哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

数据结构
Data Structures

Chapter 8 Tree

Prof. Yitian Shao
School of Computer Science and Technology

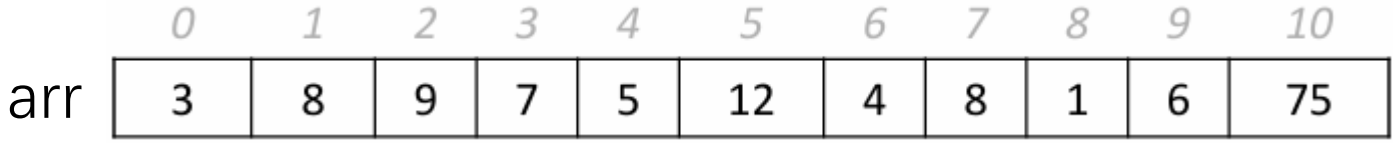
Tree

Course Overview

- Searching Complexity
- Binary Search
- Tree Structure
- Binary Search Trees
  - Definition and Anatomy
  - Traversal and Recursion
  - Searching
  - Modification

Array (Vector) Unsorted

- Store all the elements in an **unsorted** vector



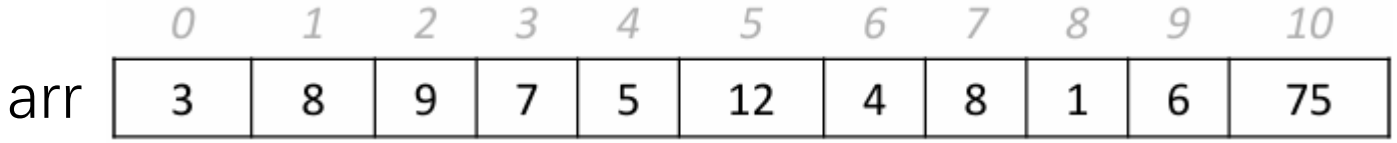
- Hand write C++ code to check whether this array contains a given integer number

```
class Solution {
public:
    bool Contain (vector<int> arr, int number) {

        // Implement your solution

    }
};
```

What is the time complexity?

Array (Vector) Unsorted

- Store all the elements in an **unsorted** vector



- Hand write C++ code to insert a given integer number to the k-th position

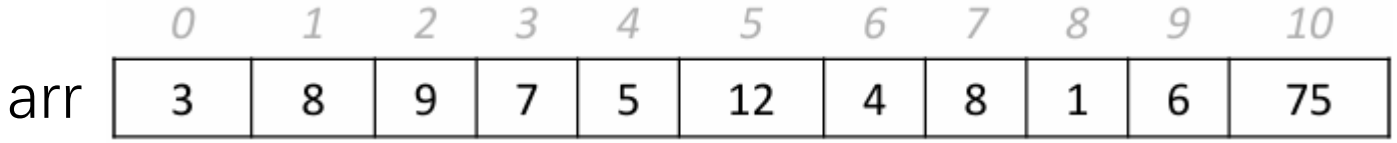
  (**Without using the insert()** function provided by the vector class)

```
class Solution {
public:
    vector<int> Insert (vector<int> arr, int number, int k) {

        // Implement your solution

    }
};
```

What is the time complexity?

Array (Vector) Unsorted

- Store all the elements in an **unsorted** vector



- Hand write C++ code to delete the k-th element from the vector

  (**Without using the erase()** function provided by the vector class)

```
class Solution {
public:
    vector<int> Delete (vector<int> arr, int k) {

        // Implement your solution

    }
};
```

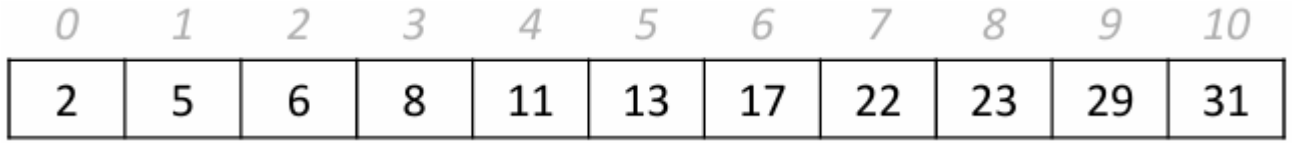
What is the time complexity?

Linked List Unsorted

- What about realizing the `Contain()` function for linked lists?
- What is the time complexity?

Array (Vector) Sorted

- What about realizing the `Contain()` functions for **sorted** array/vector?



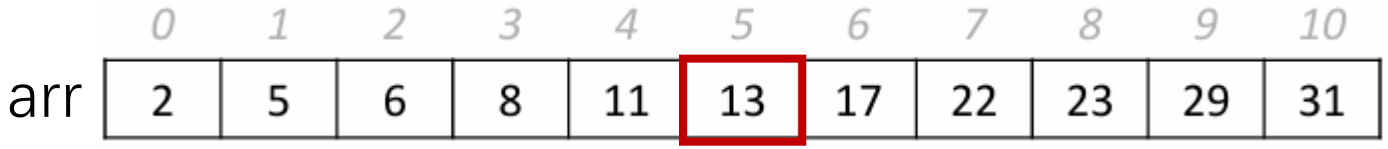
- For example, how can we find whether this sorted array contains the number 22 or not? What is the **time complexity** of your algorithm?
- **Can you find a more efficient way than O(n) time complexity?**

Binary Search

- Looping through elements one by one is slow [O(N)], and there is a faster way
- Idea:
  - Jump to the **middle element** (`number_of_element/2`):
  - if the middle is what we‘re looking for, we’re done!
  - if the middle is too small – we rule out the entire left side of elements smaller than the middle element
  - if the middle is too big – we rule out the entire right side of elements bigger than the middle element
  - Focus on the remaining elements and start over until we find the target or rule out everything

Binary Search – First Action

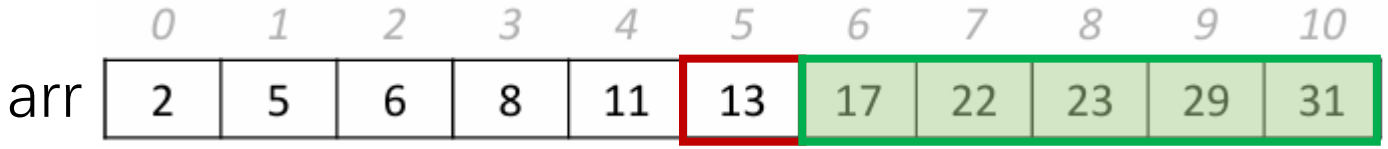
- For example, search for number 17



- The size of arr is 11, indexing from 0 to 10
- What is the middle element?
- arr[5]

Binary Search – First Action

- For example, search for number 17



- Compare our target, 22, against the middle element, arr[5] = 13
- Since 22 > 13 and we know that this array is sorted (ascendingly)
- Our target must be on the **right** half of the array!
- We rule out arr[0] to arr[5] and start over again for the remaining elements

Binary Search – Second Action

- For example, search for number 17

- New middle element of the remaining elements (index 6 – 10) is arr[8]
- Compare our target, 22, against the new middle element, arr[8] = 23
- Since 22 < 23 and we know that this array is sorted (ascendingly)
- Our target must be on the **left** half of the remaining array!
- We rule out arr[8] to arr[10] and start over again for the remaining elements

Binary Search – Third Action

- For example, search for number 17

- New middle element of the remaining elements (index 6, 7) is arr[6]
- Compare our target, 22, against the new middle element, arr[6] = 17
- Since 22 > 17 and we know that this array is sorted (ascendingly)
- Our target must be on the **right** half of the remaining array!
- We rule out arr[6] and start over again for the remaining elements

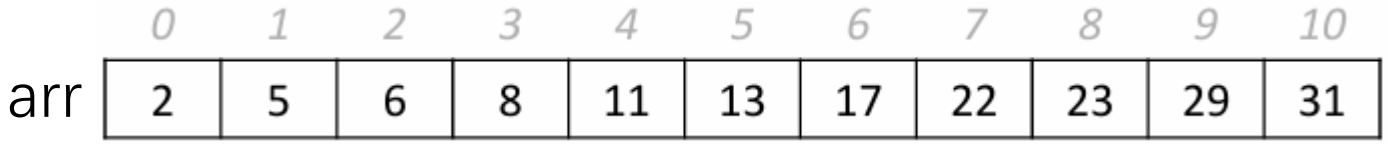
Binary Search – Fourth Action

- For example, search for number 17

- New middle element of the remaining elements (index 7) is arr[7]
- Compare our target, 22, against the new middle element, arr[7] = 22
- We find it! **Mission accomplished!**

Binary Search

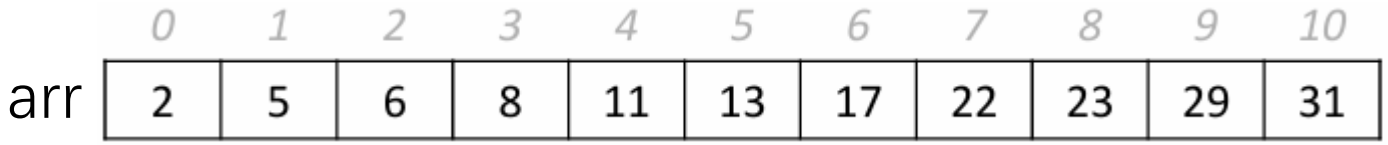
- For example, search for number 17



- The size of arr is 11
- We find our target with only **four** actions
- Actually, searching for any numbers in this array takes at most four actions, since **we can rule out the entire array in four actions**

Binary Search In-class Exercise

- Search for number 10



- Write down each actions

Binary Search Needs How Many Actions?

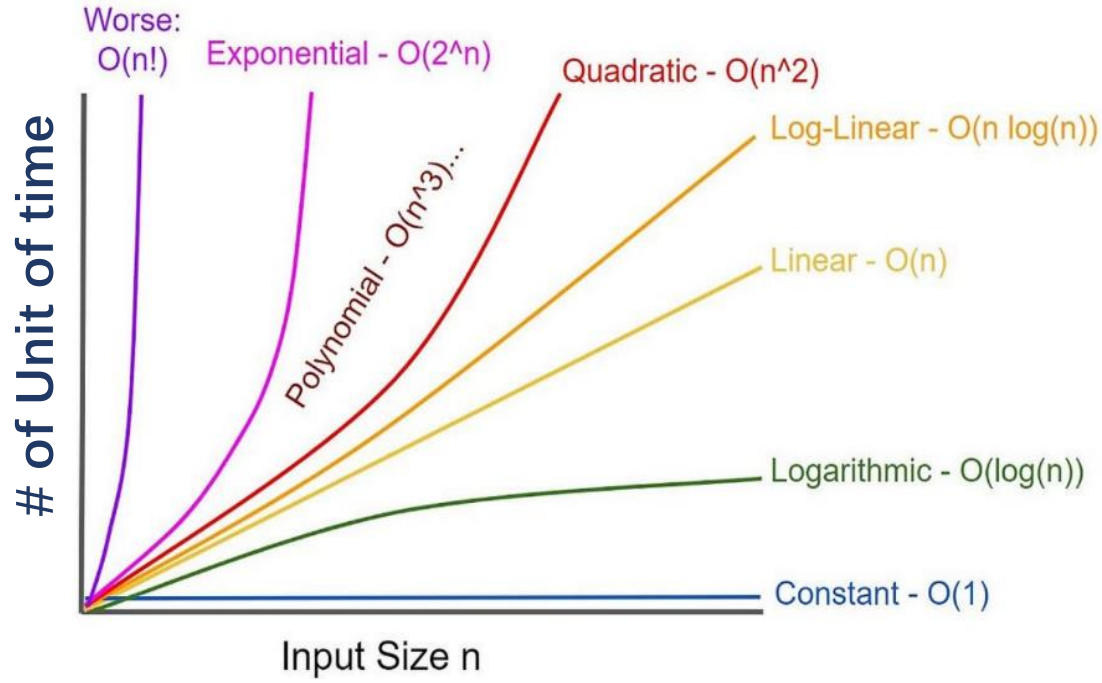
- Given an array of size N (number of element)
- For each action, we can rule out half of the array, eliminate `m = round_up[N/2]` of elements
- We repeat the actions until we get one element left
- If we reverse the process (given a very large N)
- Action K: 1 = 2^0 element remains; Action (K-1): 2 = 2^1 elemens remain; Action (K-2): 4 = 2^2 elements remain; Action (K-3): 8 = 2^3 elements remain; ⋯ ⋯ ⋯ ⋯ ⋯ ⋯ ⋯ ⋯ ; Action 1: 2^{K-1} elements remain;
- **What did you find? What is the relationship between action# and number of remaining elements?**

Binary Search Needs How Many Actions?

