prefix) {
    TrieNode pNode = root;
    if(prefix == null || prefix.length() == 0) return true;
    for(int i = 0; i < prefix.length(); i ++) {
        int index = prefix.charAt(i) - 'a';
        pNode = pNode.children[index];
        if(pNode == null) return false;
    }
    return true;
}
```

Complexity Analysis

In the insertion and finding notice that lowering a single level in the tree is done in constant time, and every time that the program lowers a single level in the tree, a single character is cut from the string.

we can conclude that every function lowers ***L*** levels on the tree and every time that the function lowers a level on the tree, it is done in constant time, then the insertion and finding of a word in a trie can be done in O(***L***) time.

The memory used in the tries depends on the methods to store the edges and how many words have prefixes in common.

Other Kinds of Tries

We used the tries to store words with lowercase letters, but the tries can be used to store many other things. We can use bits or bytes instead of lowercase letters and every data type can be stored in the tree, not only strings.

Homework

Validate Binary Search Tree

Unique Binary Search Trees II

Unique Binary Search Trees

Recover Binary Search Trees

Serialize and Deserialize Binary Tree