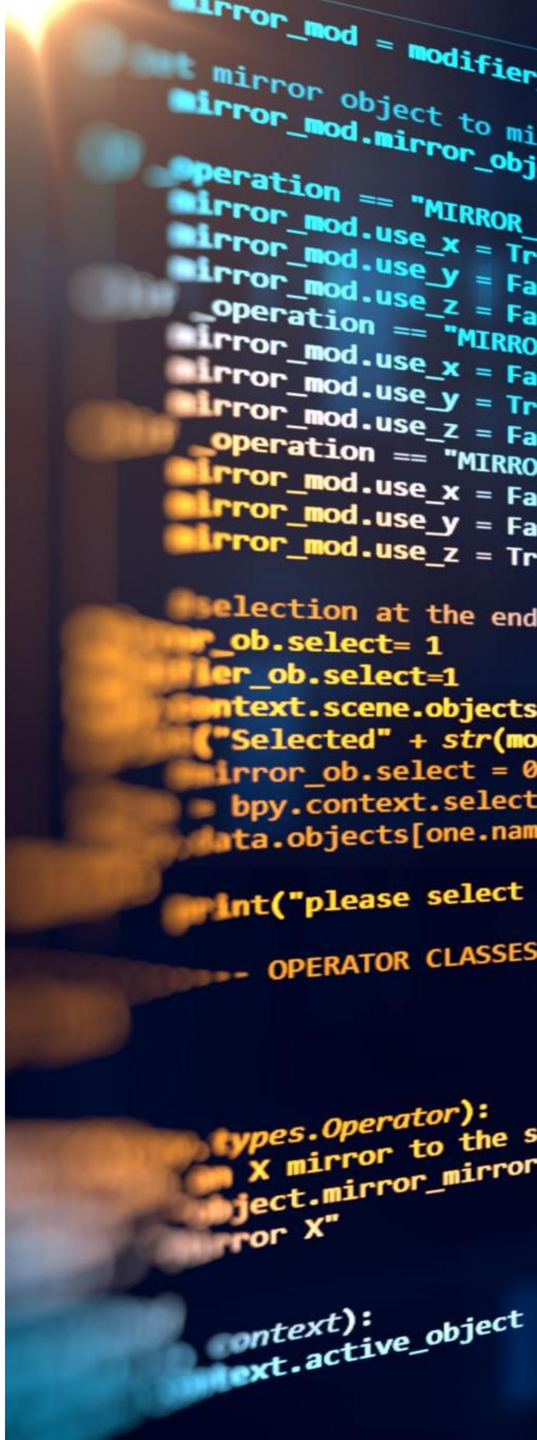
Please use the following QR code to check in and record your attendance.

CS 1037
Fundamentals of Computer Science II

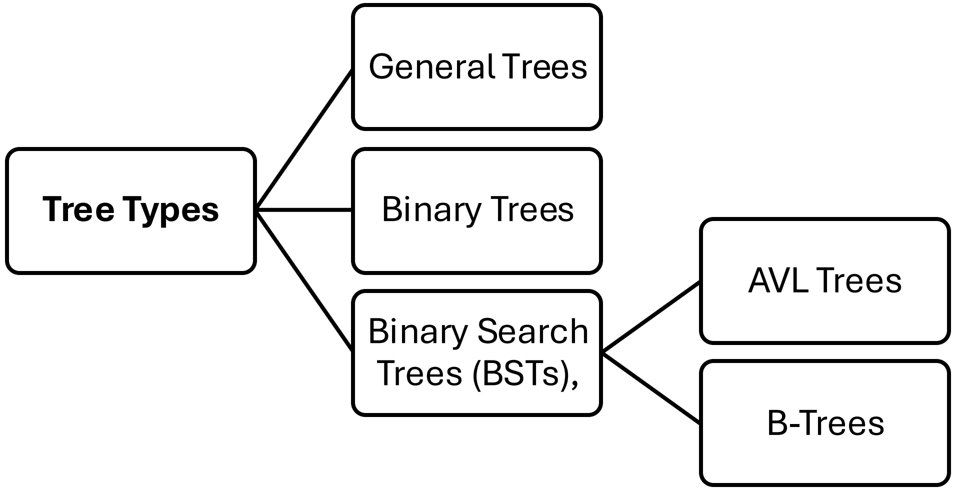
Tree ADT (cont.)

Ahmed Ibrahim



Trees & Binary Trees Recap

- **Tree Structure**: Nodes connected in a parent-child hierarchy.
- **Terminology**: Root, child, parent, depth, height, siblings, leaves.



Types of Binary Trees

Full	Complete	Perfect
Every node has either 0 or 2 children.	All levels filled except possibly the last, filled from the left.	All internal nodes have 2 children, and all leaves are at the same level.

Tree Traversals Recap

Pre-order – Root → Left → Right (for computations before children).

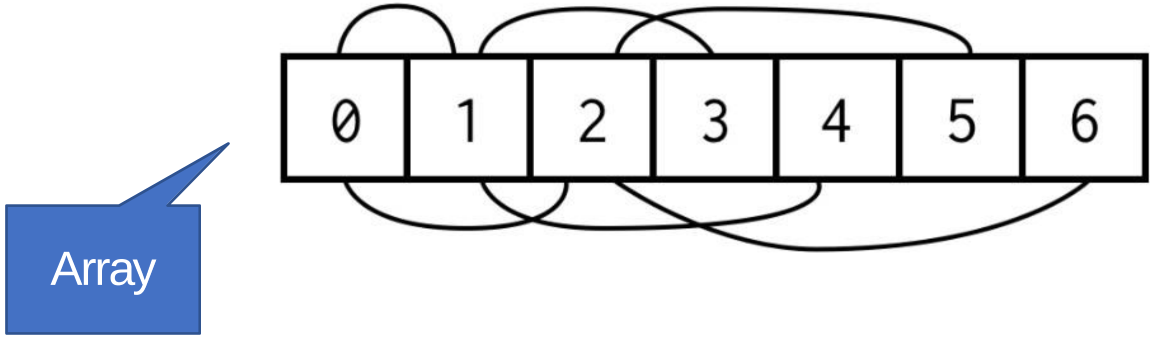
In-order – Left → Root → Right (used to retrieve BST in sorted order).