- If we reverse the process (given a very large N)
- Action (K-0): 1 = 2^0 element remains;
- Action (K-1): 2 = 2^1 elemens remain;
- Action (K-2): 4 = 2^2 elements remain;
- Action (K-3): 8 = 2^3 elements remain; … … … … … … … … … …;
- Action (K-(K-1)): 2^{K-1} elements remain;
- Before Action (K-(K-1)), **the original array should have in total 2^K elements**!
- Though **K actions**, we can rule out an array of **2^K elements**

Binary Search Efficiency

- Though **K actions**, we can rule out an array of 2^K **elements**
- **What it means for searching efficiency?**
- Given any sorted array of size **N**, we can find a target by taking at most $\log_2 N$ searching actions
- The time complexity for binary search is $O(\log_2 N)$



Array (Vector) Sorted

- What is the Big-O of **Contain()**?

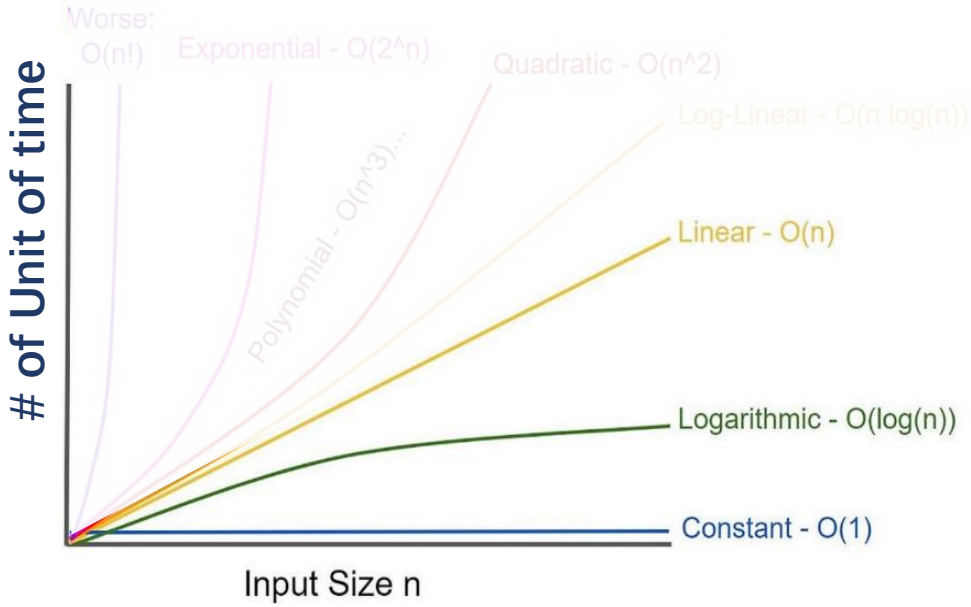
  $O(\log_2 N)$

- What is the Big-O of **Insert()** an element?

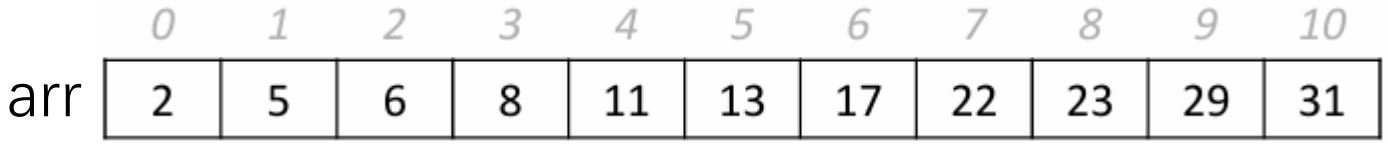
  O(N)

- What is the Big-O of **Delete()** an element?

  O(N)



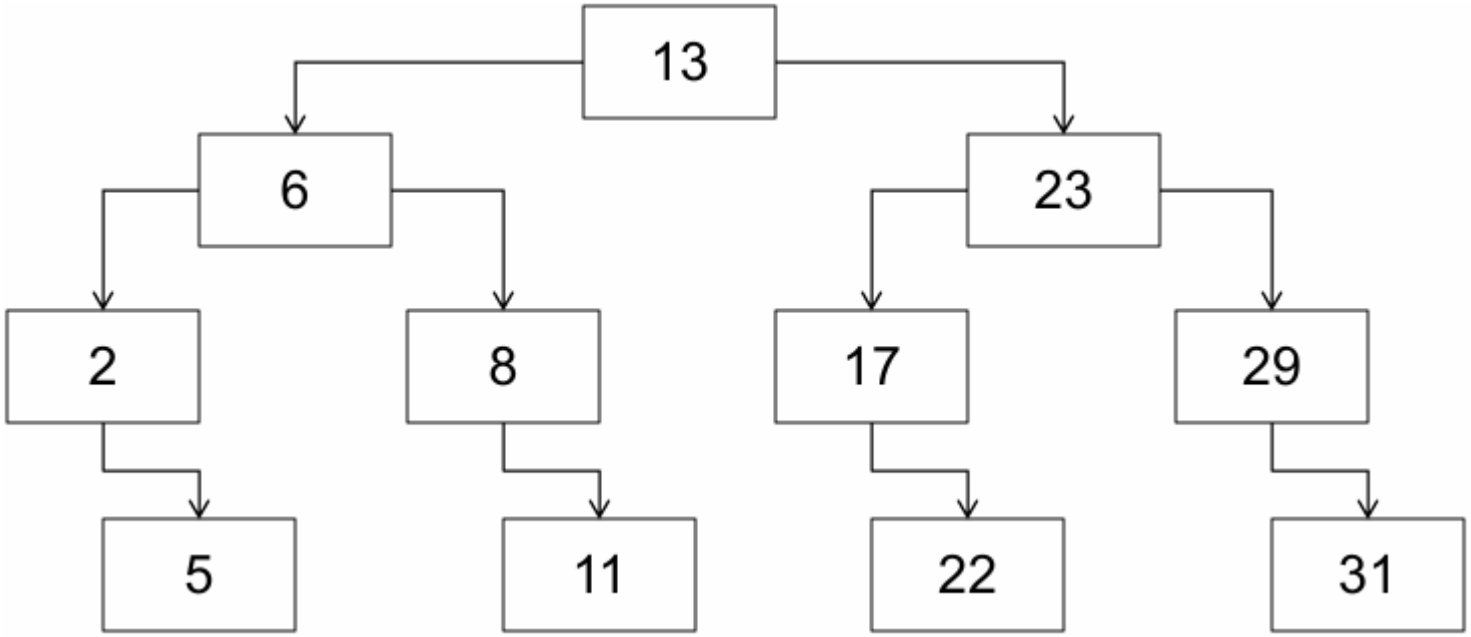
We Need a Better Data Structure



- Problem: a sorted array is still slow to insert into or remove from
- Our solution was a linked list– have each element connected to one other element
  - Easy to add/remove elements
  - Can't skip elements – need to go in order
- Maybe we can find some way to implement the jumps necessary for binary search..

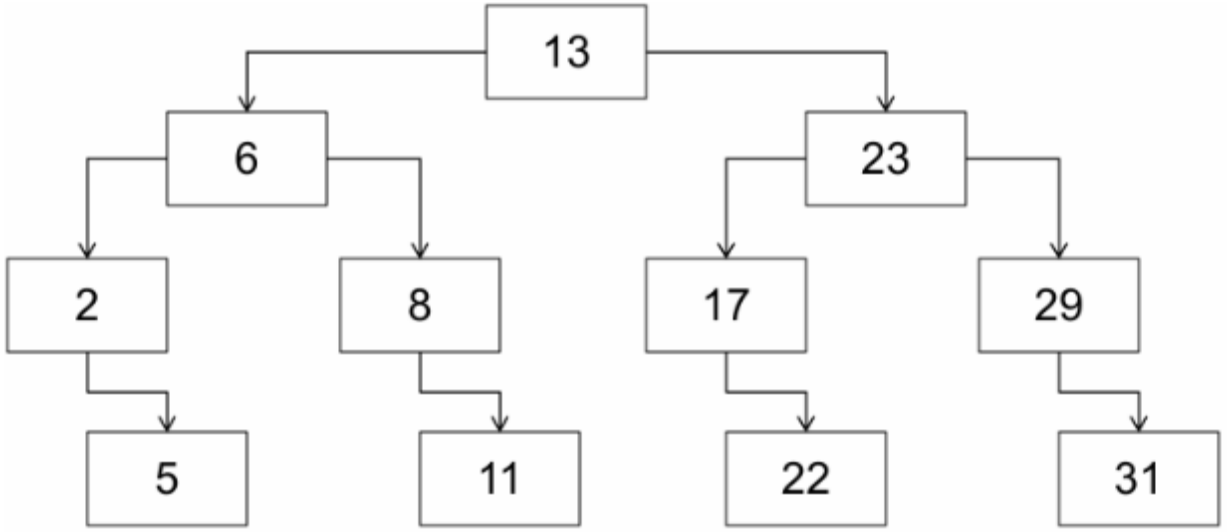
Tree Structure

	0	1	2	3	4	5	6	7	8	9	10
arr	2	5	6	8	11	13	17	22	23	29	31



Tree Structure

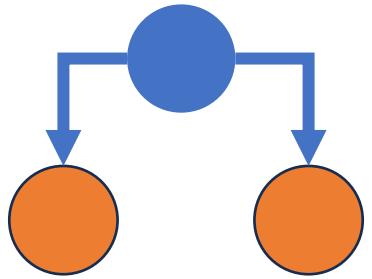
- In this structure, we always jump to one of two elements in binary search (depending on if the element we're looking at is bigger or smaller)
- What if we had a **Linked List** with **each node stores two pointers**, allowing us to make those jumps quickly?



Binary Search Tree (BST)

- A **tree** is a data structure where each element (**parent**) stores two or more pointers to other elements (its **children**)
  - A doubly-linked list doesn't count because, just like outside of computer science, a child can not be its own ancestor
- Each node in a **binary tree** has **two** pointers
  - Some of these pointers may be **nullptr**

  (just like in a linked list)



Binary Search Tree (BST)

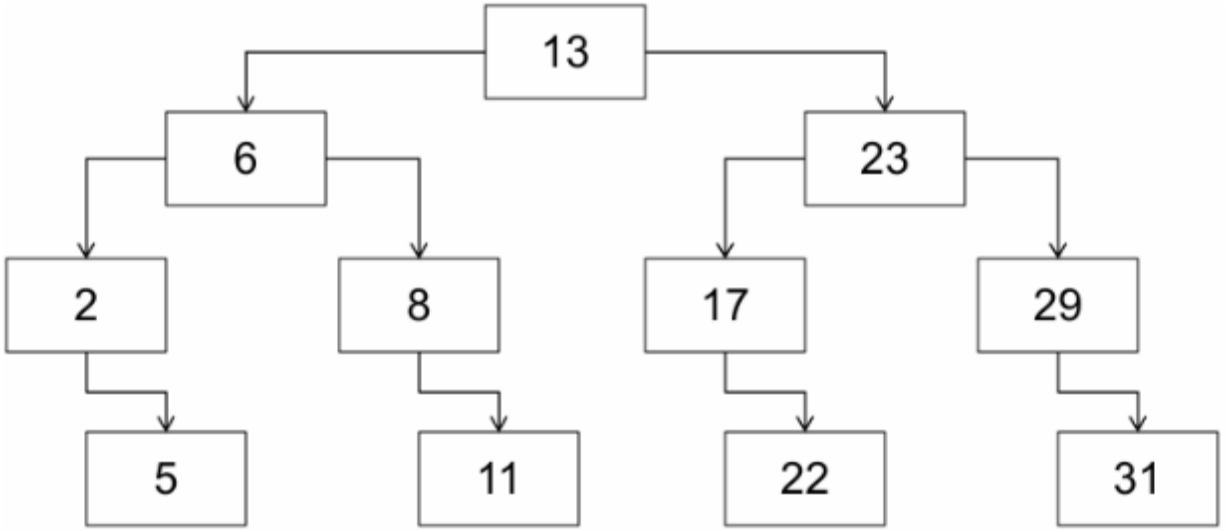
- A **binary search tree (BST)** is a binary tree with special ordering properties that make it easy to do binary search
- Similar to a Linked List:
  - Each element in its own block of memory
  - Have to travel through pointers (can't skip "generations")

