Algorithms and Data Structures (CSci 115)

California State University Fresno

College of Science and Mathematics

Department of Computer Science

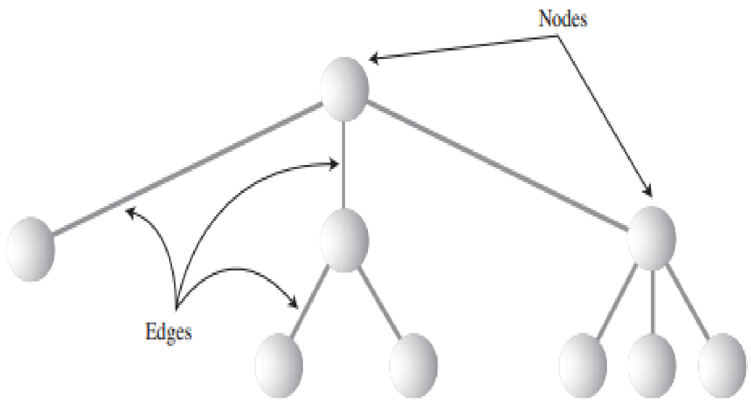
H. Cecotti

Learning outcomes

- Binary trees
  - Tree terminology
  - Binary tree structure
  - Binary Search Tree
  - Binary tree in C++

Tree Terminology

- A **tree** (T) consists of
  - a collection of **nodes** connected by a number of **edges**
- There is one specially designated node called the **root** of the tree (denoted as **R**)
- There can be **zero** or more **subtrees** connected to the root node
- The root node of each subtree is a **child** of the root
- There is an **edge** from a node to each of its children, and a node is said to be the **parent** of its children



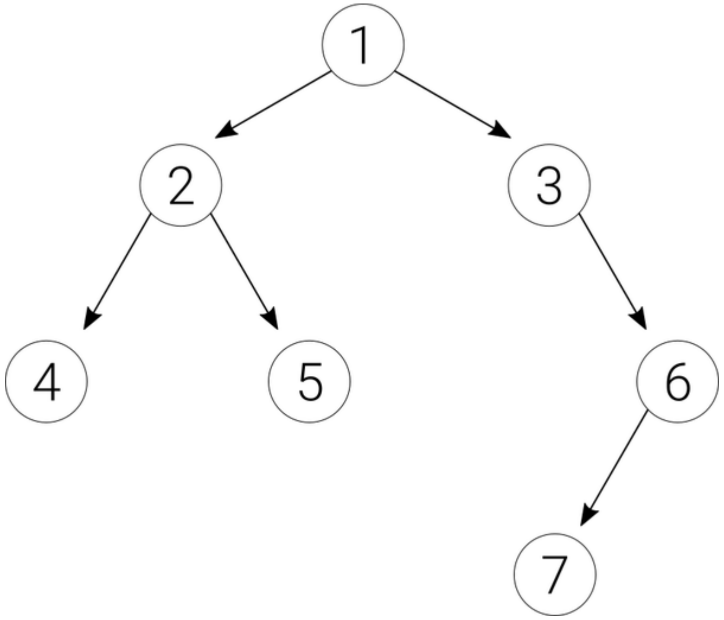

```
struct TreeNode {
        int item;           // The data in this node.
        TreeNode *left;     // Pointer to the left subtree.
        TreeNode *right;    // Pointer to the right subtree.
    }
```

Tree Terminology

- **Traversing**
  - To traverse a tree means to "visit all the nodes in a specified order".
  - Example: you might visit all the nodes in order of ascending key value
- **Depth:** The depth of a node: the number of **edges from** the **root** to the **node**.
- **Height:** The height of a node: the number of **edges from** the **node** to the deepest leaf.
  - height of a tree == a height of the root.
- **Levels:**
  - The **level** of a particular node refers to how many generations the node is **from** the root
    - The Root node is at Level 0 (start at 0)
    - The Root node's children are at Level 1, the Root node's grandchildren are at Level 2 etc.
- **Keys:**
  - One data field in an object is usually designated a **key value**
  - This key value is used to search for the item
  - In tree diagrams the key value of the item is typically shown in the circle
- **Size:** the total number of nodes in that tree

Tree Terminology

- **Example**
  - **height** of node **2** : 1
    - from **2** there is a path to 2 leaf nodes (**4** and **5)**
    - each of the 2 paths is only 1 edge long
    - → the largest is 1.
  - **height** of node **3** :2
    - from **3** there is a path to only 1 leaf node (**7**), and it has of 2 edges.
  - **height** of the tree: 3
    - from the root node **1** there is a path to 3 leaf nodes (**4**, **5**, and **7**)
    - the path to the **4** and **5** consists of 2 edges while the path to the **7** consists of 3 edges
    - → get the largest: 3.
  - **size** of the tree: 7.
  - **depth** of node **2** is 1
  - **depth** of node **3** is 1; the depth node **6** is 2.
  - **depth** of the binary tree == height of the tree == 3.



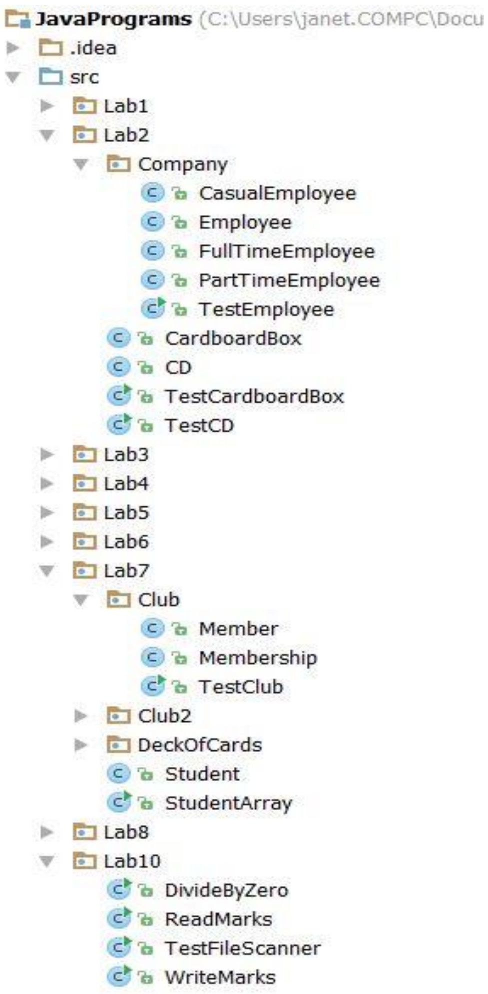
Tree Terminology

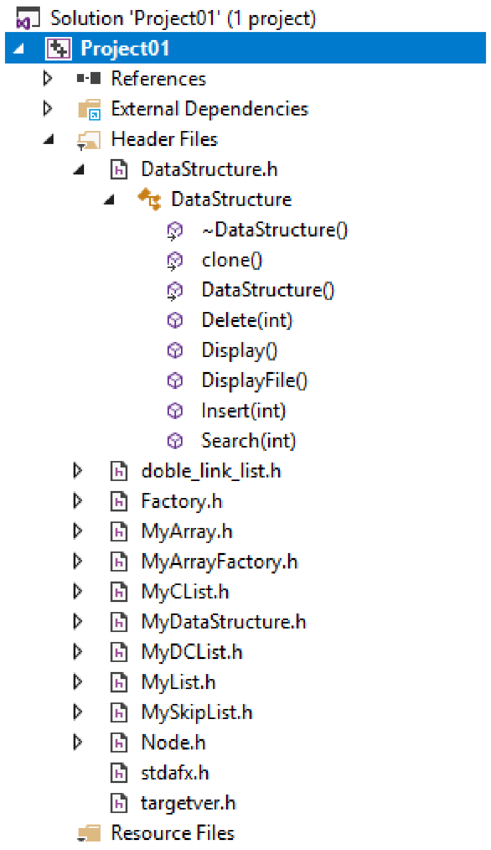
- **Definitions**
  - **Warning**
    - Be careful for Midterm 2 and the final !!!
      - Number of edges vs. Numbed of nodes
      - What is what?