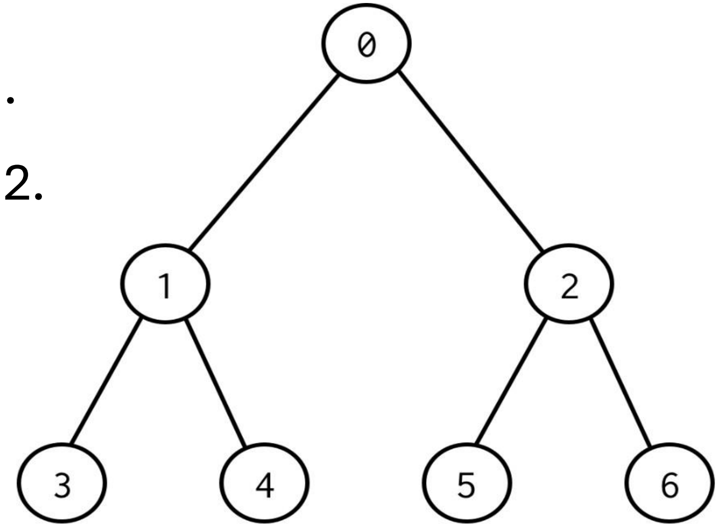
Post-order – Left → Right → Root (for deleting nodes).

Level-order – Each level from root to leaves (Breadth-First).

Array-Based Implementation of BT

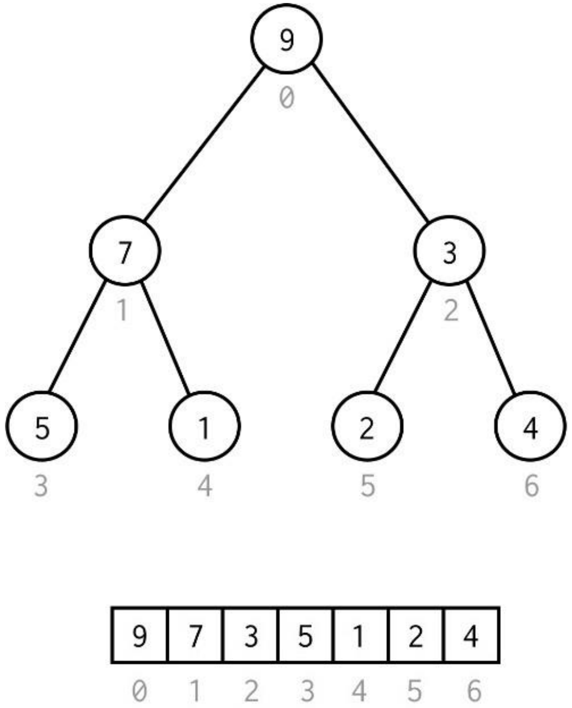
- Nodes are stored in an array
- For each node q, define
  - If q is the root, index(q) = 0
  - If a node is at index p, its left child is stored at index 2 * p + 1.
  - If a node is at index p, its right child is stored at index 2 * p + 2.

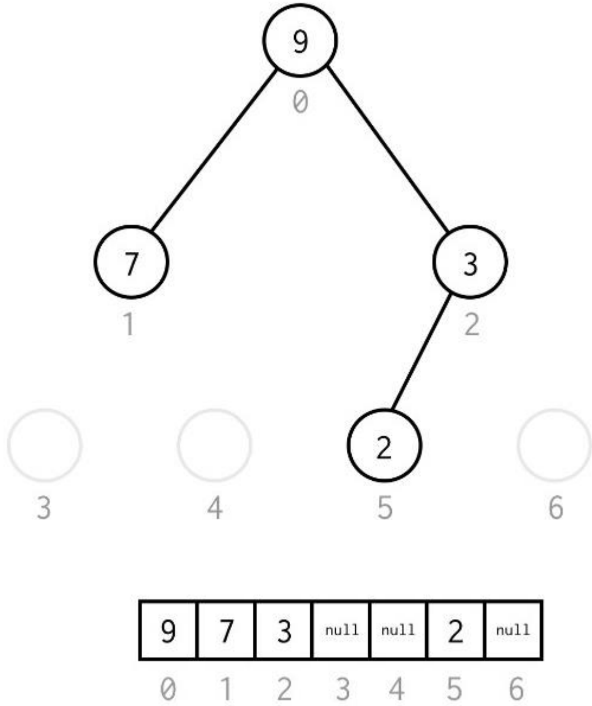




a[i] has children a[2*i+1] and a[2*i+2]

Examples

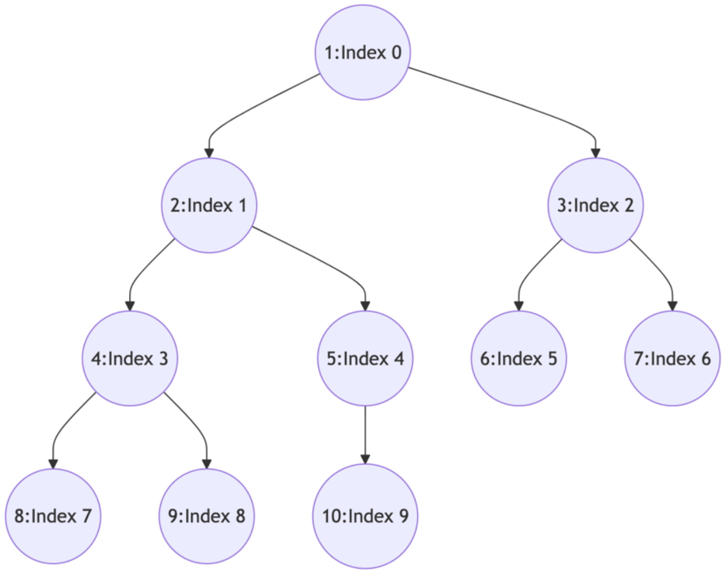

9
0
7
1
3
2
5
3
1
4
2
5
4
6
9 7 3 5 1 2 4
0 1 2 3 4 5 6


9
0
7
1
3
2
3
4
2
5
6
9 7 3 null null 2 null
0 1 2 3 4 5 6

Question!