(Binary) TreeNode

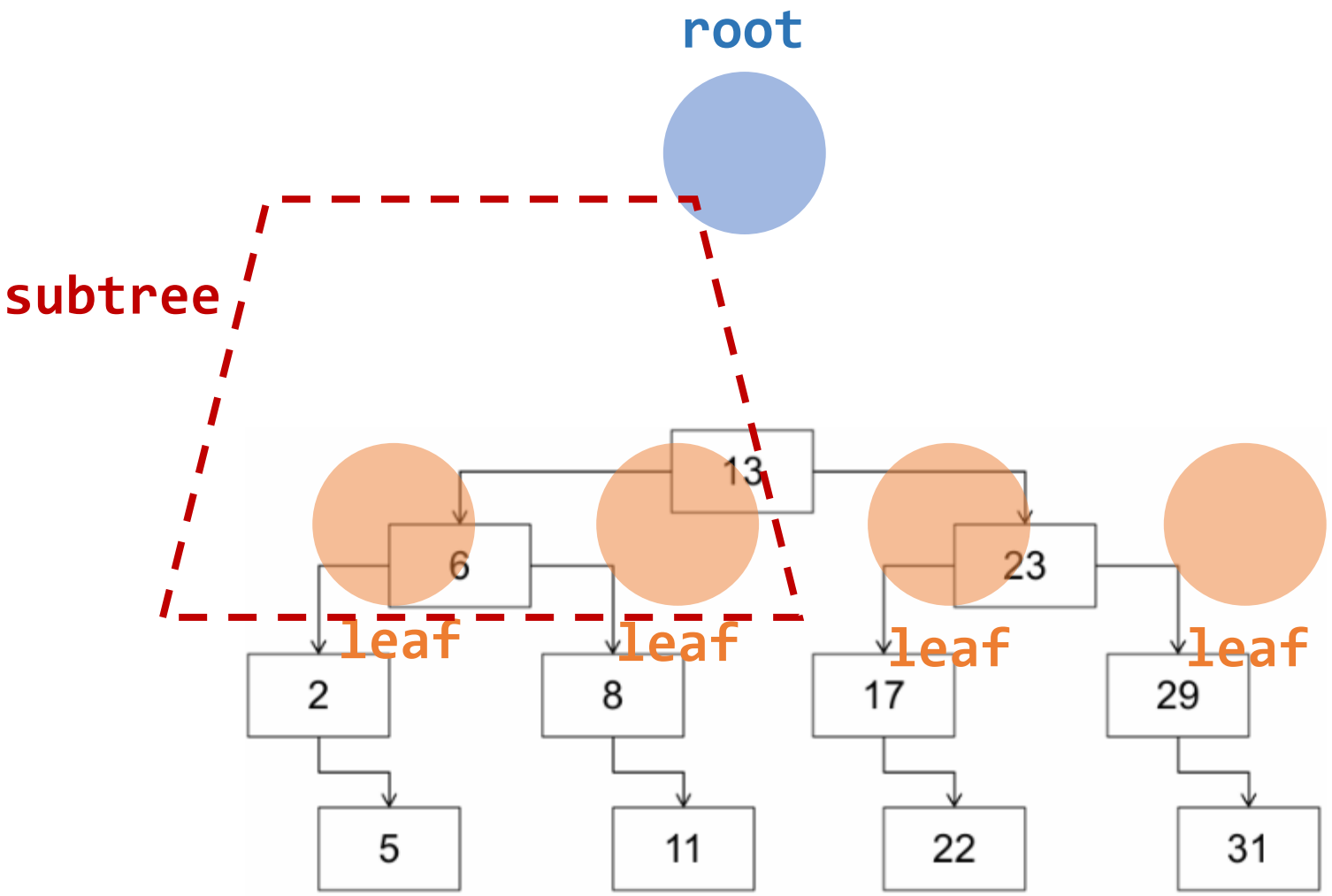
```
struct TreeNode {
    int data; // assume that the tree stores integers
    TreeNode* left;
    TreeNode* right;
 };
```

Binary Search Tree (BST)

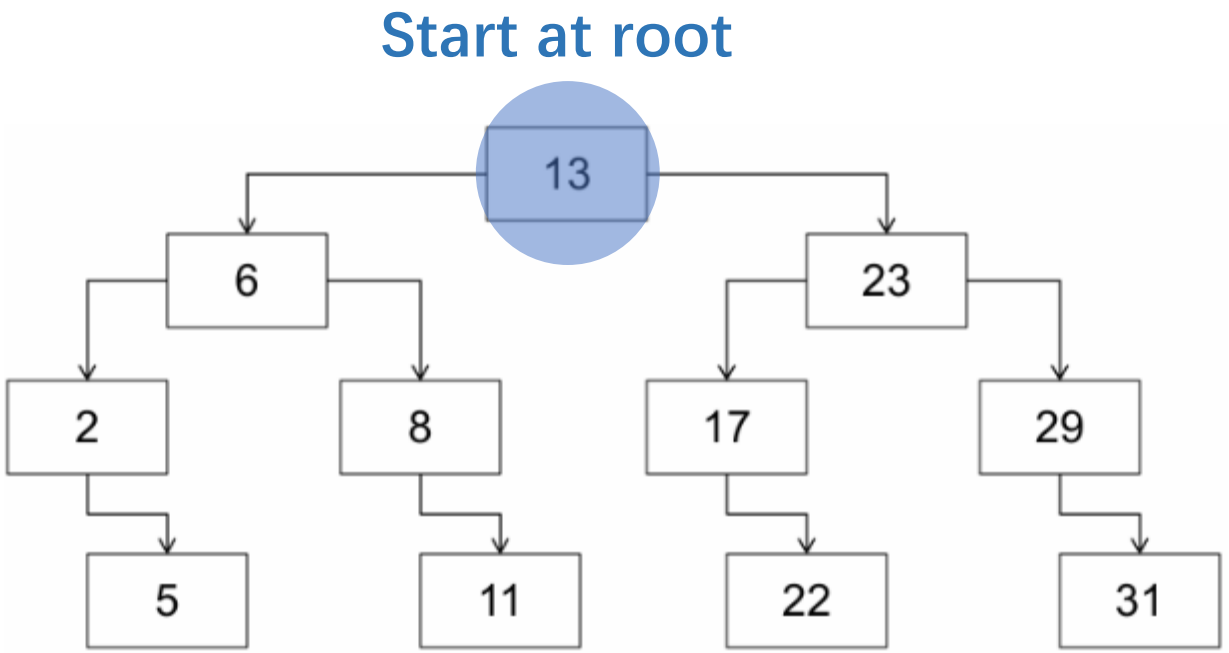
- A binary search tree has the following property:
  - All elements to the left of an element are smaller than that element
  - All elements to the right of an element are bigger than that element
  - Just like our sorted array!



Tree Anatomy

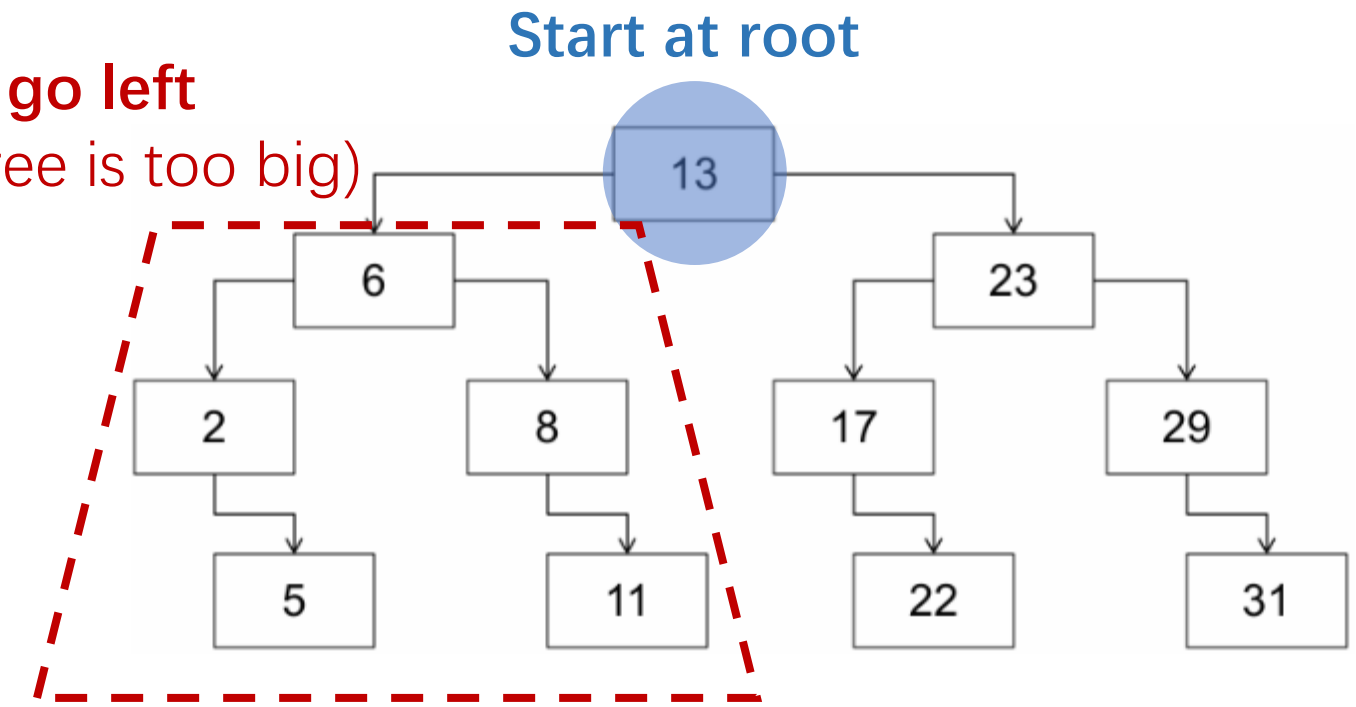


BST Contain()

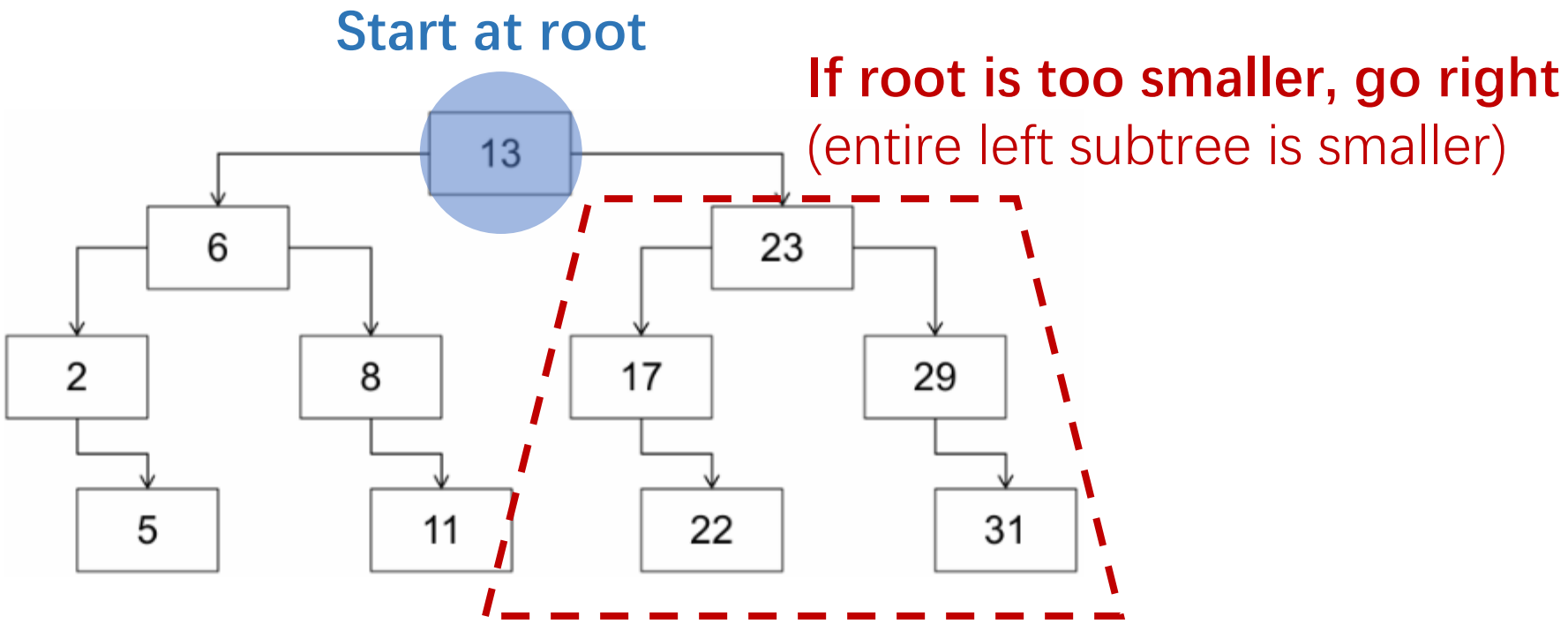


BST Contain()