Tree Topology

- Typically
  - There is **one node** in the **top row of a tree**
    - with lines connecting to more nodes on the second row, even more on the third row, and so on…

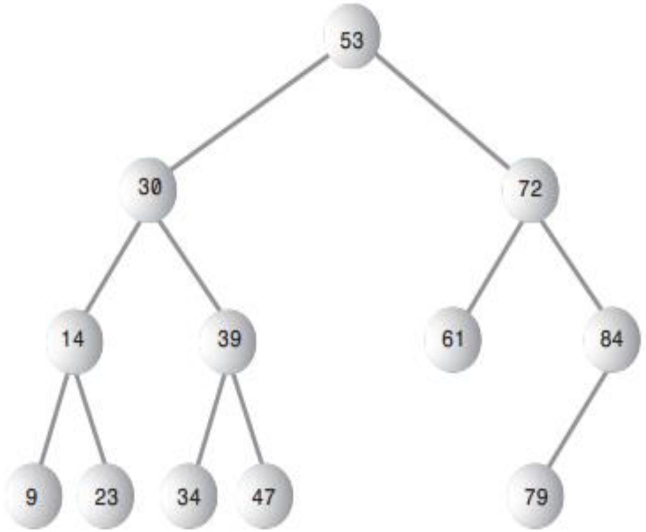
- Why might you want to use a tree?
  - Usually, because it combines the advantages of two other structures:
    - An ordered array, and
    - A linked list





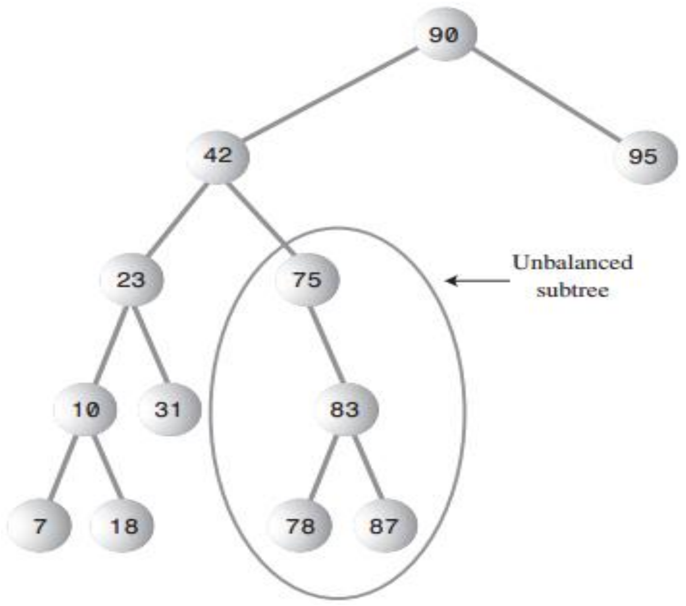
Binary Tree

- A **Binary Tree** is a tree in which any given node can have a maximum of 2 children
- The 2 children of each node in a binary tree: **left child** and the **right child**
- A node in a binary tree does **not** have to have exactly 2 children. It may have only
    - A left child (e.g. node 84 only has a left child),
    - A right child (no examples in diagram), or
    - No children at all (leaf nodes) (e.g. 9, 23, 61 …)
- Defining characteristics of a **Binary Search Tree** :
    - A node's left child has a key **less than** its parent, and
    - A node's right child has a key **greater than or equal** to its parent



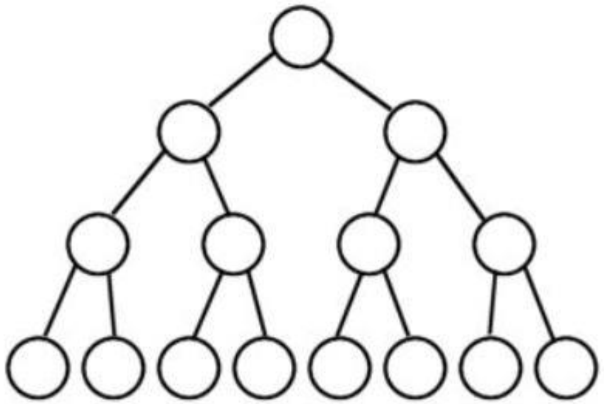
Unbalanced Trees

- An **unbalanced** tree has most of its nodes to 1 side of the root node
  - ➢ Either to the left or to the right of the root
  - ➢ Individual subtrees may also be unbalanced
- Trees become unbalanced because of the order in which the data items are inserted
  - ➢ If the key values are inserted randomly, the tree is likely to be more or less balanced
- If an ascending sequence or a descending sequence is generated the tree will be unbalanced.
  - ➢ Why?

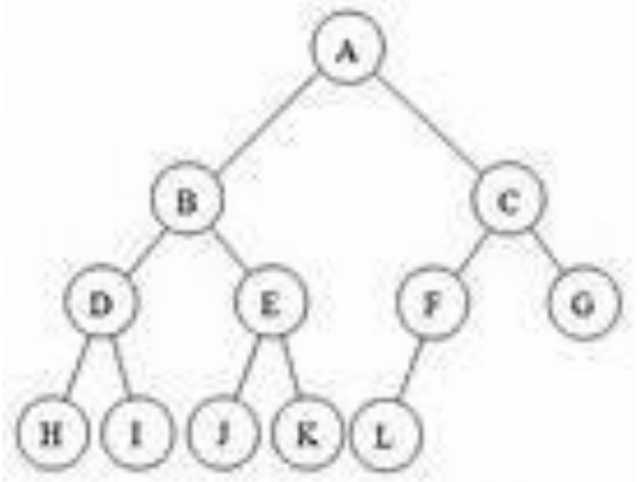


Complete binary trees

- In a **complete binary tree**
  - ➢All the nodes at one level must have values before starting the next level
  - ➢All the nodes in the last level must be completed from left to right

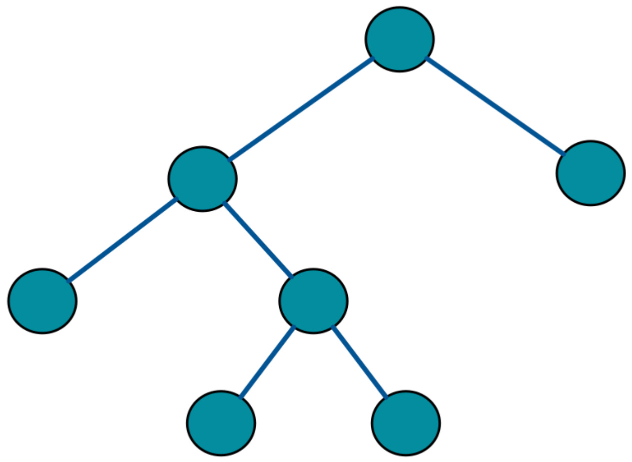


- Warning
  - ➢This notion will come back with the **Heaps**



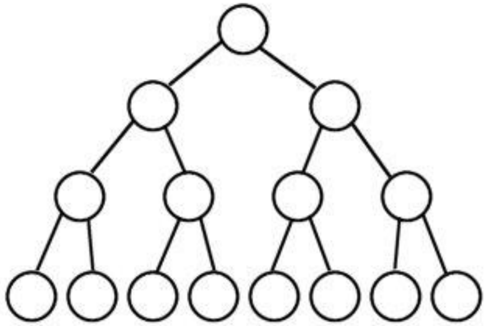
Full binary trees

- Definition:
  - A binary tree in which every node has either **0 or 2 children**



Perfect binary trees

- Definition
  - A binary tree in which all interior nodes have **2 children** and all leaves have the **same** depth or same level
    1. It is a full binary tree
    2. All leaf nodes are at the same level