Given the following C code:

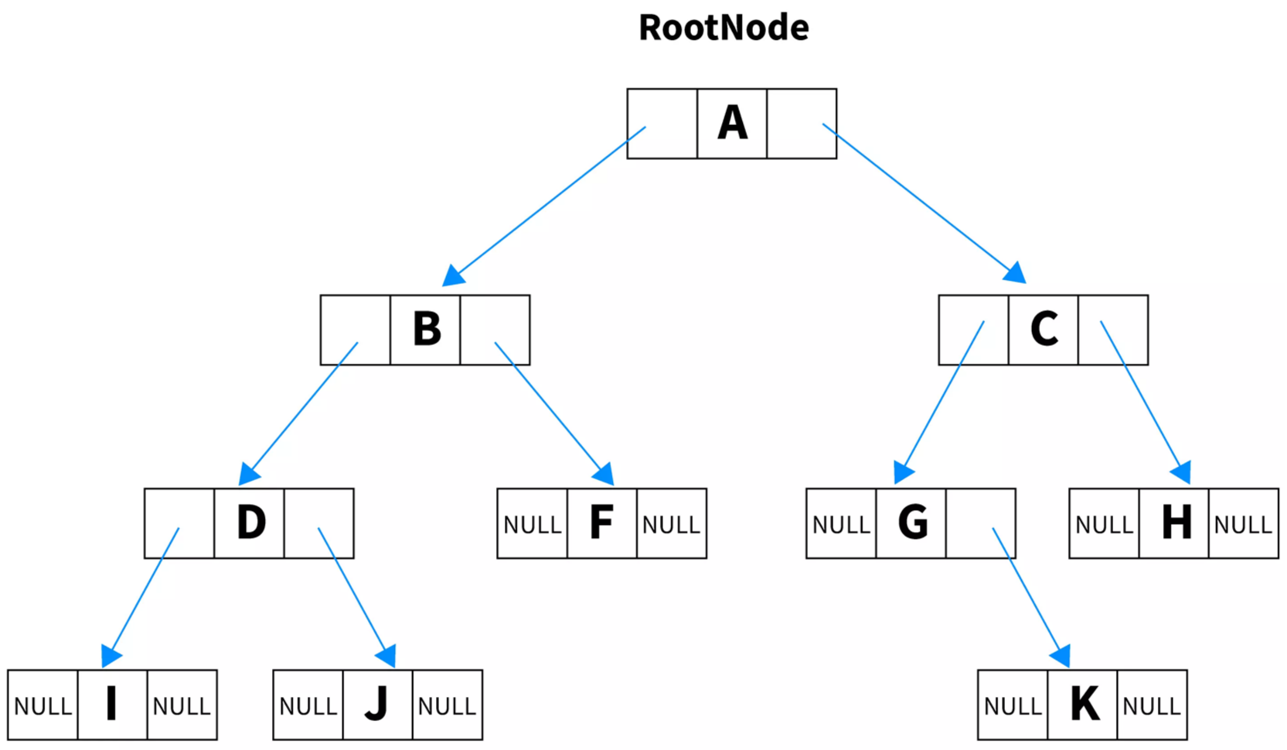
```
#include <stdio.h>

void find_child_values(int array[], int index) {
int left_index = 2 * index + 1;
int right_index = 2 * index + 2;

int left_child;
if (left_index < 10) {left_child = array[left_index];} else {
left_child = -1;}

int right_child;
if (right_index < 10) {right_child = array[right_index];} else {
right_child = -1;

printf("Left child: %d, Right child: %d\n", left_child, right_child);}

int main() {
int array_tree[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
find_child_values(array_tree, 1);
find_child_values(array_tree, 3);
return 0;}
```



What will be the output of this code?

Linked List - Based BT Implementation

- A node is represented by an object storing
  - **Element**: The data or value of the node.
  - **Left Child Node**: A reference to the left child, if it exists.
  - **Right Child Node**: A reference to the right child, if it exists.



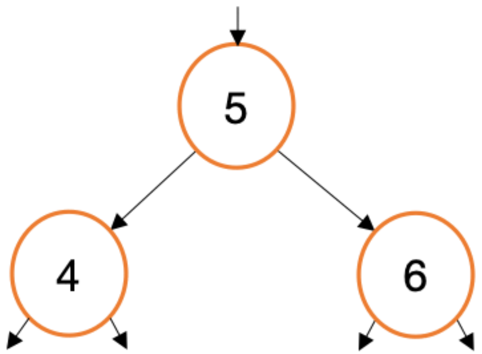
Source: https://www.scaler.com/topics/binary-tree-in-c

Linked List-Based BT Implementation

```
// Define the structure for a node using typedef
typedef struct Node {
char element; // The value of the node
struct Node* left; // Pointer to the left child
struct Node* right;// Pointer to the right child
} Node;


// Function to create a new node
Node* createNode(char value) {
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->element = value;
newNode->left = NULL; // Initialize left child as NULL
newNode->right = NULL; // Initialize right child as NULL
return newNode;
}
```

Insertion in a Binary Tree

- In a binary tree, nodes are added **level** by **level** from **left** to **right**, typically at <u>the first available position</u>.



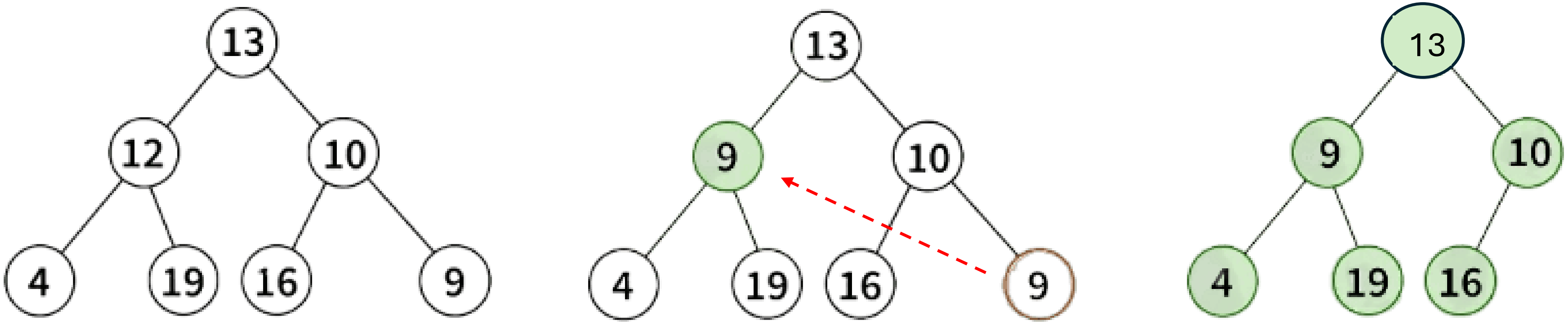

```
Algorithm (Level-order):
1. Start from the root node.
2. Use a queue to traverse the tree level by level.
For each node:
   4. Check if it has a left child; if not, insert the new node here and stop.
   5. If there is a left child, move to the right child.
   6. If there is no right child, insert the new node here.
   If both left and right children exist, add them to the queue to continue the traversal.
```

Add a new value, 3, to existing tree of three nodes

Source: https://www.gormanalysis.com/blog/making-a-binary-search-tree-in-cpp/

Deletion in a Binary Tree

- When deleting a node in a binary tree, the node is **replaced** by the deepest rightmost node to maintain the tree structure.