If root is bigger, go left
(entire right subtree is too big)



BST Contain()

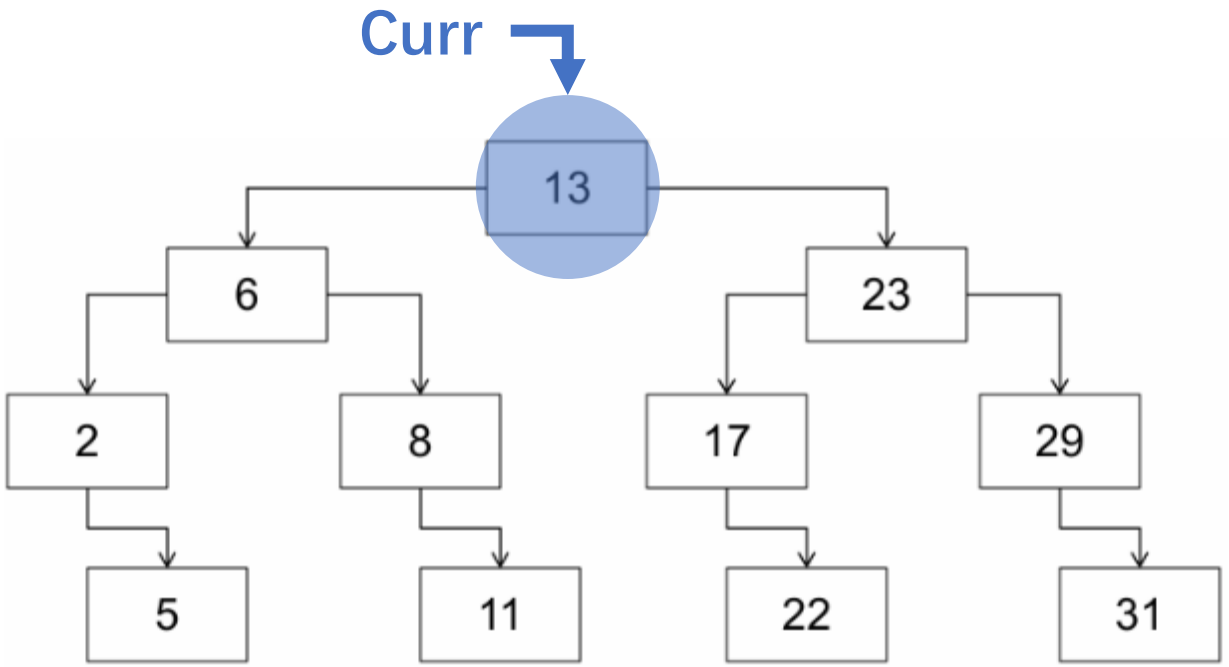

Start at root
If root is too smaller, go right
(entire left subtree is smaller)
13
6
23
2
8
17
29
5
11
22
31

Trees and Recursion

- Trees are fundamentally recursive (**subtrees are smaller trees**)
- Start at root:
  - If root is bigger, go left
  - If root is smaller, go right
- Go to the subtree and start over

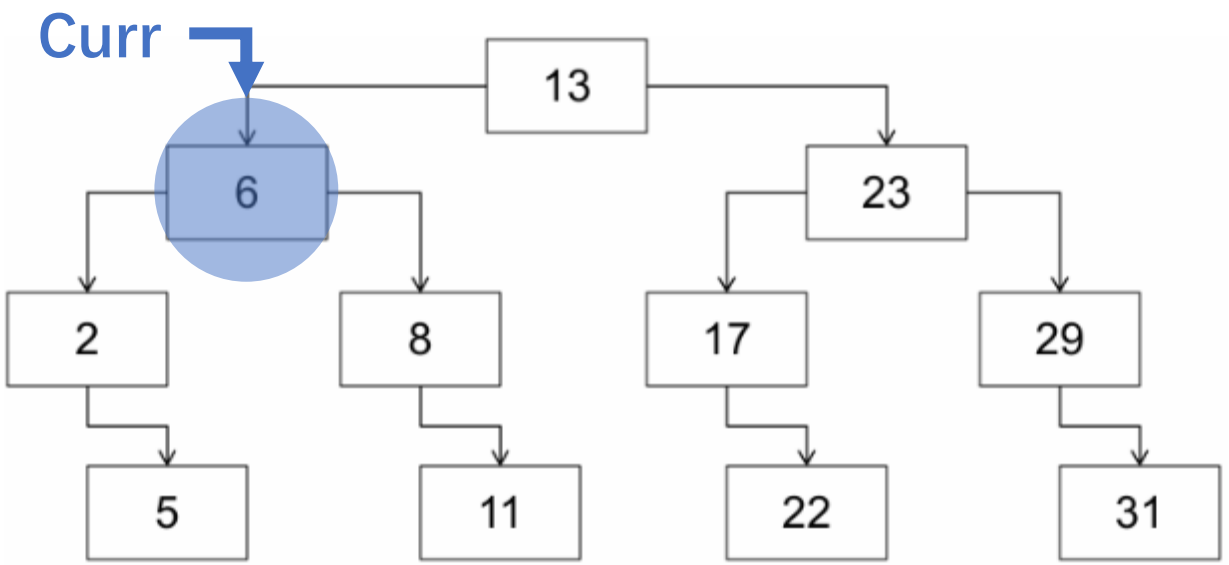
BST Contain()

- Search for 5



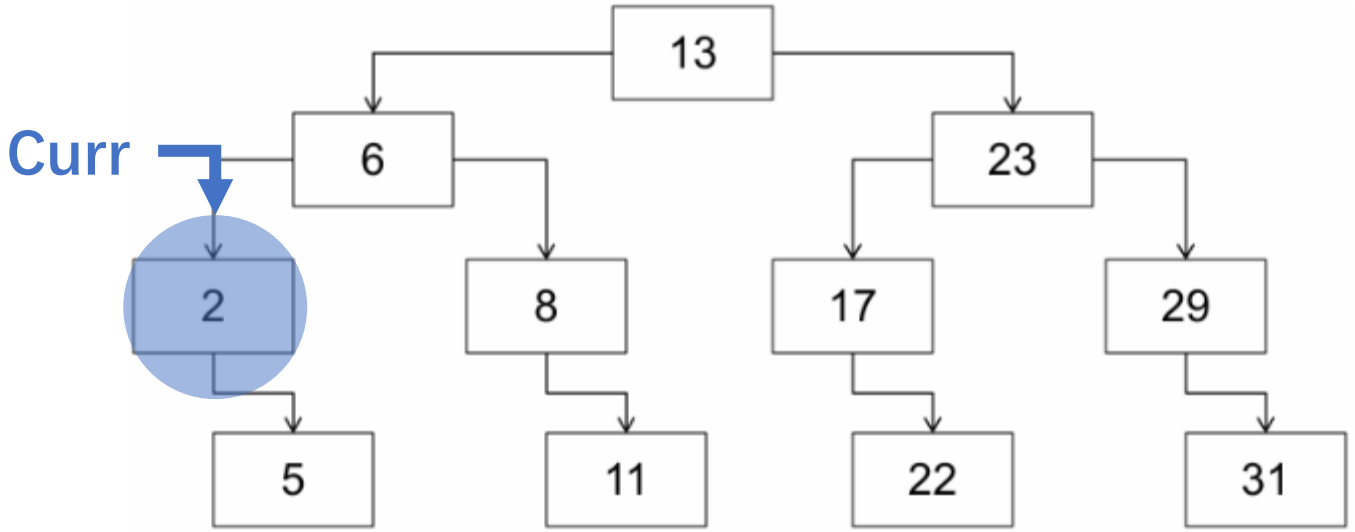
BST Contain()

- Search for 5



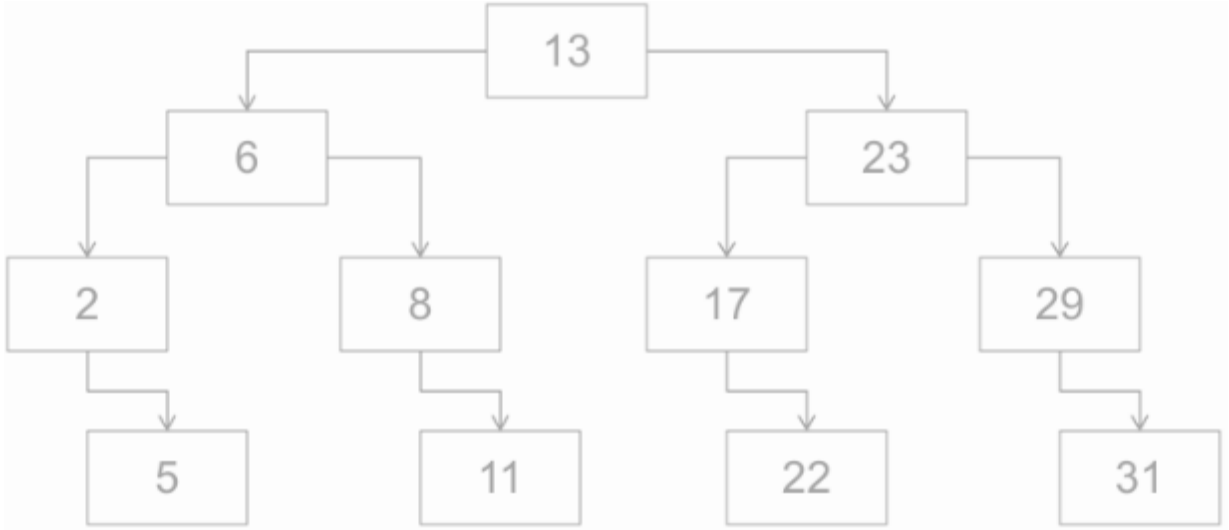
BST Contain()