Perfect binary trees

- **Definitions**
  - # nodes at depth d = 2^d
  - A perfect binary tree of height h has: $2^{h+1} - 1$ nodes
  - Number of leaf nodes in a perfect binary tree of height h = 2^h
  - Number of internal nodes in a perfect binary tree of height h = $2^h - 1$

Number of nodes

- Example:
  - C++ code

```
int countNodes( TreeNode *root ) {
        // Count the nodes in the binary tree to which
        // root points, and return the answer.
    if ( root == NULL )
        return 0;  // The tree is empty.  It contains no nodes.
    else {
        int count = 1;   // Start by counting the root.
        count += countNodes(root->left);  // Add the number of nodes
                                         //     in the left subtree.
        count += countNodes(root->right); // Add the number of nodes
                                         //    in the right subtree.
        return count;  // Return the total.
    }
 } // end countNodes()
```

Binary Trees

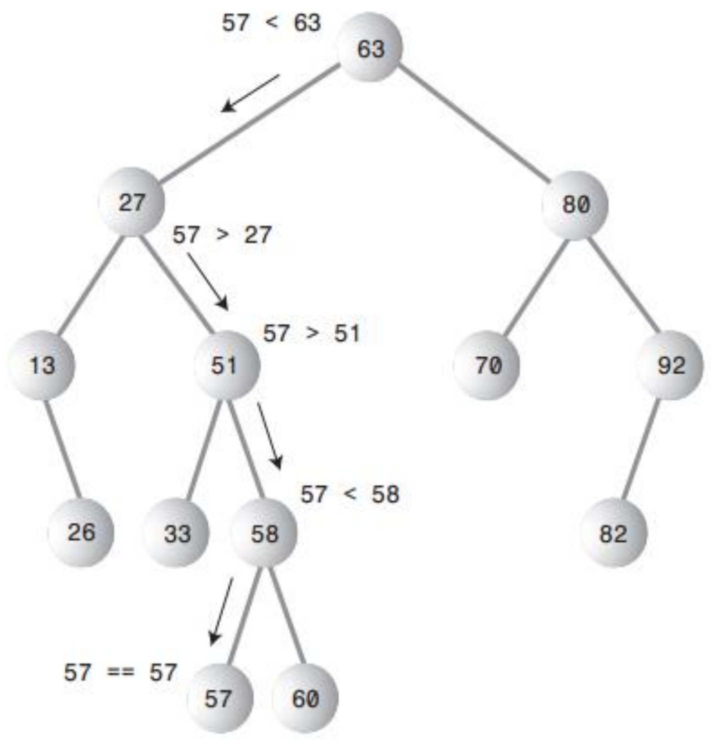
- The first thing needed to represent the tree
  - **Class to represent the Node objects**
- These **Node** objects contain
  - **Data**
    - Representing the objects being stored (int, double, string,…)
    - Example
      - The employees in an employee database
  - **Pointers to**
    - Each of the node’s two children

Finding a Node

- Remember that the Nodes in a **Binary Search Tree** correspond to objects that contain information
- Example, the Nodes might be:
  - **Person objects**
    - With an *employee number* as the key
    - Perhaps *name*, *address*, *telephone number*, *salary* etc.
  - **Car part objects**
    - With a *part number* as the key value
    - Fields for *quantity available*, *price* etc.
- A Node is therefore created with these characteristics, which are kept throughout its life

Finding the Node 57

- Find the Node representing the item with key value 57



Finding a Node - Technique

- This method uses a variable called **current** to hold the node it is currently examining
- The parameter **key** is the value to be found
- The routine starts at the **root** – why?

```
Set current to point to the root
DO
    IF ((current = null) OR (current.data = key))
        Set finished to true
    ELSE
        IF (key < current.data)
            Go to the LEFT (Set current to current.left)
        ELSE
            Go to the RIGHT (Set current to current.right)
WHILE (! finished)
return current
```

Finding a Node

- C++

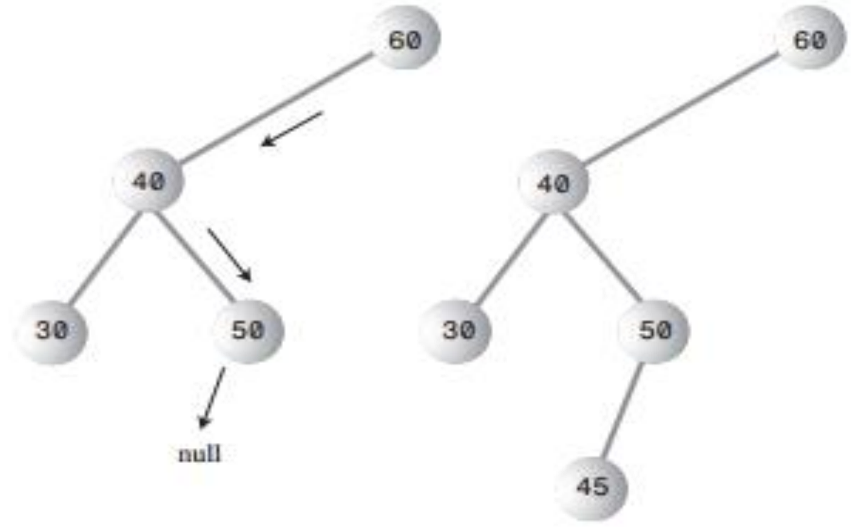
```cpp
bool treeContainsNR( TreeNode *root, string item ) {
// Return true if item is one of the items in the binary
// sort tree to which root points.   Return false if not.
 TreeNode *runner;  // For "running" down the tree.
 runner = root;     // Start at the root node.
 while (true) {
    if (runner == NULL) {
       return false; // fallen off the tree without finding item.
    }
    else if ( item == runner->item ) {
       return true; // We've found the item.
    }
    else if ( item < runner->item ) {
          // If the item occurs, it must be in the left subtree,
          // So, advance the runner down one level to the left.
       runner = runner->left;
    }
    else {
          // If the item occurs, it must be in the right subtree.
          // So, advance the runner down one level to the right.
       runner = runner->right;
    }
 }
}
```

Inserting a Node

- We must find the correct place to insert the node
- We use a similar technique as trying to find a node that turns out not to exist
- The **path** from the **root** to the **appropriate node** is followed
  - This will be the **parent** of the new node
  - The new node is connected as its left or right child
    - This depends on whether the new node's key is less than or greater than that of the parent

Inserting a Node

- Assume we are trying to insert a new Node with the key 45
  - The value **45** is less than **60** and then greater than **40**, we arrive at node **50**
  - As **45** is less than **50** we would now expect to go left BUT **50** has no left child; its leftChild field is null.
  - On seeing this null, the insertion routine has found the place to insert the new node
  - **The algorithm creates a new node with the value 45 and connects it as the left child of 50**
- A place to insert a new node will always be found
  - Unless you run out of memory
    - When a place is found, and the new node is attached



a) Before insertion

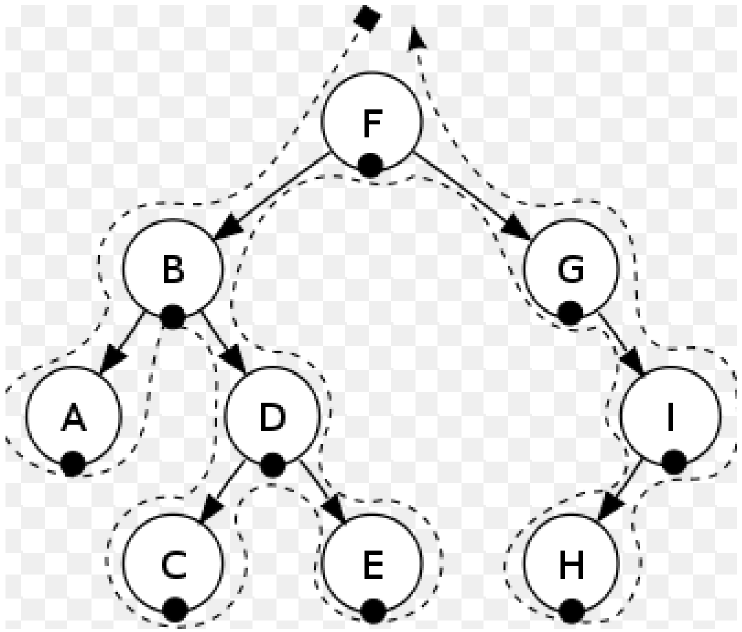
b) After insertion

Traversing the Tree

- Traversing a tree means **visiting each node in a specified order**
- This process is not as common as finding, inserting or deleting nodes
  - Traversal is not particularly fast
- 3 ways to traverse a tree:
  1. **Pre-order**
  2. **In-order**
  3. **Post-order**
- The order most commonly used for binary search trees is **in-order**