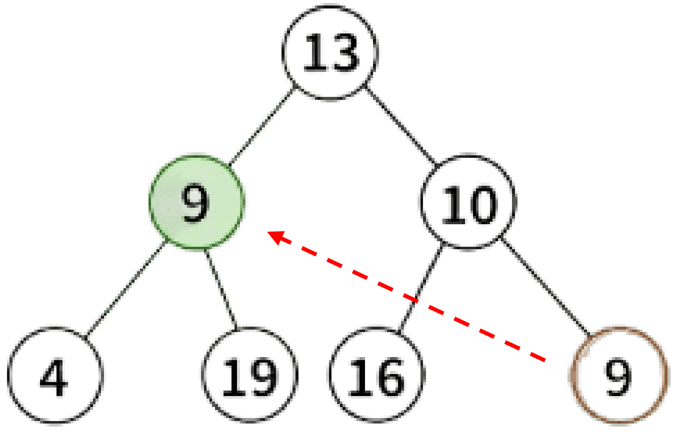


Node to be deleted in 12 | Replacing 12 with the deepest node | The tree after deletion

Deletion in a Binary Tree

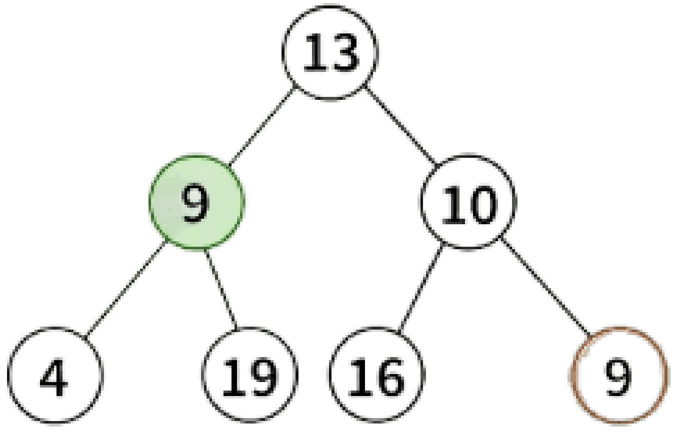
Algorithm:

1. Start from the root node.
2. Use a level-order traversal (queue-based) to locate the target and deepest rightmost nodes.
3. Replace the target node's value with the deepest rightmost node's value.
4. Delete the deepest rightmost node.



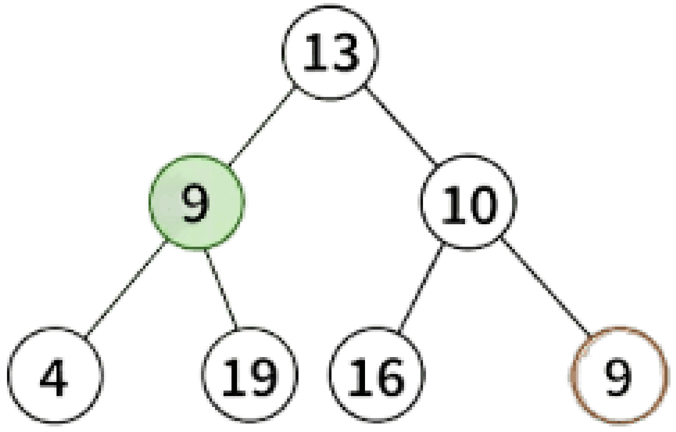
Find And Remove Deepest Rightmost

```
Node* findAndRemoveDeepestRightmost(Node* root) {
if (root == NULL) return NULL;
Queue* queue = createQueue();
enqueue(queue, root);
Node* current = NULL;
Node* parent = NULL;

   while (!isQueueEmpty(queue)) {
   parent = current; // Track the parent of the last node
   current = dequeue(queue); // Current node being processed
   if (current->left) {enqueue(queue, current->left);}
   if (current->right) {enqueue(queue, current->right);}
   }

// `current` now points to the deepest rightmost node, and `parent` is its parent.
// Remove the deepest rightmost node by setting the appropriate child of `parent` to NULL.
   if (parent) {
   if (parent->right == current) {parent->right = NULL;}
else {parent->left = NULL;}}
return current;
}
```



Delete a Node in a Binary Tree

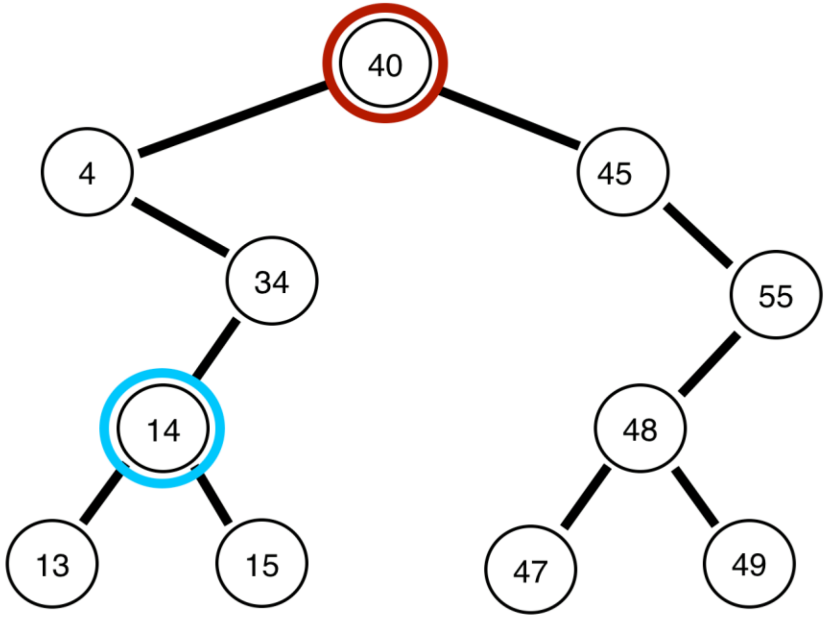
```
Node* deleteNode(Node* root, int key) {
// Check if the tree is empty or not
// Special case: tree is the only one node (root)
Queue* queue = createQueue();
enqueue(queue, root);
Node* nodeToDelete = NULL;

    while (!isQueueEmpty(queue)) {
    Node* current = dequeue(queue);
    if (current->data == key) nodeToDelete = current;
    if (current->left) enqueue(queue, current->left);
    if (current->right) enqueue(queue, current->right);}

if (nodeToDelete) {
        Node* deepestNode = findAndRemoveDeepestRightmost(root);
        nodeToDelete->data = deepestNode->data; // Replace data
      free(deepestNode); } // Free the deepest rightmost node
   return root;
}
```