- Search for 5



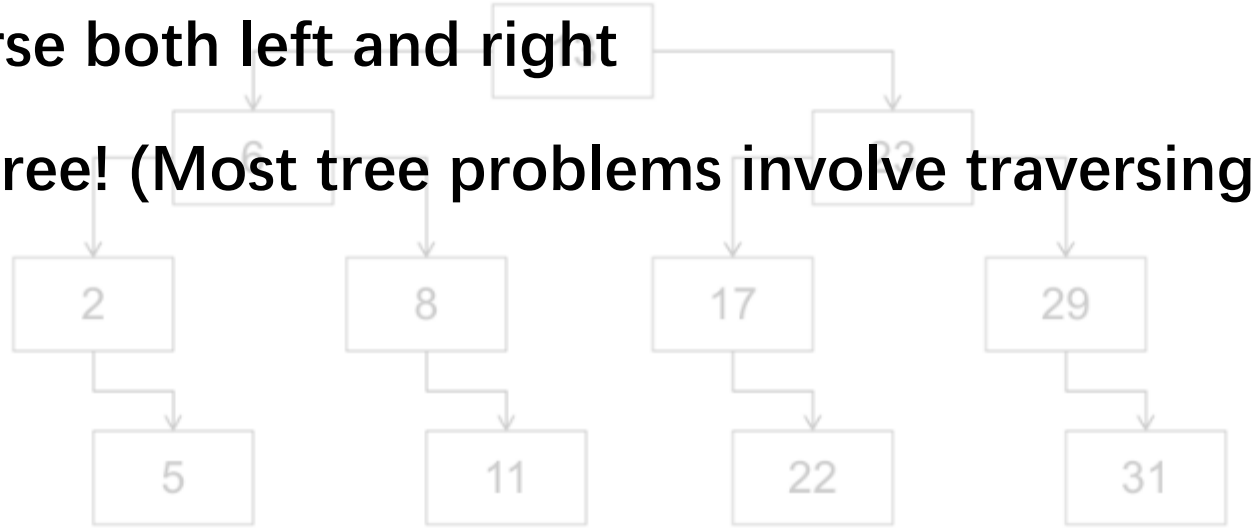
Printing Trees

- **How would we print a tree?**



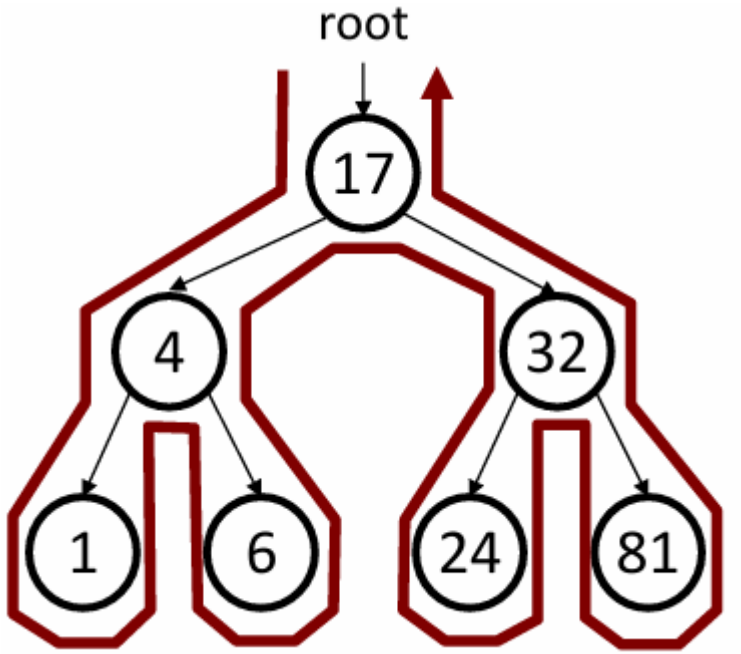
Printing Trees

- How would we print a tree?
- Need to recurse both left and right
- Traverse the tree! (Most tree problems involve traversing the tree)



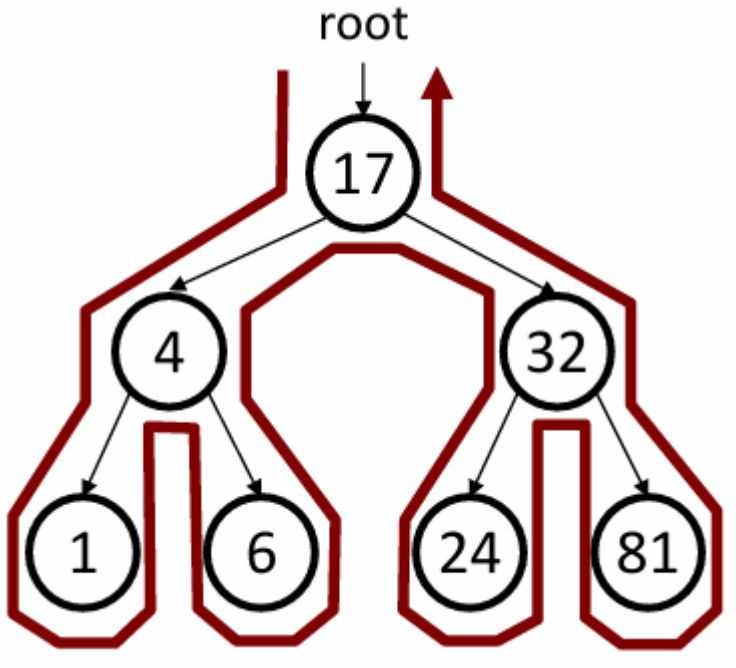
Traversal Methods

- To quickly generate a traversal:
  - Trace a path **counterclockwise**
  - As you pass a node on the proper side, process it
  - **pre-order**: **root** prioritized (Root – Left - Right)
    17→4→1→6→32→24→81
  - **in-order**: **leftmost** prioritized (Left - Root - Right)
    1→4→6→17→24→32→81
  - **post-order**: **child** prioritized (Left - Right - Root)
    1→6→4→24→81→32→17



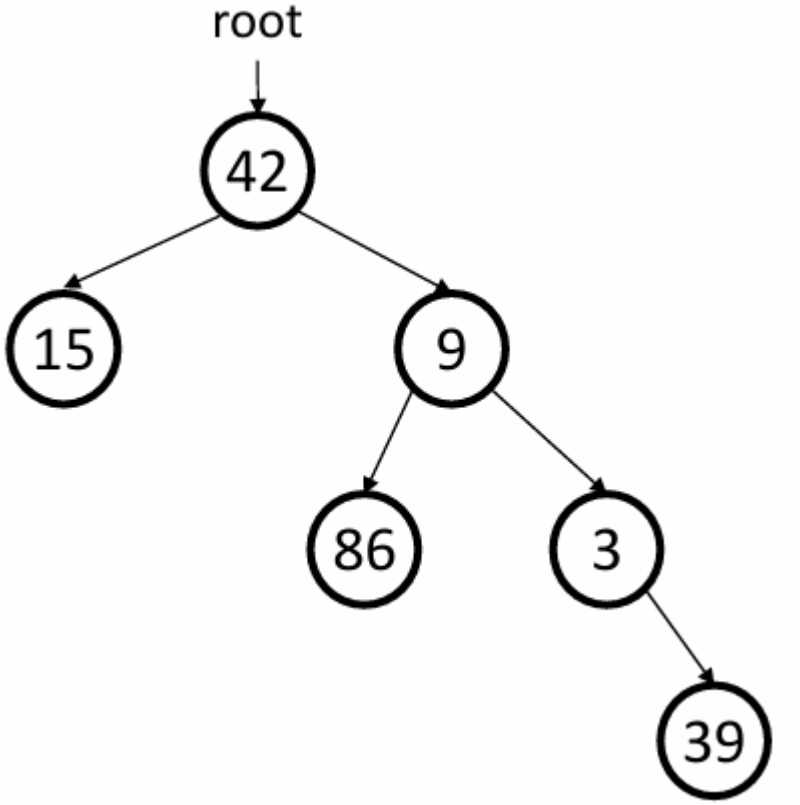
Traversal Methods

- **pre-order**: Can be used to **create** a copy of the tree
  17→4→1→6→32→24→81
- **in-order**: In the case of binary search trees, In-order traversal gives nodes in **non-decreasing order**
  1→4→6→17→24→32→81
- **post-order**: Can be used to **delete** the tree
  1→6→4→24→81→32→17



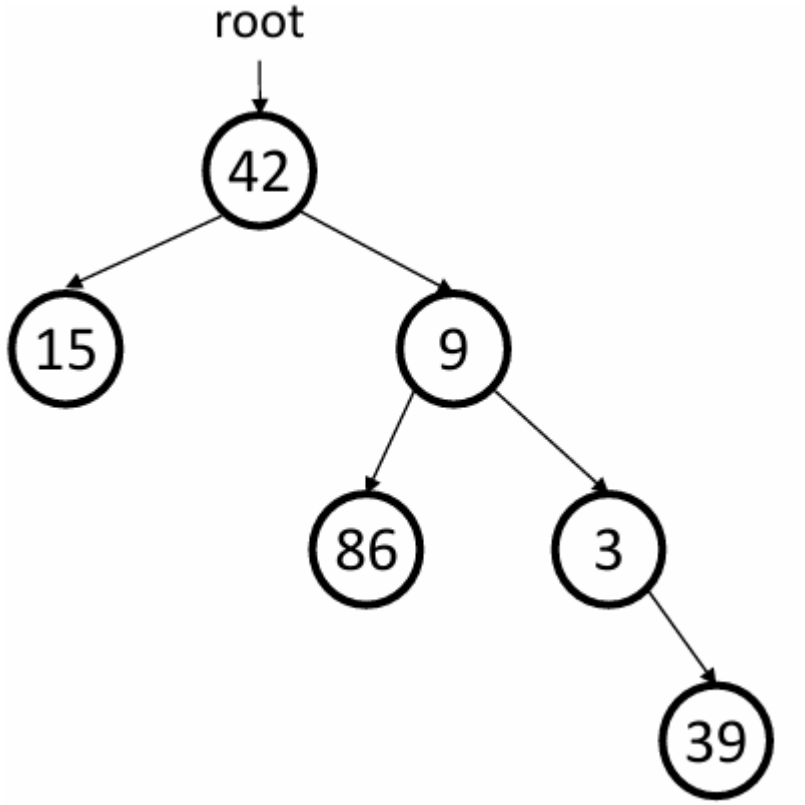
Traversal Exercise

- Give pre-, in-, and post-order traversals for the following tree:



Traversal Exercise – Solution

- Give pre-, in-, and post-order traversals for the following tree:



pre-order: **root** side prioritized

42→15→9→86→3→39

in-order: **leftmost** prioritized

15→42→86→9→3→39

post-order: **child** side prioritized

15→86→39→3→9→42

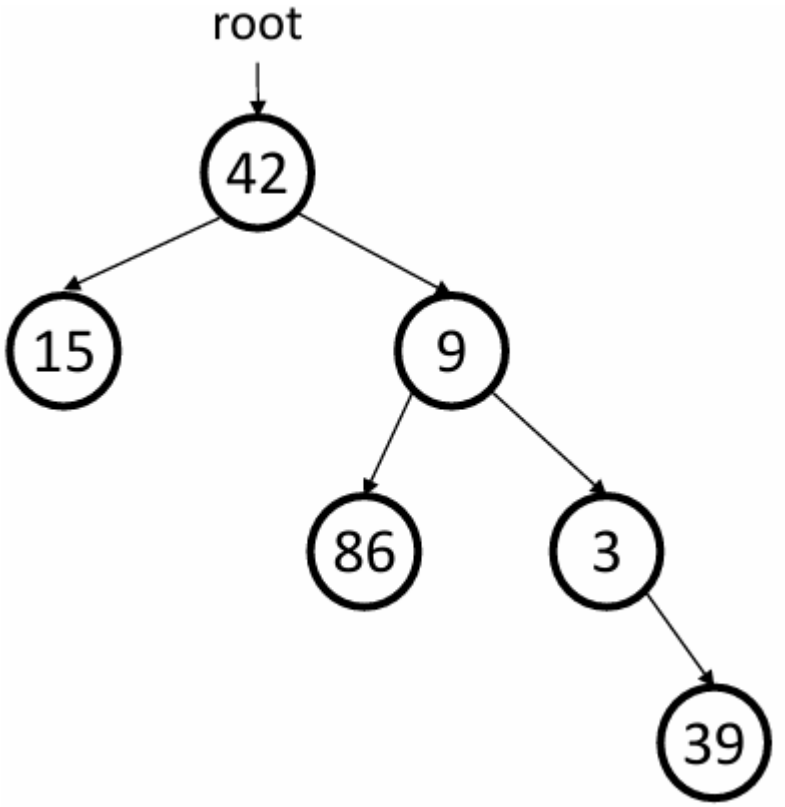
Print Traversal

- What is printed after traversal the tree if place the following line of code:

```
cout << node->data << endl;
```

at locations A - D?