In-order Tree Traversal

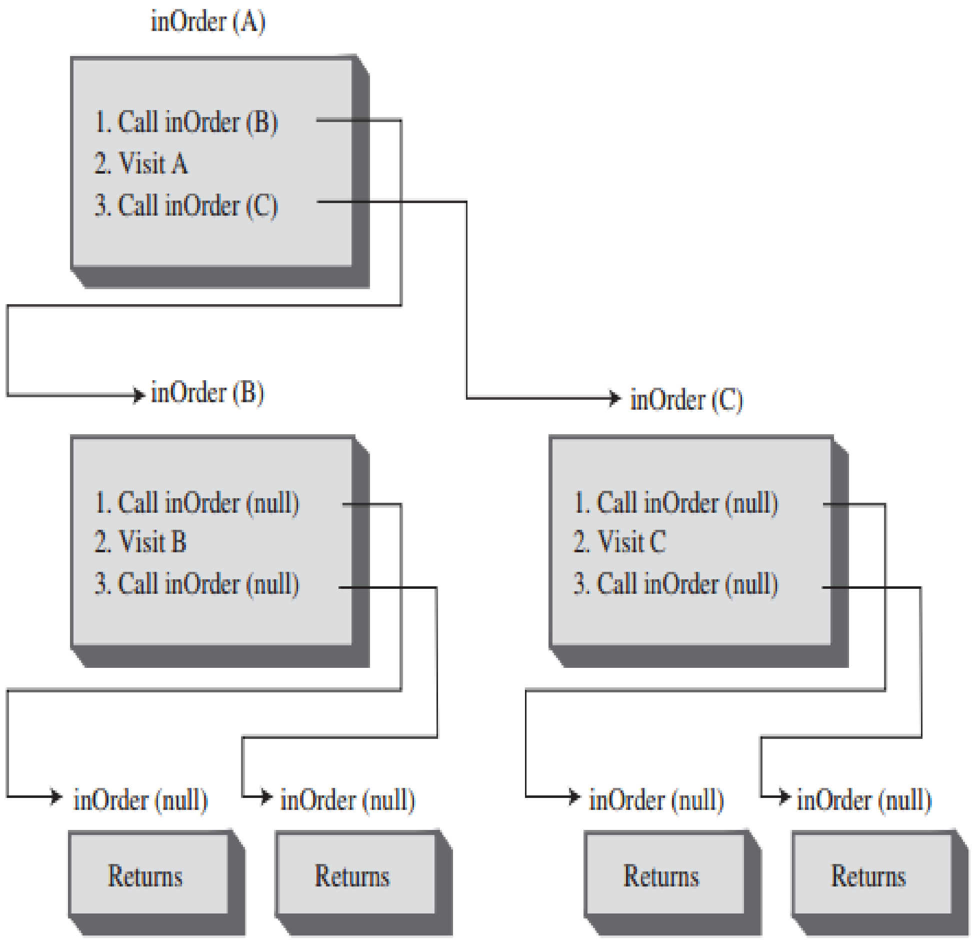
- An **in-order** traversal of a binary search tree will cause all the nodes to be visited in **ascending order**, based on their key values
  - ➢ If you want to create a sorted list of the data in a binary tree, this is one way to do it
- **The simplest way to carry out a traversal is the use of recursion**
  - ➢ A recursive method to traverse the entire tree is called with a node as a parameter

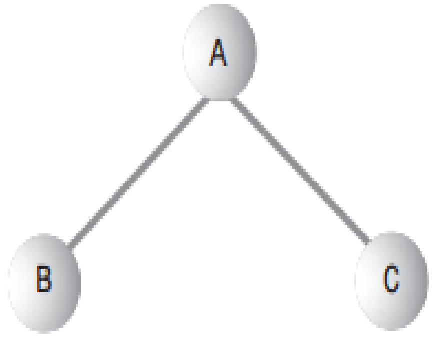


In-order Tree Traversal

- Start at the root
- The **`inorder(…)`** method needs to do only 3 things:
  1. Call itself to traverse the Node's **LEFT** subtree
  2. **VISIT THE NODE**
  3. Call itself to traverse the Node's **RIGHT** subtree
- **Visiting** a Node
  - → doing something to it such as displaying the key, writing it to a file, etc.
- The traversal mechanism does not pay any attention to the key values of the Nodes - it only concerns itself with whether a node has children

Traversing a 3-Node Tree





Pre-order and Post-order Tree Traversals

- You can traverse the tree in 2 other ways:
  - **Pre-order**
  - **Post-order**
- Consider a binary tree that represents an algebraic expression involving the binary arithmetic operators +, -, /, and *
- The root node holds an operator, and the other nodes hold either
  - a variable name (like A, B, or C), or
  - another operator
- Each **subtree** is a valid algebraic expression

Tree Representing an Algebraic Expression

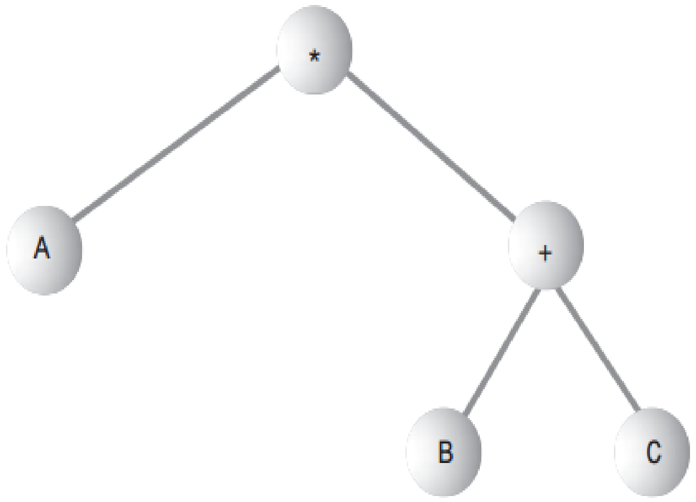
- The binary tree shown represents the algebraic expression

  A * (B + C)

- This is called **infix notation** - the notation normally used in algebra
- Using in-order traversal of the tree will generate the correct in-order sequence:

  A * B + C

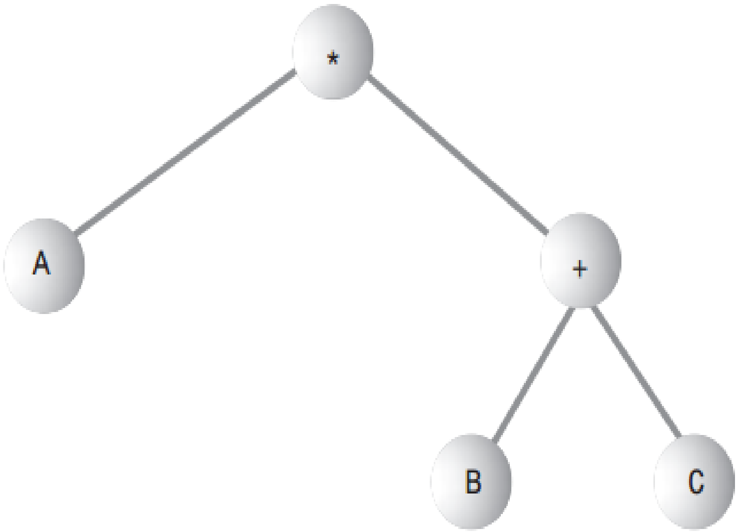
- You need to insert the parentheses yourself