Search in a Binary Tree

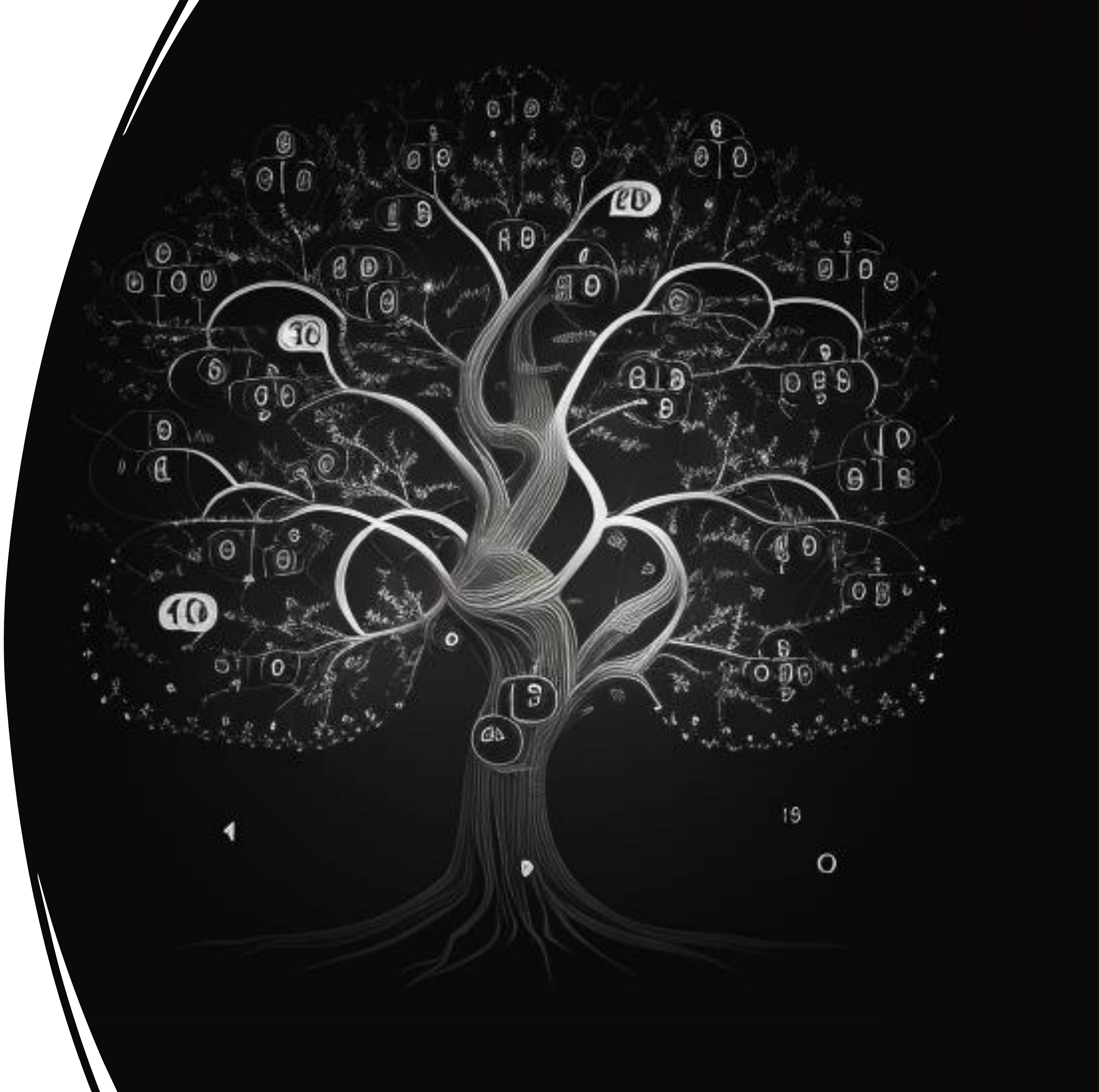
- Searching involves locating a node with a specific value using either **depth-first** or **breadth-first** search methods.

```
Algorithm (using DFS):
1. If the root node is NULL, return NULL.
2. If the root node's data matches the target
value, return the node.
3. Recursively search the left subtree, then
the right subtree.
Return the node if found; otherwise, return
NULL.
```



Source: https://medium.com/@tarlanahad/binary-trees-and-bsts-oxygen-source-of-data-structures-a6eb0f8e629b

Binary Search Tree



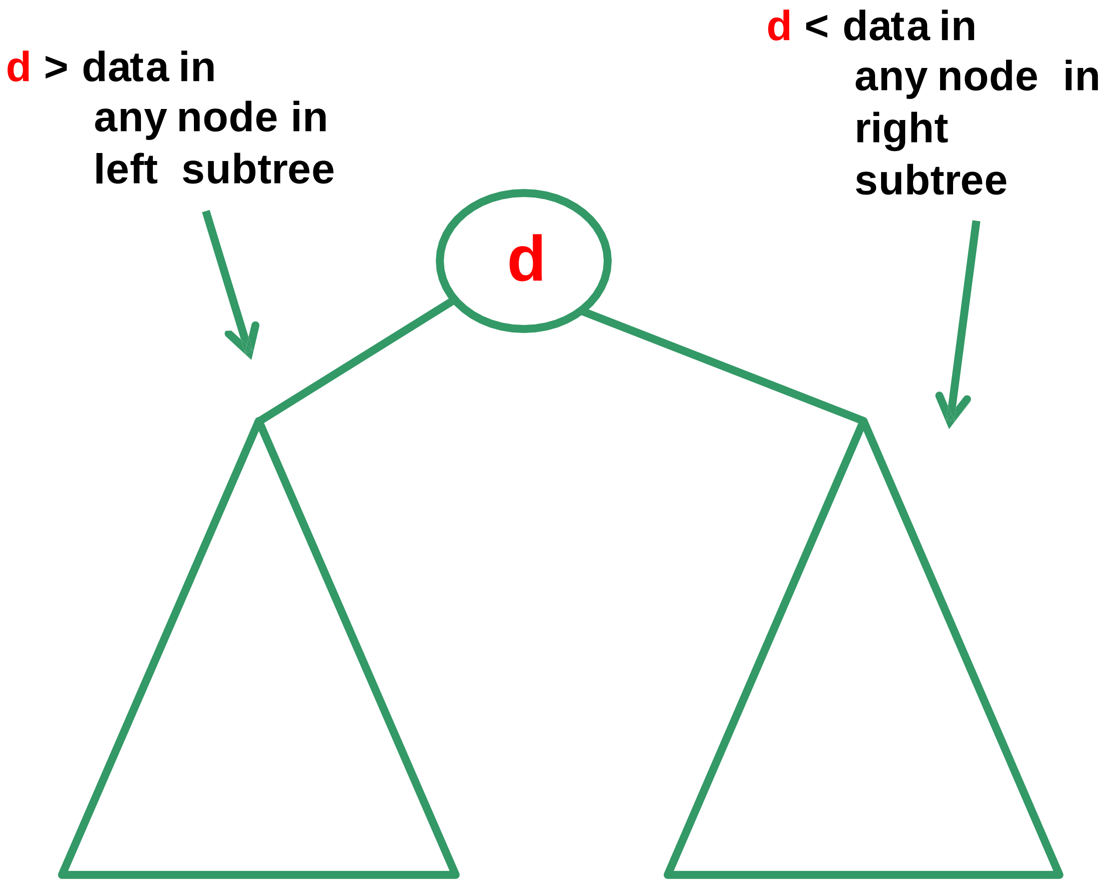
Binary Search Trees (BST)

- What is a Binary Search tree?