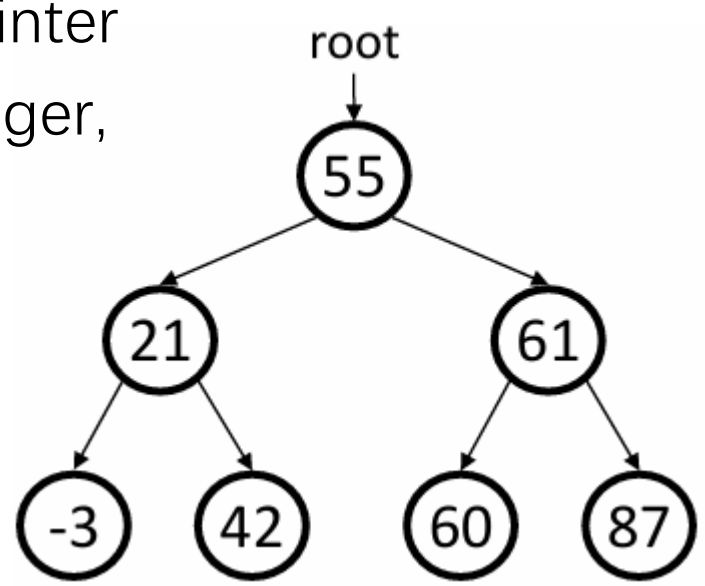
```
void printNode(TreeNode* node) {
    // Place the code at (A)
        if (node != nullptr) {
          // Place the code at (B)
          printNode(node->left);
          // Place the code at (C)
          printNode(node->right);
          // Place the code at (D)
        }
  }
```



In-class Exercise: Tree Contain()

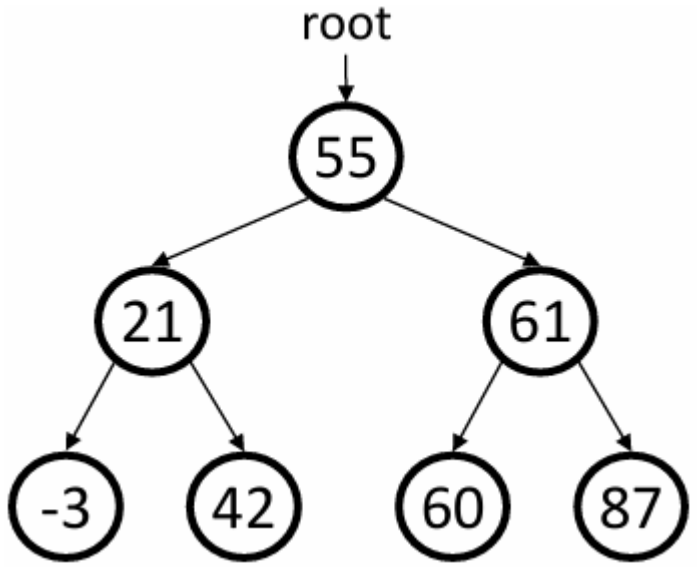
- Write a function contains that accepts a tree node pointer as its parameter and searches the BST for a given integer, returning true if found and false if not

```
contain(root, 87) → true
contain(root, 60) → true
contain(root, 63) → false
contain(root, 44) → false
```



In-class Exercise: Tree Contain()

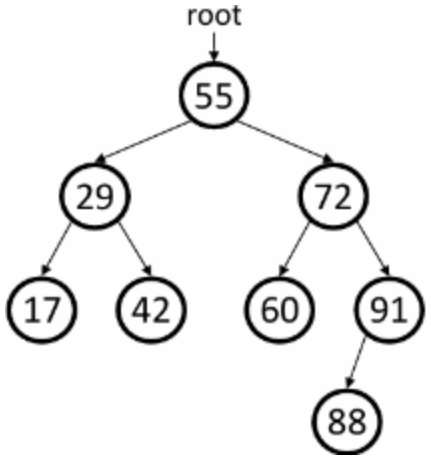
```
// Returns whether this BST contains the given integer.
bool contain(TreeNode* node, int value) {
    if (node == nullptr) {
       return false;  // base case: not found here
    }
    else if (node->data == value) {
       return true;  // base case: found here
    }
    else if (node->data > value) {
       return contain(node->left, value);
    }
    else {  // root->data < value
       return contain(node->right, value);
    }
}
```



getMin() and getMax()

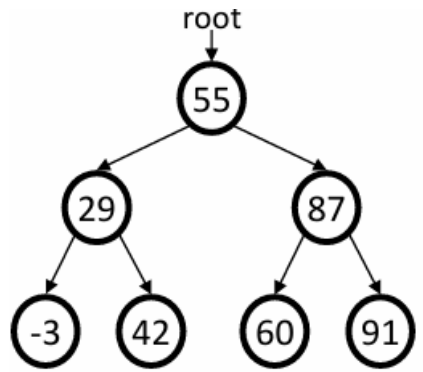
- **Sorted arrays** can find the smallest or largest element in O(1) time **(how?)**

- How could we get the smallest or largest values in a binary search tree?

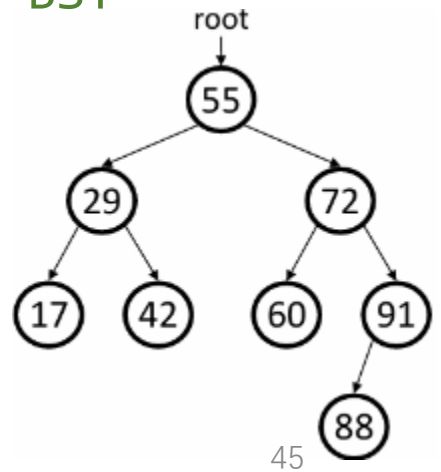


getMin() and getMax()

```
int getMin(TreeNode* root) { // Returns the minimum value from a BST
    if (root->left == nullptr) {
      return root->data;
    } else {
      return getMin(root->left);
    }
 }
```

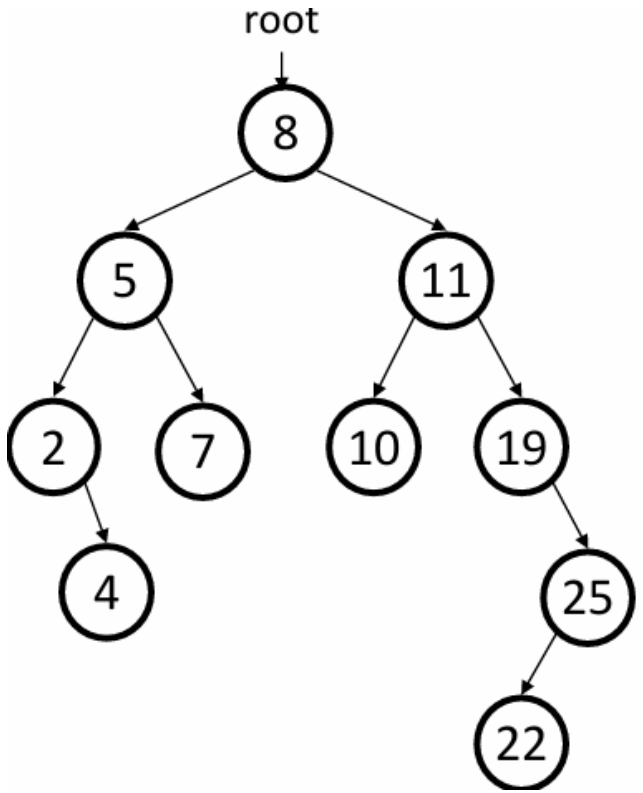



```
int getMax(TreeNode* root) { // Returns the maximum value from a BST
    if (root->right == nullptr) {
      return root->data;
    } else {
      return getMax(root->right);
    }
 }
```



Add node to a BST

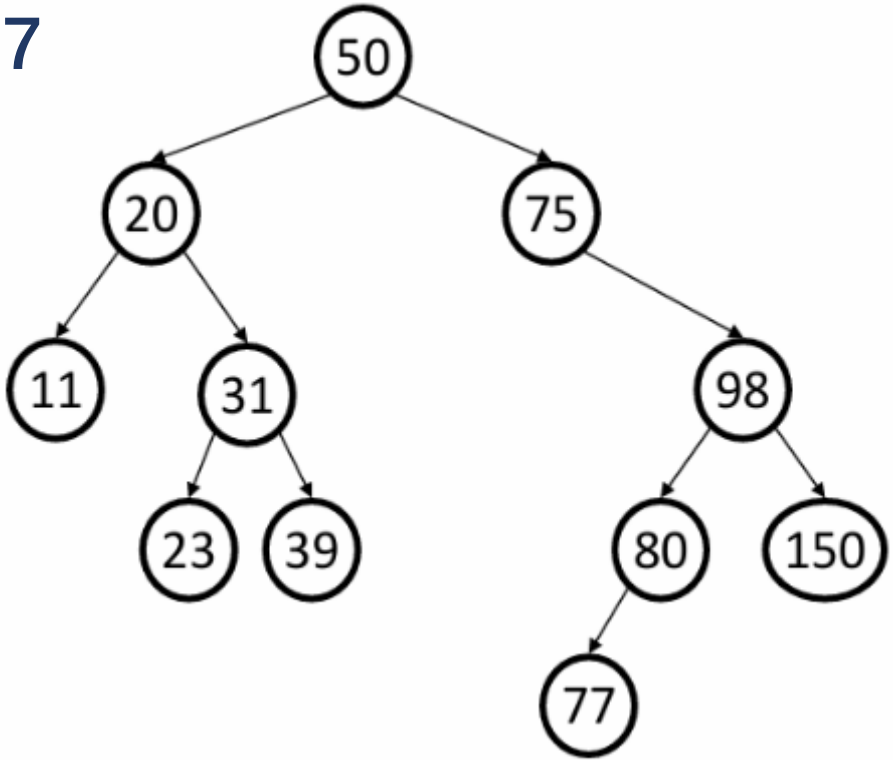
- If we want to add new values to the BST below
  - Where should the value **14** be added?
  - Where should **3** be added?
  - If the tree is empty, where should a new value be added?



Add node to a BST

- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:

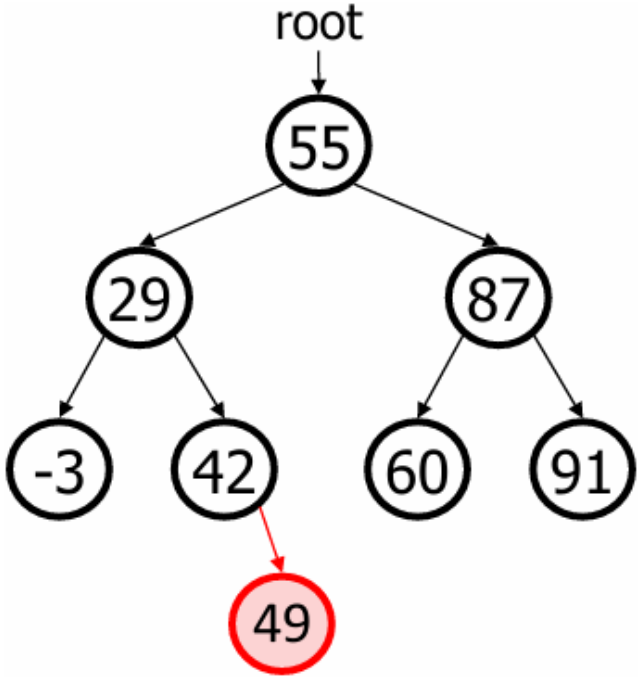
50, 20, 75, 98, 80, 31, 150, 39, 23, 11, 77



In-class Exercise: add()

- Write a function add() to add a given integer to a BST. Add the new value in the proper place to **maintain BST ordering**.