Pre-order Tree Traversal

- What does this have to do with pre-order and post-order traversals?
- For these traversals the same 3 steps are used as for in-order, but in a different sequence
- The sequence for a `preorder(…)` method is:
  - Visit the node
  - Call itself to traverse the node's **left subtree**
  - Call itself to traverse the node's **right subtree**
- Traversing the tree using **preorder** would generate the expression:

  * A + B C

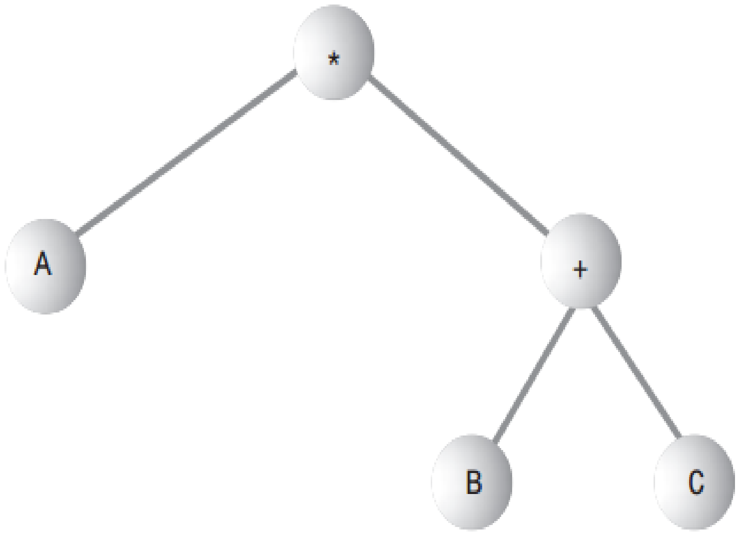


Post-order Tree Traversal

- The sequence for a `postorder(…)` method is as follows:
  - Call itself to traverse the node's **left subtree**
  - Call itself to traverse the node's **right subtree**
  - Visit the node
- For the tree presented, this would generate the expression:

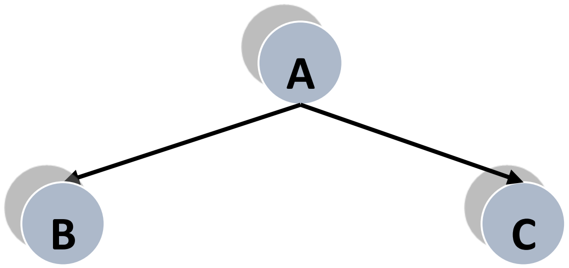
A B C + *

- This is called **postfix notation**
- It means "apply the last operator in the expression, *, to the first and second things"
- The first thing is A, and the second thing is BC+



Tree traversal

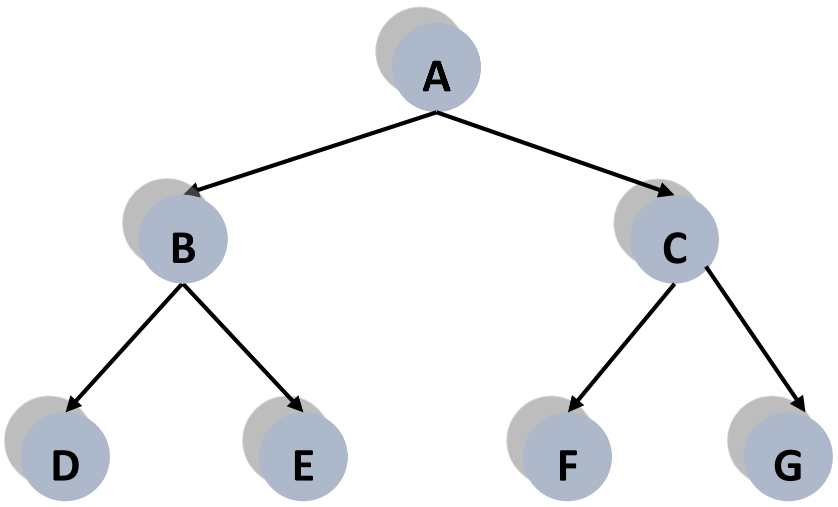
- Order:
  - Start with the root
    - **Pre**: Root **before**
    - **In**: Root in the **middle**
    - **Post**: Root **after**, at the end



Traversal	**Order of Node Visitation**			
Pre-Order	A	B	C	Root → Left → Right
In-Order	B	A	C	Left → Root → Right
Post-Order	B	C	A	Left → Right → Root

Tree traversal

- Example



Traversal	Order of Node Visitation						
PreOrder	A	B	D	E	C	F	G
InOrder	D	B	E	A	F	C	G
PostOrder	D	E	B	F	G	C	A

Tree traversal

- C++ code

```
void preorderPrint( TreeNode *root ) {
        // Print all the items in the tree to which root points.
        // The item in the root is printed first, followed by the
        // items in the left subtree and then the items in the
        // right subtree.
    if ( root != NULL ) {
        cout << root->item << " ";        // Print the root item.
        preorderPrint( root->left );      // Print items in left subtree.
        preorderPrint( root->right );     // Print items in right subtree.
    }
}

void inorderPrint( TreeNode *root ) {
        // Print all the items in the tree to which root points.
        // The items in the left subtree are printed first, followed
        // by the item in the root node, followed by the items in
        // the right subtree.
    if ( root != NULL ) {
        inorderPrint( root->left );      // Print items in left subtree.
        cout << root->item << " ";       // Print the root item.
        inorderPrint( root->right );     // Print items in right subtree.
    }
}

void postorderPrint( TreeNode *root ) {
        // Print all the items in the tree to which root points.
        // The items in the left subtree are printed first, followed
        // by the items in the right subtree and then the item in the
        // root node.
    if ( root != NULL ) {
        postorderPrint( root->left );      // Print items in left subtree.
        postorderPrint( root->right );     // Print items in right subtree.
        cout << root->item << " ";         // Print the root item.
    }
}
```

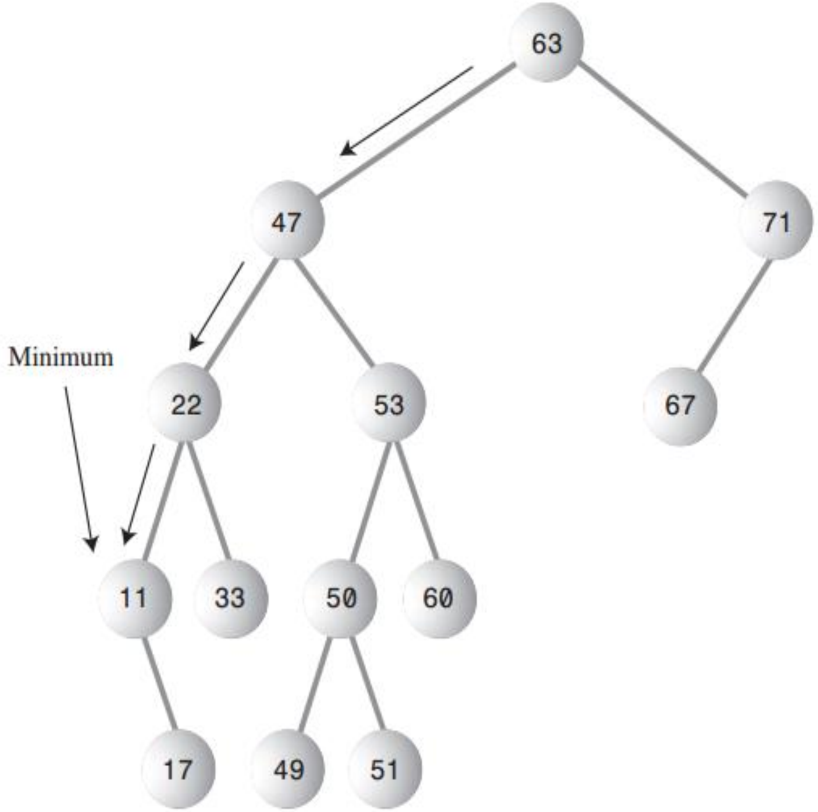
Finding the Minimum Value in a Tree

For the **minimum**,

- Go to the left child of the root;
- Then go to the left child of that child, and so on, until **you come to a node that does not have a left child**.