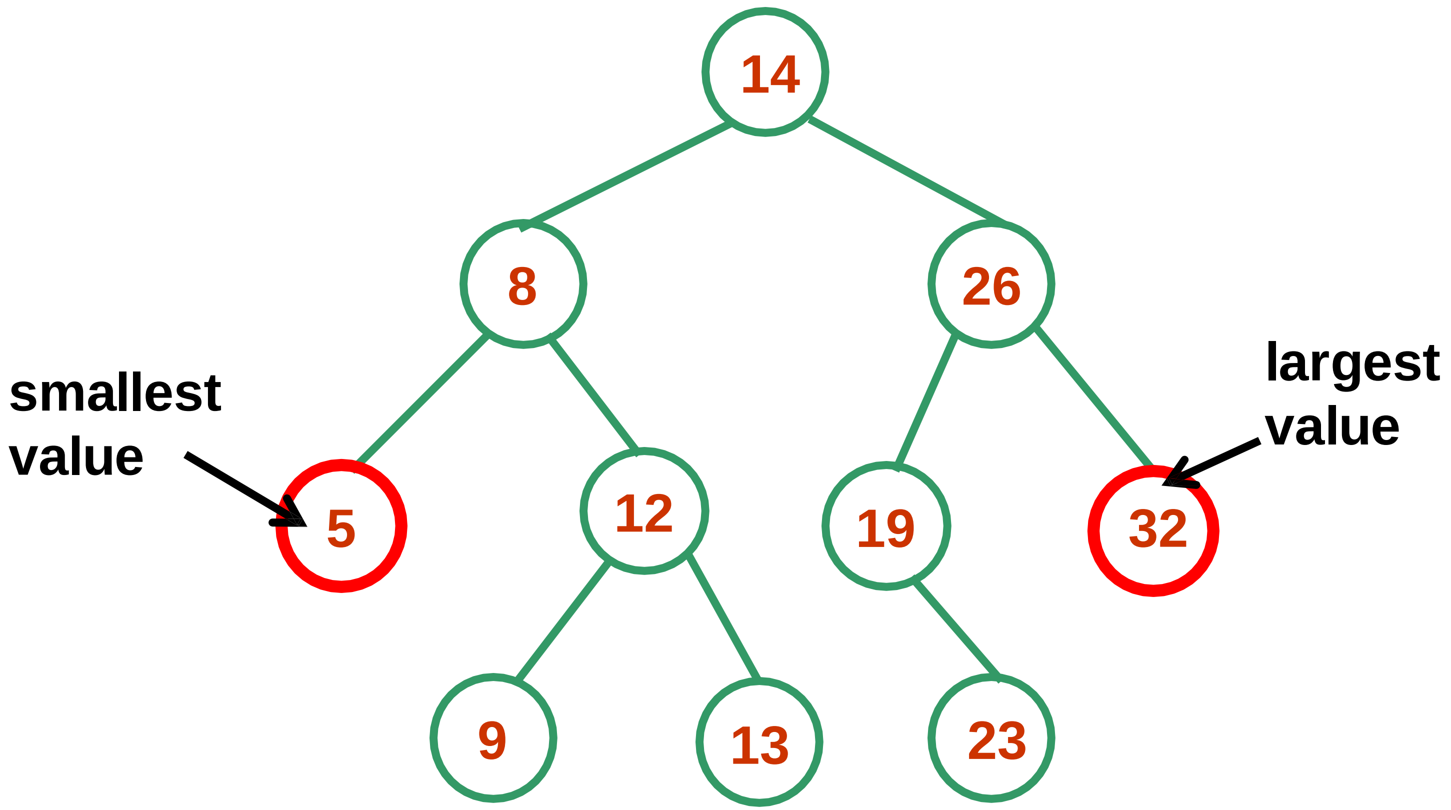
  A binary search tree is a **binary tree** in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

- Every BST is a BT, but every BT must not be a BST.

- There must be no duplicate nodes (in general).



Properties of Binary Search Trees



Searching for a Value in a BST

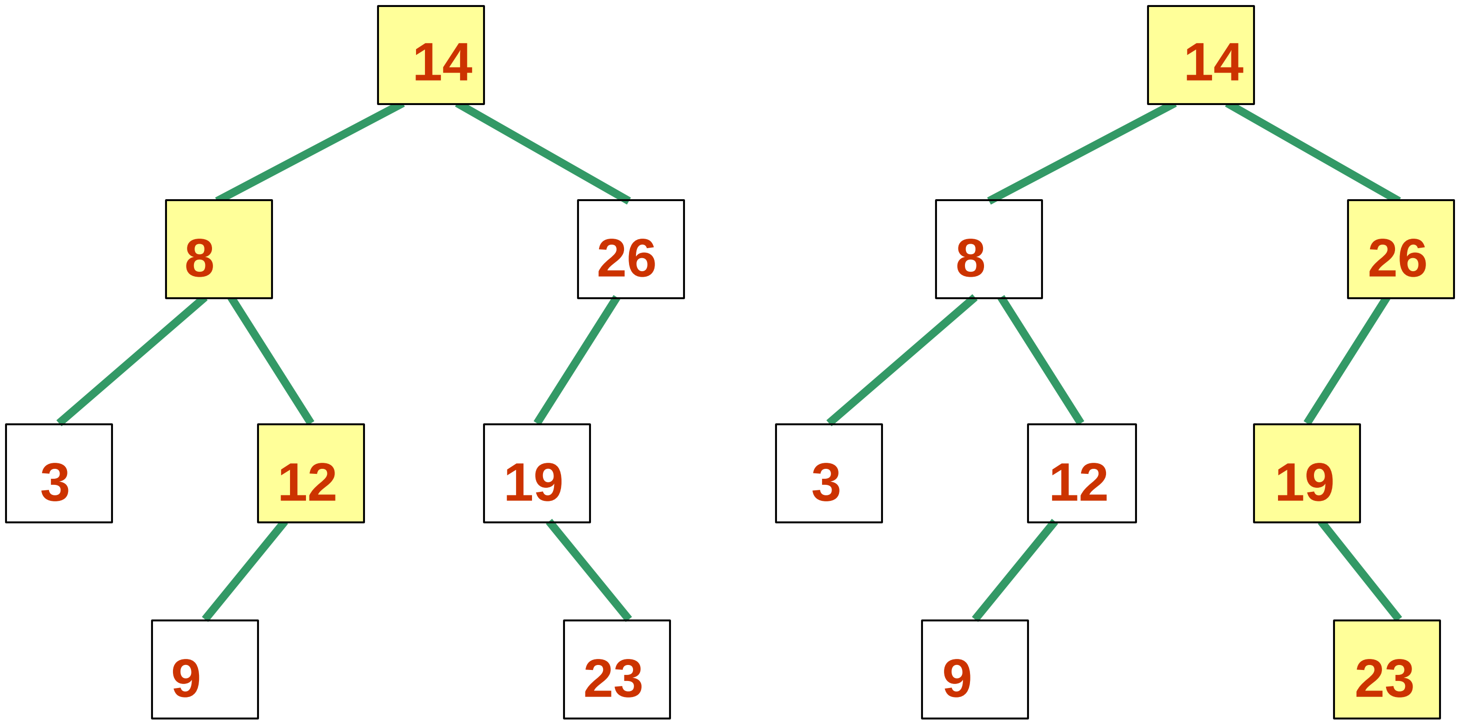
- A binary search tree is a special case of a binary tree.
  - So, it has all the **operations** of a **binary tree**.
- It also has *operations specific to a BST*:
  - add an element (requires that the **BST property be maintained**)
  - remove an element (requires that the **BST property be maintained**)
  - remove the maximum element
  - remove the minimum element