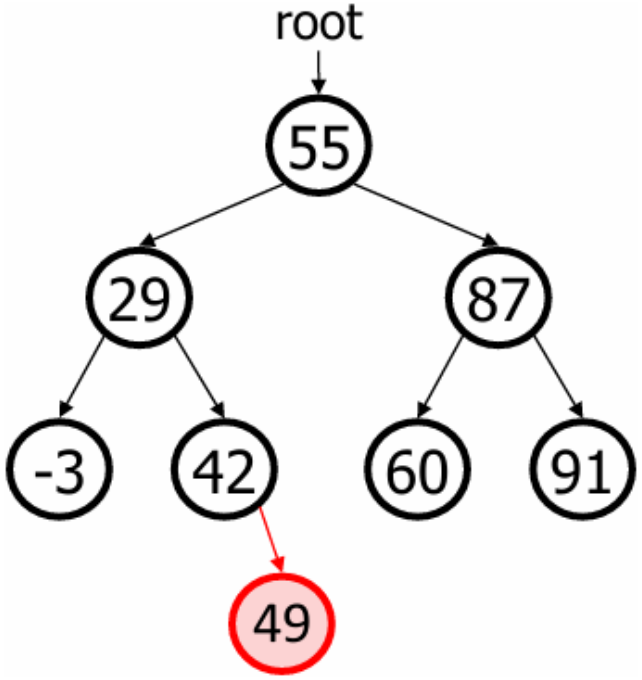
```
tree.add(root, 49);
```



In-class Exercise: add()

- Write a function add() to add a given integer to a BST. Add the new value in the proper place to **maintain BST ordering**.

```
tree.add(root, 49);
```

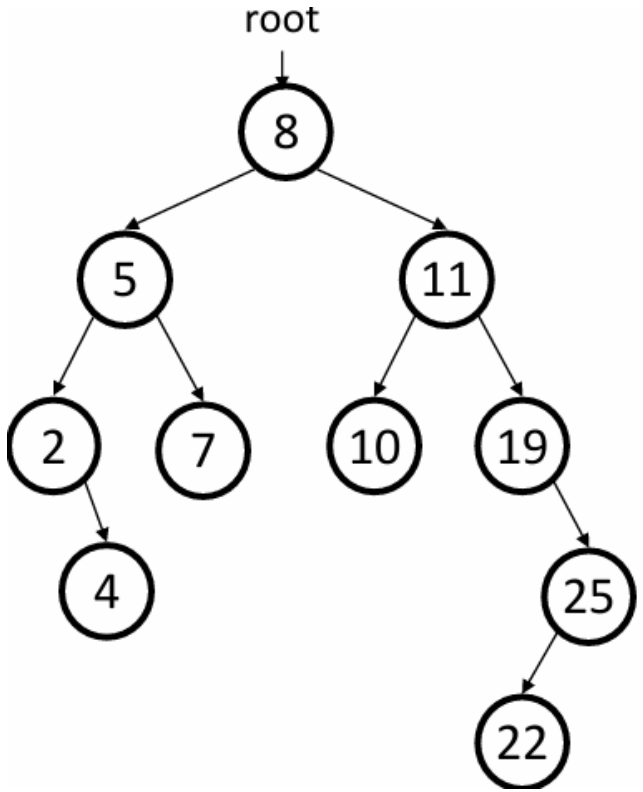


(pass the current node by reference for changes to be seen)

```
void add(TreeNode*& node, int value) {
    if (node == nullptr) {
      node = new TreeNode(value);
    } else if (node->data > value) {
      add(node->left, value);
    } else if (node->data < value) {
      add(node->right, value);
    }
}
```

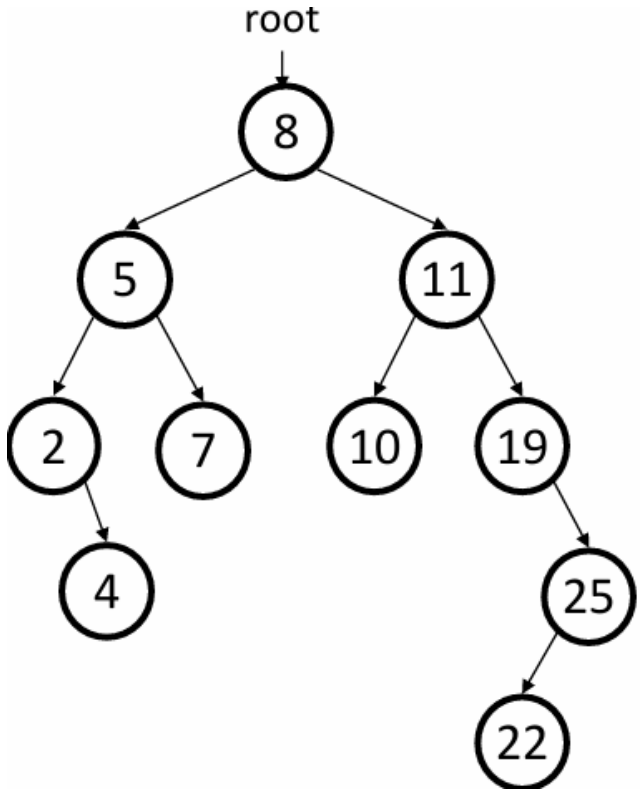
Free the Memory of a Tree

- To avoid leaking memory when discarding a tree, we must free the memory for every node.
- Like most tree problems, often written recursively
- must free the node itself, and its left/right subtrees
- this is another traversal of the tree
  - **should it be pre-, in-, or post-order?**



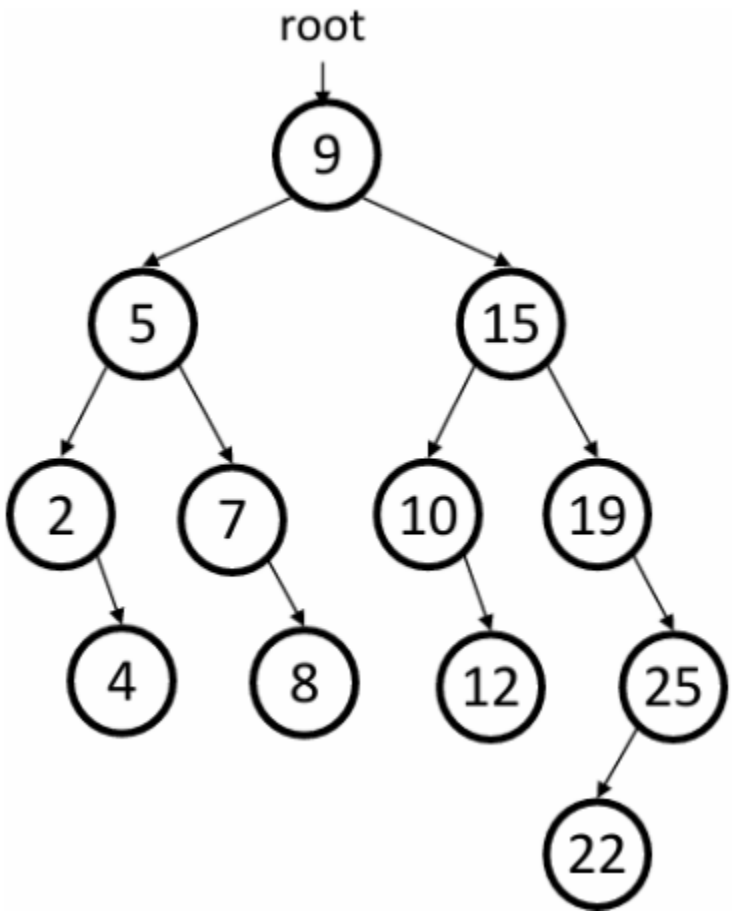
Free the Memory of a Tree

```
void freeTree(TreeNode*& node) {
   if (node == nullptr) {
     return;
   }
   freeTree(node->left);
   freeTree(node->right);
   delete node;
}
```



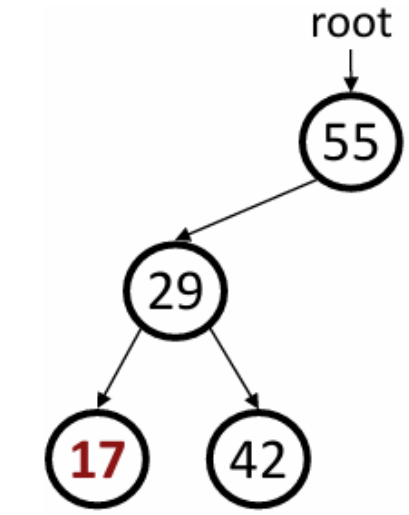
Removing from a BST

- If we want to remove values from the BST below.
- Removing a leaf like 4 or 22 is easy.
- What about removing 2? or 19?
- How can you remove a node with two large subtrees under it, such as 15 or 9?

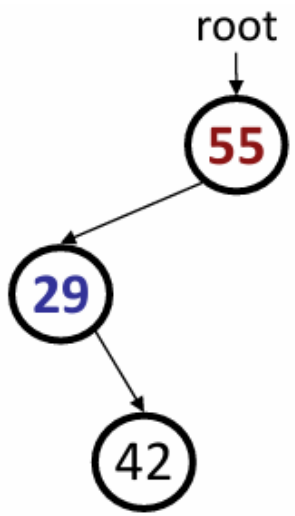


Cases for Removal

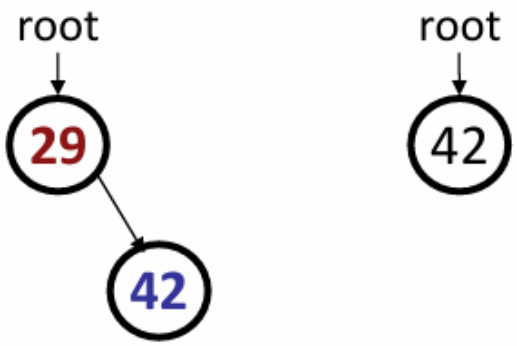
1. a **leaf**: **Replace with nullptr**
2. a node **with a left child only**: **Replace with left child**
3. a node **with a right child only**: **Replace with right child**



remove(root, 17);



remove(root, 55);

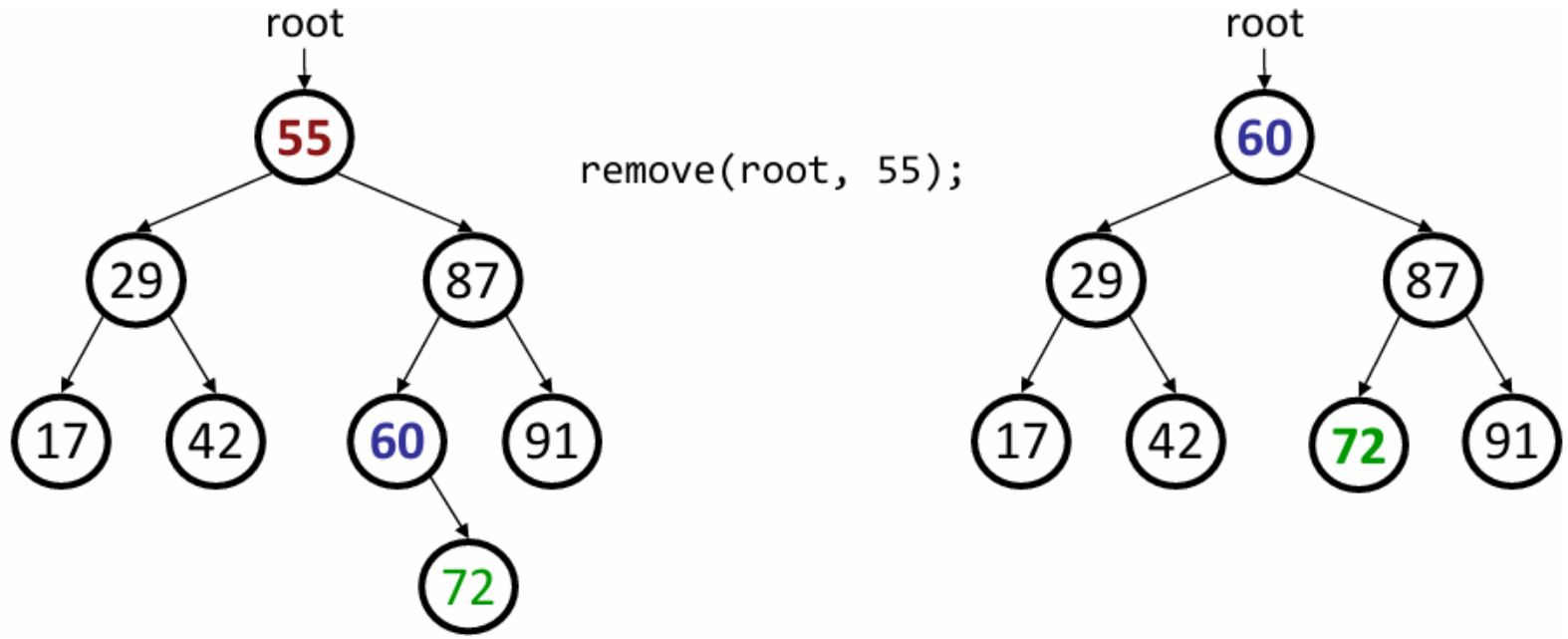


remove(root, 29);