This node is the minimum.

```
protected Node minimum() {
    Node current = root, last = null;
    while (current != null) {
        last = current;
        current = current.getLeft();
    }//while
    return last;
}//minimum()
```



Finding the Maximum value in a Tree

- For the maximum value in the tree
- follow the same procedure as for the minimum value
- **Go from right child to right child, until you find a node without a right child**
- **This node is the maximum**
- The code is the same except that the last statement in the loop is:

```
current = current.getRight();
```

Finding Minimum and Maximum

- C++ code

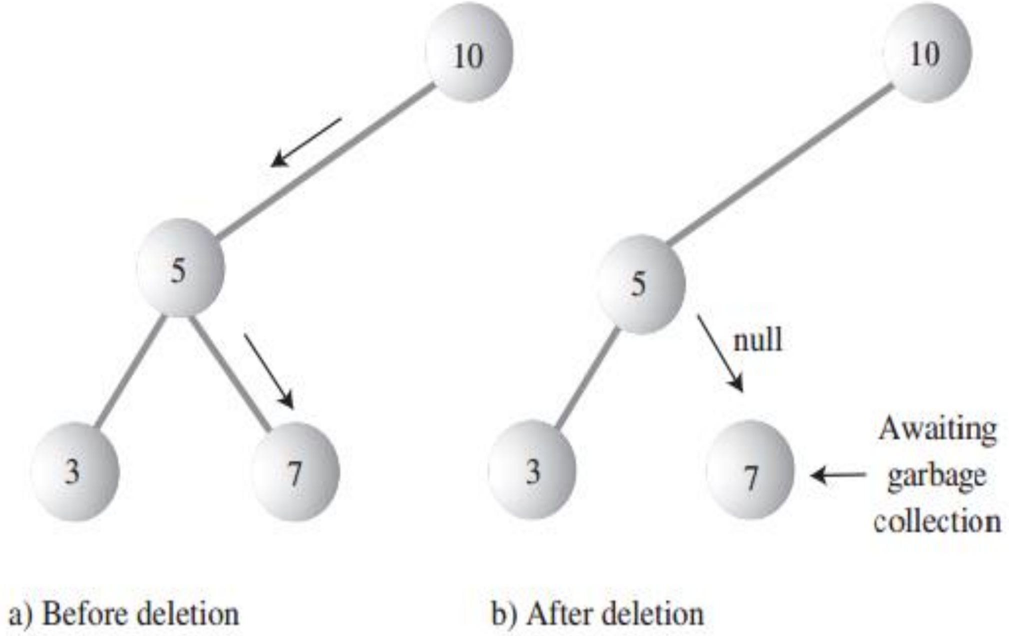
```
int FindMin(TreeNode *root) {
    if (root == NULL) {
        return INT_MAX; // or undefined.
    }
    if (root->left != NULL) {
        return FindMin(root->left); // left tree is smaller
    }
    return root->data;
}

int FindMax(TreeNode *root) {
    if (root == NULL) {
        return INT_MAX; // or undefined.
    }
    if (root->right != NULL) {
        return FindMax(root->right); // right tree is bigger
    }
    return root->data;
}
```

Deleting a Node

- Deleting a node
  - the most complicated common operation required for binary search trees
- You start by finding the node you want to delete
  - using the same approach in **find()** and **insert()**
- Once the node to be deleted has been found
  - 3 cases to consider:
    1. The node to be deleted is a leaf (does not have any children)
    2. The node to be deleted has one child
    3. The node to be deleted has two children

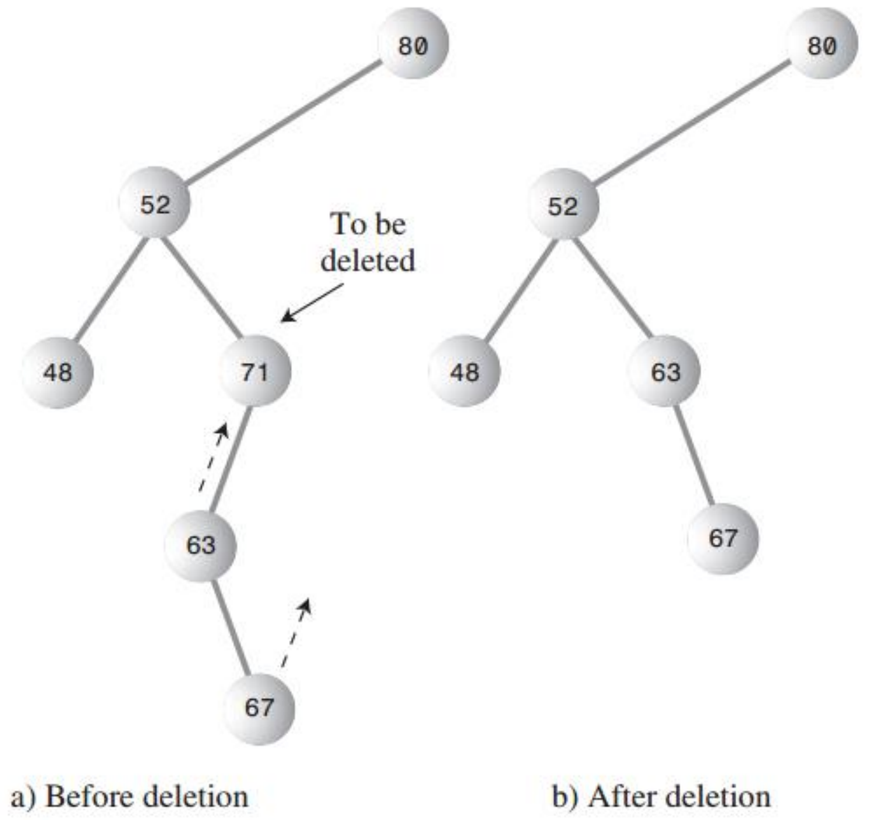
Node to be Deleted – No Children

- To delete a leaf node with NO CHILDREN, change the appropriate child field in the node's parent to point to null, instead of to the node
- The node will still exist
  - but it will no longer be part of the tree



Node to be Deleted – One Child

- In this case, the node only has 2 connections:
  1. to its parent, and
  2. to its only child
- **You need to "snip" the node out of this sequence by CONNECTING its PARENT directly to its CHILD**
- This process involves changing the appropriate reference in the parent (leftChild or rightChild) to point to the deleted node's child