Algorithm of Searching for a Value in a BST

```
Node* searchBST(Node* node, int key) {
if (node == NULL) {return NULL;} // Not found
if (node->data == key) {return node;} // Found the node
if (key < node->data) {
  return searchBST(node->left, key); // Search in the left subtree
  } else {
  return searchBST(node->right, key); // Search in the right subtree
  }
}
```

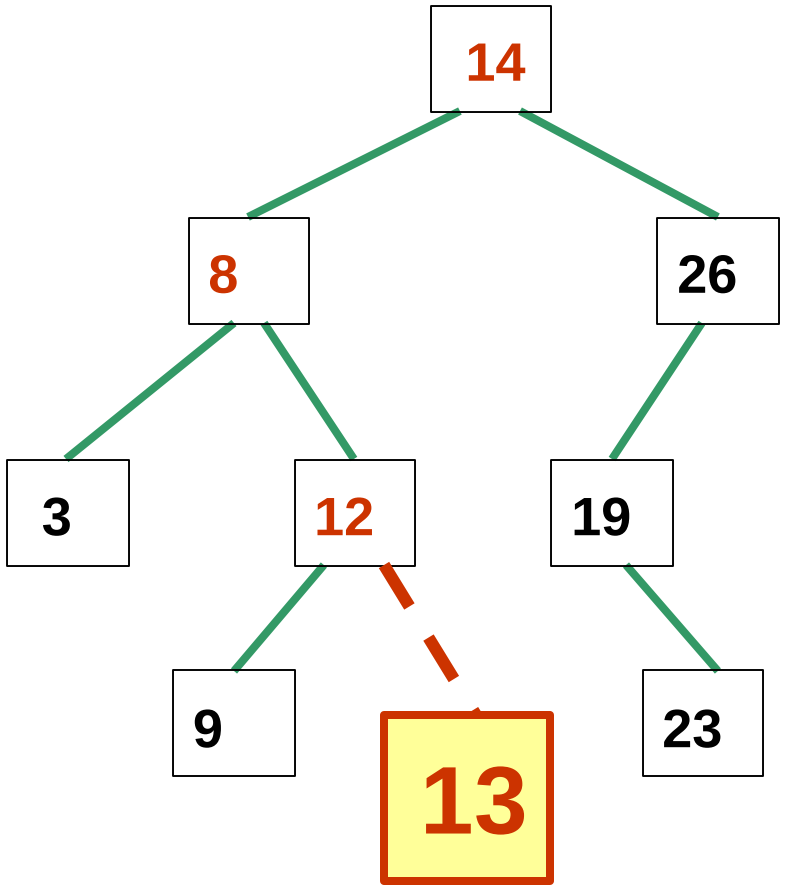
Searching for a Value in a BST

- Example #1: Search for 13: visited nodes are colored yellow; return false when a node containing 12 has no right child.
- Example #2: Search for 22: return false when a node containing 23 has no left child.



Insert a Value in a BST

- Same nodes are visited as when searching for 13.
- Instead of returning false when the node containing 12 has no right child, build the new node, attach it as the right child of the node containing 12, and return true.



Insert a Value in a BST (cont.)

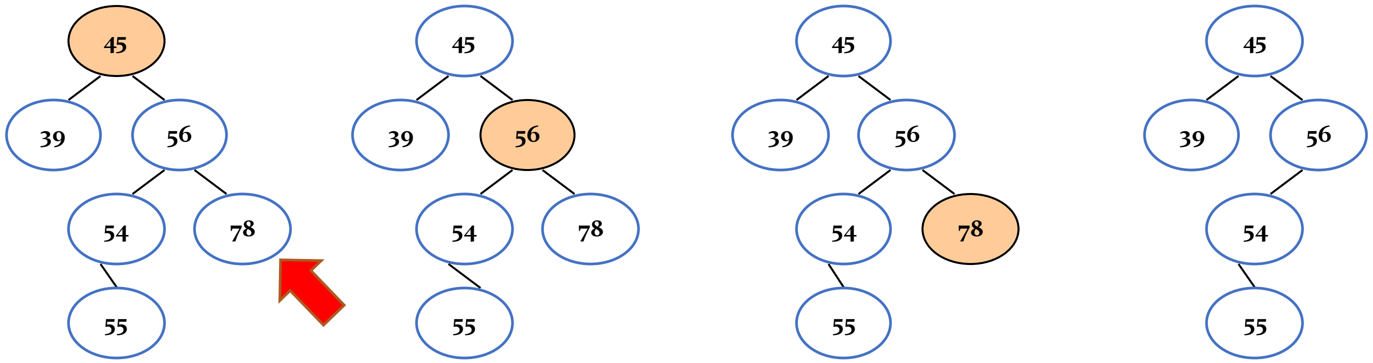
```
Node* insertBST(Node* node, int key) {
if (node == NULL) {return createNode(key);} // If the tree is empty, create a new node

// Otherwise, recur down the tree
if (key < node->data) {
   node->left = insertBST(node->left, key); // Insert in the left subtree
} else if (key > node->data) {
  node->right = insertBST(node->right, key); // Insert in the right subtree
}
 return node;
}
```

Deleting a Value from a BST

- Care should be taken that the BSTs' properties are <u>not violated</u> and nodes are not lost in the process.
- The deletion of a node involves any of the three following cases.