Cases for Removal

4. a node **with both children**: **Replace with min from right**

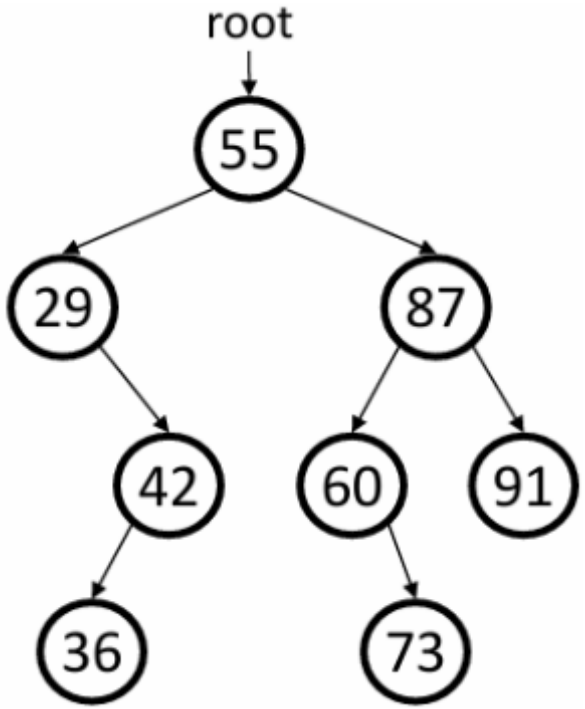
(replacing with max from left would also work)



In-class Exercise: remove()

- Add a function **remove()** that accepts a root pointer and removes a given integer value from the tree, if present. Remove the value in such a way as to maintain BST ordering.

```
remove(root, 73);
remove(root, 29);
remove(root, 87);
remove(root, 55);
```



remove() - solution

```
// Removes a given value from a BST
void remove(TreeNode*& node, int value) {
 if (node == nullptr) {
   return;
 } else if (value < node->data) {
   remove(node->left, value); // too small; go left
 } else if (value > node->data) {
   remove(node->right, value); // too big; go right
 } else { // value == node->data; remove this node!
   deleteNode(node);
 }
}
```

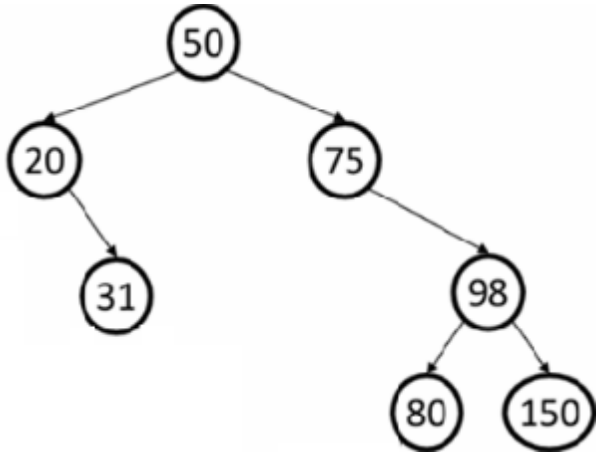
(continued on next slide)

remove() - solution

```
deleteNode(TreeNode*& node){
if (node->right == nullptr) { // case 1 or 2: no R child; replace with left
    TreeNode* trash = node;
    node = node->left;
    delete trash;
 } else if (node->left == nullptr) { // case 3: no L child; replace with right
    TreeNode* trash = node;
    node = node->right;
    delete trash;
 } else { // case 4: L+R both child; replace with min from right
    int min = getMin(node->right);
    remove(node->right, min);
    node->data = min;
 }
}
```

Overflow

- We saw how to add to a binary search tree. **Does it matter what order we add in?**
- Try adding: 50, 20, 75, 98, 80, 31, 150



- Now add the same numbers but in sorted order: 20, 31, 50, 75, 80, 98, 150

Exercise 8.1

- Complete LeetCode 700

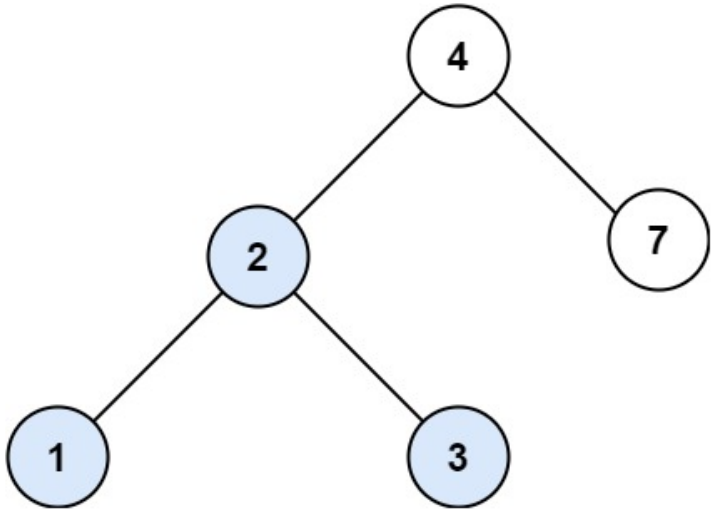
700. Search in a Binary Search Tree

Easy | Topics | Companies

You are given the `root` of a binary search tree (BST) and an integer `val`.

Find the node in the BST that the node's value equals `val` and return the subtree rooted with that node. If such a node does not exist, return `null`.

Example 1:




```
Input: root = [4,2,7,1,3], val = 2
Output: [2,1,3]
```

Exercise 8.2

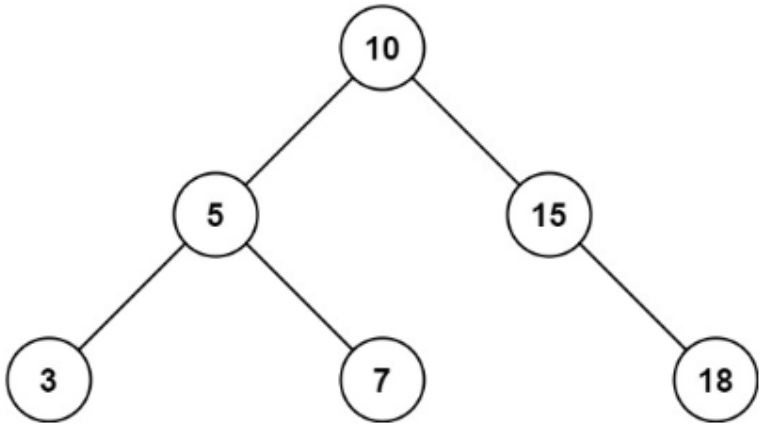
- Complete LeetCode 938

938. Range Sum of BST

Easy | Topics | Companies

Given the `root` node of a binary search tree and two integers `low` and `high`, return *the sum of values of all nodes with a value in the* ***inclusive*** *range* `[low, high]`.

Example 1:




```
Input: root = [10,5,15,3,7,null,18], low = 7, high = 15
Output: 32
Explanation: Nodes 7, 10, and 15 are in the range [7, 15]. 7 + 10 + 15 = 32.
```

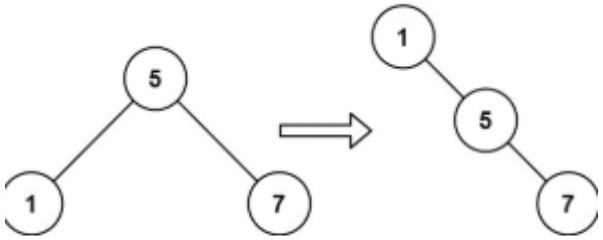
Exercise 8.3

- Complete LeetCode 897

897. Increasing Order Search Tree

Easy | Topics | Companies

Given the `root` of a binary search tree, rearrange the tree in **in-order** so that the leftmost node in the tree is now the root of the tree, and every node has no left child and only one right child.



Input: root = [5,1,7]
Output: [1,null,5,null,7]

Exercise 8.4

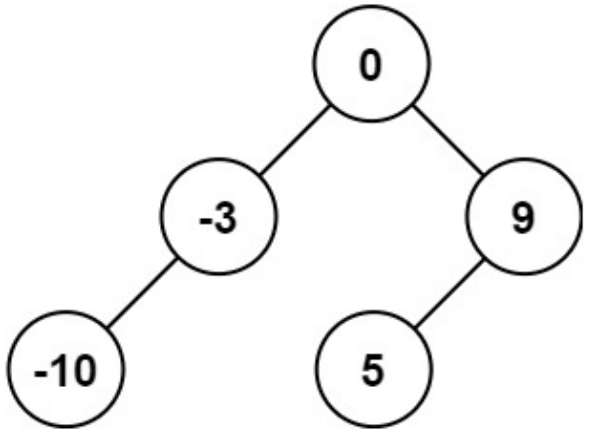
- Complete LeetCode 108

108. Convert Sorted Array to Binary Search Tree

Easy | Topics | Companies

Given an integer array `nums` where the elements are sorted in **ascending order**, convert *it to a* ***height-balanced*** *binary search tree.*

Example 1:




```
Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: [0,-10,5,null,-3,null,9] is also accepted:
```

A **height-balanced** binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

Exercise 8.5

- Complete LeetCode 701

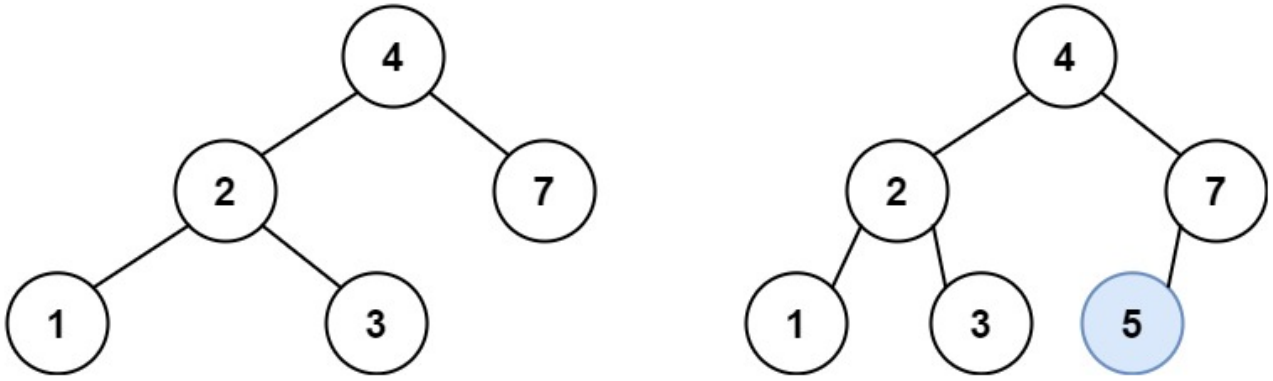
701. Insert into a Binary Search Tree

Medium | Topics | Companies

You are given the `root` node of a binary search tree (BST) and a `value` to insert into the tree. Return *the root node of the BST after the insertion*. It is **guaranteed** that the new value does not exist in the original BST.

Notice that there may exist multiple valid ways for the insertion, as long as the tree remains a BST after insertion. You can return **any of them**.

Example 1:




```
Input: root = [4,2,7,1,3], val = 5
Output: [4,2,7,1,3,5]
```

Exercise 8.6

- Complete LeetCode 1008

1008. Construct Binary Search Tree from Preorder Traversal

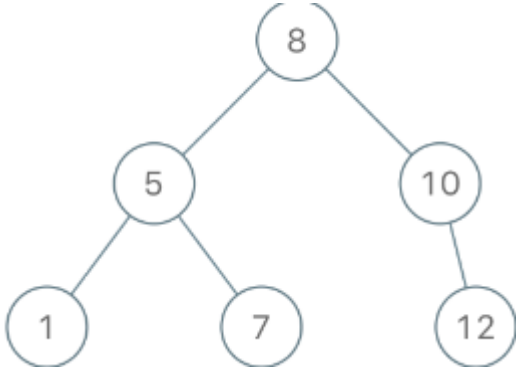
Medium | Topics | Companies

Given an array of integers preorder, which represents the **preorder traversal** of a BST (i.e., **binary search tree**), construct the tree and return *its root*.

It is **guaranteed** that there is always possible to find a binary search tree with the given requirements for the given test cases.

A **binary search tree** is a binary tree where for every node, any descendant of `Node.left` has a value **strictly less than** `Node.val`, and any descendant of `Node.right` has a value **strictly greater than** `Node.val`.

A **preorder traversal** of a binary tree displays the value of the node first, then traverses `Node.left`, then traverses `Node.right`.



Input: `preorder = [8,5,1,7,10,12]`
Output: `[8,5,10,1,7,null,12]`