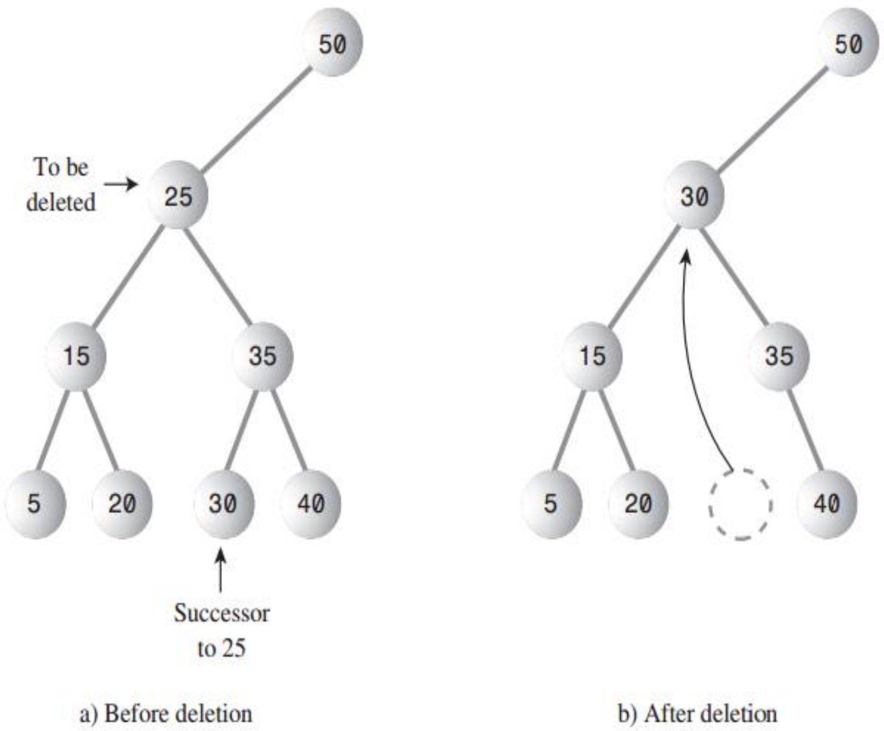


Node to be Deleted – One Child

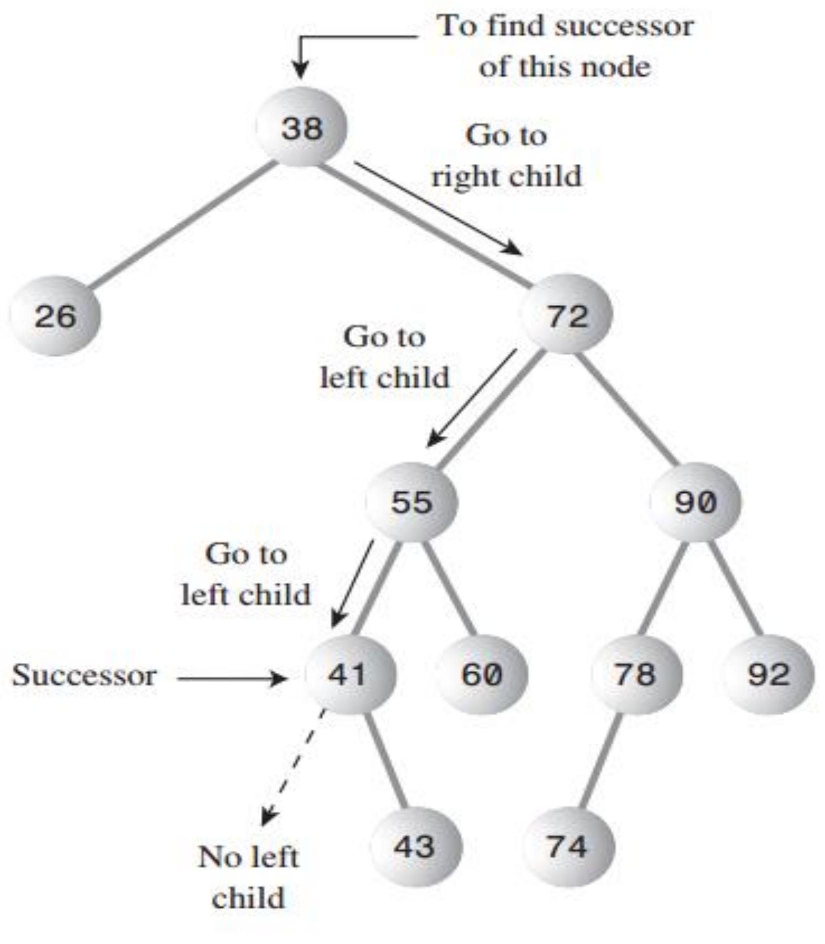
- There are 4 variations of this code:
  - The **SINGLE CHILD OF THE NODE TO BE DELETED** may be either
    - a **LEFT CHILD** or
    - a **RIGHT CHILD**
  - **For each of these 2 cases,** the **NODE TO BE DELETED** may be either
    - The left or
    - The right child of its parent
- Special case
  - The node to be deleted may be the root
  - This has no parent and is simply replaced by the appropriate subtree

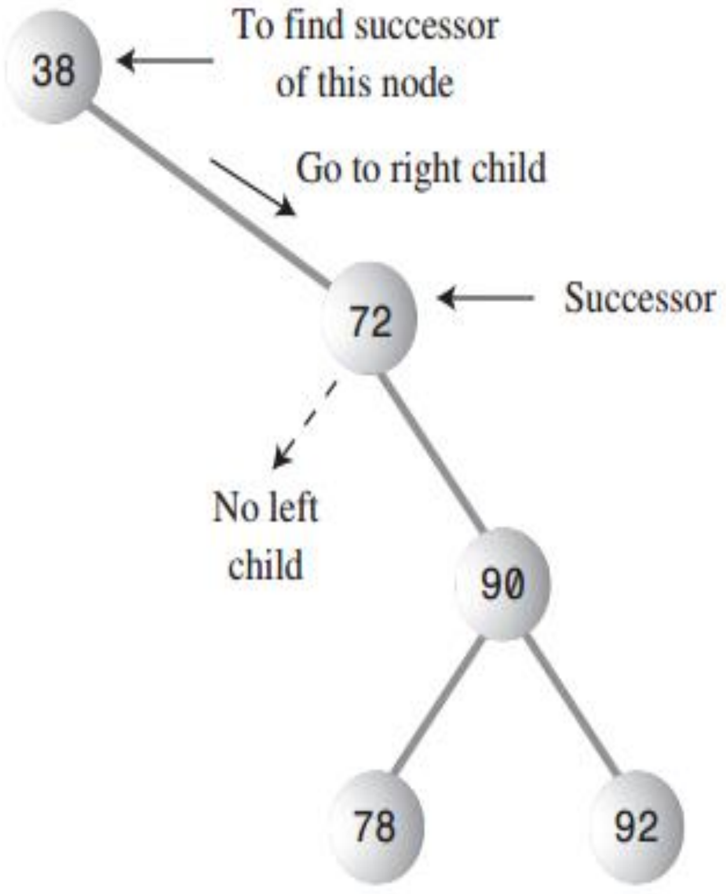
Node to be Deleted – Two Children

- In a **Binary Search Tree** the nodes are arranged in order of ascending keys
- For each node
  - the node with the next-highest key is called its **in-order successor**, or simply its **successor**
    - **"Next element if they are ordered in an array"**
- To delete the node with two children, replace the **node** with its **in-order successor**



Finding the (InOrder) Successor of a Node





Delete a Node

- C++ code

```cpp
struct TreeNode* Delete(struct TreeNode *root, int data) {
  if (root == NULL) {
     return NULL;
  }
  if (data < root->data) {  // data is in the left sub tree.
      root->left = Delete(root->left, data);
  } else if (data > root->data) { // data is in the right sub tree.
      root->right = Delete(root->right, data);
  } else {
     // case 1: no children
     if (root->left == NULL && root->right == NULL) {
        delete(root);
        root = NULL;
     }
     // case 2: 1 child (right)
     else if (root->left == NULL) {
        struct TreeNode *temp = root; // save current node as a backup
        root = root->right;
        delete temp;
     }
     // case 3: 1 child (left)
     else if (root->right == NULL) {
        struct TreeNode *temp = root; // save current node as a backup
        root = root->left;
        delete temp;
     }
     // case 4: 2 children
     else {
        struct TreeNode *temp = FindMin(root->right); // find minimal value of right sub tree
        root->data = temp->data; // duplicate the node
        root->right = Delete(root->right, temp->data); // delete the duplicate node
     }
  }
  return root; // parent node can update reference
}
```