  Case 1: Deleting a node that has no children.

  For example, deleting node 78 in the tree below.



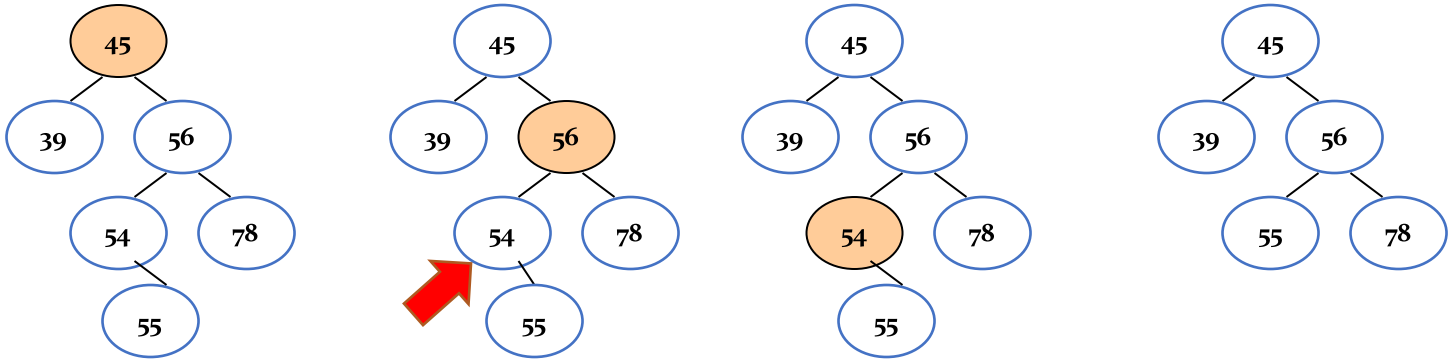
Delete a Leaf Node

```
void deleteLeafNode(Node** root, int key) {
if (*root == NULL) {return;} // Tree is empty or node not found

// Find the node to delete and check if it's a leaf node
if (key < (*root)->data) {
  deleteLeafNode(&(*root)->left, key); // Search in the left subtree
 } else if (key > (*root)->data) {
  deleteLeafNode(&(*root)->right, key); // Search in the right subtree
 } else {
// Node with the key is found, check if it's a leaf
  if ((*root)->left == NULL && (*root)->right == NULL) {free(*root); *root = NULL; }
 }
}
```

Deleting a Value from a BST

- ***Case 2:*** Deleting a node with one child (either left or right).
- To handle the deletion, the node's child is set to be the child of the node's parent.
  - Replace that node with its child node.
  - Remove the child node from its original position.

For example, deleting node 54 in the tree below.



Delete a Leaf Node

```
void deleteNodeWithOneChild(Node** root, int key) {
// Tree is empty or node not found

// Search in the left subtree
// Search in the right subtree

     // Node with key found, check if it has one child
     if ((*root)->left == NULL && (*root)->right != NULL) {
     // Node has only right child
     Node* temp = (*root)->right;
     free(*root);
     *root = temp; // Update parent’s pointer to bypass the deleted node
     } else if ((*root)->right == NULL && (*root)->left != NULL) {
     // Node has only left child
     Node* temp = (*root)->left;
     free(*root);
     *root = temp; // Update parent’s pointer to bypass the deleted node
       }
// If node has two children or no children, we’re not handling that here
  }
}
```

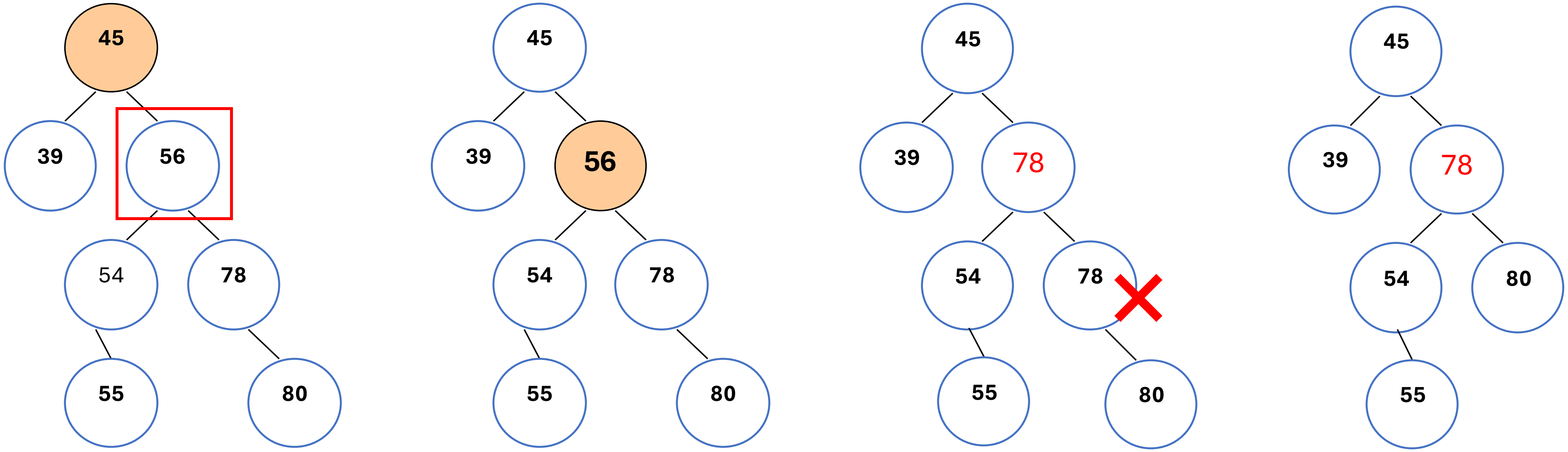
Deleting a Value from a BST

- ***Case 3:*** Deleting a node with two children.
  - Get the in-order successor of that node. The in-order successor of a node is the next node in the **in-order traversal** of the tree.
    - Simply, it is the node with the smallest value greater than the given node.
  - Replace the node with the in-order successor.
  - Remove the in-order successor from its original position.

  In the following slide an example of deleting node 56 in the tree.

Deleting a Value from a BST

- In in-order traversal, we go left -> root -> right, so the smallest value greater than the current node will be the leftmost node in the right subtree.




Thank you