Invert a tree

- Input

```
    4
  /   \
 2     7
/ \   / \
1   3 6   9
```

- Output

```
    4
  /   \
 7     2
/ \   / \
9   6 3   1
```

C++ code

auto: the compiler will deduce the type

```cpp
TreeNode* invertTree(TreeNode* root) {
    if (root == NULL) {
        return NULL; // terminal condition
    }
    auto left = invertTree(root->left);   // invert left sub-tree
    auto right = invertTree(root->right); // invert right sub-tree
    root->left = right; // put right on left
    root->right = left; // put left on right
    return root;
}
```

Max Depth

- Find the maximum depth of a binary tree
  - Useful to check if the tree is balanced or not!
- C++ code
  - 2 versions

```
int MaxDepth(TreeNode *root){
    if(root == NULL)
        return 0;
    else
        return 1 + max(MaxDepth(root->left),MaxDepth(root->right));
}

int MaxDepth(struct TreeNode* root) {
  if (root==NULL) {
    return 0;
  }
  else {
    // compute the depth of each subtree
    int leftDepth = MaxDepth(root->left);
    int rightDepth = MaxDepth(root->right);
    // use the larger subtree
    if (leftDepth > rightDepth)
        return leftDepth+1;
    else
        return rightDepth+1;
  }
}
```

Min Depth

- Find the minimum depth of a binary tree
  - Useful to check if the tree is balanced or not!
- C++ code

```
int MinDepth(TreeNode *root) {
    if (root == NULL)
         return 0;
    // Base case : Leaf Node. This accounts for height = 1.
    if (root->left == NULL && root->right == NULL)
       return 1;
    // If left subtree is NULL, recur for right subtree
    if (!root->left)
       return MinDepth(root->right)+1;
    // If right subtree is NULL, recur for right subtree
    if (!root->right)
       return MinDepth(root->left)+1;
    return min(MinDepth(root->left),MinDepth(root->right)) + 1;
}
```

Comparison of Trees

- Check if 2 data structures contain the same information
  - → Compare 2 binary trees.
- C++ code

```
int SameTrees(struct TreeNode* a, struct TreeNode* b)
{
    // both empty
    if (a==NULL && b==NULL)
        return 1;
    // both non-empty
    if (a!=NULL && b!=NULL) {
        return
        (
            (a->data == b->data) && // same data in the current node
            (SameTrees(a->left,b->left)) && // same left tree
            (SameTrees(a->right,b->right)) // same right tree
        );
    }
    // one empty, one not -> false
    return 0;
}
```

Find if it is a Binary Search Tree

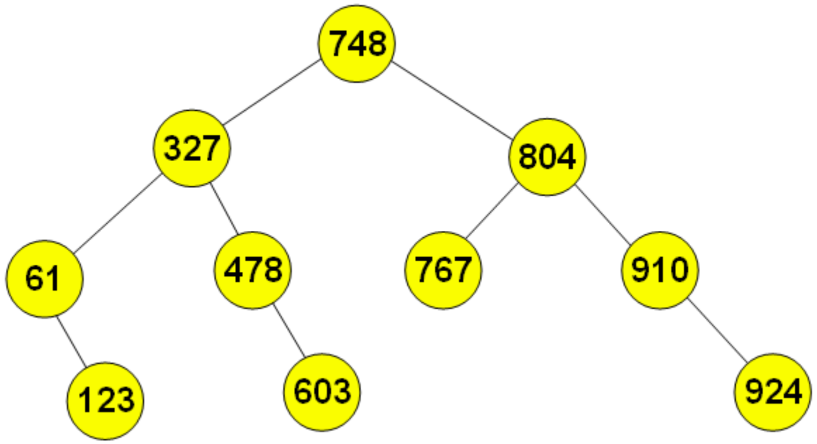
- Verify that the relationships between the different nodes is correct
- C++ code
  - 2 versions

```
int isBSTv1(struct TreeNode* root) {
  if (root==NULL) return(true);
  // false if the max of the left is > than us
  // (bug -- an earlier version had min/max backwards here)
  if (root->left!=NULL && maxValue(root->left) > root->data)
    return(false);
  // false if the min of the right is <= than us
  if (root->right!=NULL && minValue(root->right) <= root->data)
    return(false);
  // false if, recursively, the left or right is not a BST
  if (!isBST(root->left) || !isBST(root->right))
    return(false);
  // passing all that, it's a BST
  return(true);
}
```

```
int isBSTv2(struct TreeNode* root) {
  return(isBSTUtil(root, INT_MIN, INT_MAX));
}
// Returns true if the given tree is a BST and its
// values are >= min and <= max.
int isBSTUtil(struct TreeNode* node,int min,int max) {
  if (node==NULL) return(true);
  // false if this node violates the min/max constraint
  if (node->data<min || node->data>max) return(false);
  // otherwise check the subtrees recursively,
  // tightening the min or max constraint
  return
    isBSTUtil(node->left, min, node->data) &&
    isBSTUtil(node->right, node->data+1, max)
  );
}
```

Searching a Tree

- Number of comparisons to find each value:
  - 748 — 1 → 1
  - 327 & 804 — 2 each → 4
  - 61, 478, 767 & 910 — 3 each → 12
  - 123, 603 & 924 — 4 each → 12



- Total comparisons to find ALL values:

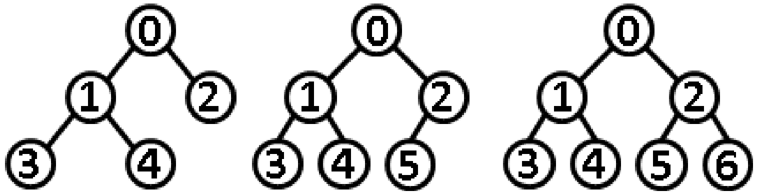
  1 + 4 + 12 + 12 → 29

Average Number of Comparisons

- Average Number of Comparisons
  - **Total number of comparisons / Number of values**
- In our example:
  - **29 / 10**
  - **2.9** comparisons per search

Summary

- Trees consist of nodes (circles) connected by edges (lines)
- The root node is the topmost node in a tree
  - It has no parent
- In a binary tree
  - A node has at most 2 children
  - All the nodes that are left descendants of node A have key values less than A;
  - All the nodes that are A's right descendants have key values greater than (or equal to) A
- Nodes represent the data objects being stored in the tree
- Edges are most commonly represented in a program by references to a node's children
  - sometimes to its parent
- Traversing a tree means visiting all its nodes in some order



full binary tree

complete binary tree

perfect binary tree

Summary

- The simplest traversals:
  - pre-order, in-order and post-order
- An in-order traversal visits nodes in order of ascending keys
- Pre-order and post-order traversals are useful for parsing algebraic expressions
  - When you have to take into account brackets
- Unbalanced tree
  - one whose root has many more left descendants than right descendants (or vice-versa)
- Searching for a node involves
  - Comparing the value to be found with the key value of a node
  - Visiting that node's left child (if the key search value is less)
  - Visiting node's right child (if the search value is greater)
- Insertion involves finding the place to insert the new node and then changing a child field in its new parent to refer to it

Summary

- Deleting a node has 0 children
  - Set the child field in its parent to null
- Deleting a node with 1 child
  - Set the child field in its parent to point to its child
- Deleting a node with 2 children
  - Replace it with its successor
- The successor to a node A can be found
  - by finding the minimum node in the subtree whose root is A's right child
- In a deletion of a node with 2 children, different situations arise, depending on whether the successor is the right child of the node to be deleted or one of the right child's left descendants
- Trees can be represented in the computer's memory as an array although the reference-based approach is more common

Before you finish

- Properly delete your binary tree /!\
  - Delete each node!
    - Each node can contain a pointer to an object that you have created
    - Example
      - Binary Tree of images, Binary Tree of arrays, …
- C++

```
void DestroyTree(TreeNode *root) {
  if(root!=NULL) {
    DestroyTree(root->left);
    DestroyTree(root->right);
    delete root;
  }
}

void MyBinaryTree::~MyBinaryTree() {
  DestroyTree(root);
}
```

Questions ?

- Reading
  - ➢CSci 115 book - Section 7.1
  - ➢Chapter 12, Binary Search Trees, Introduction to Algorithms, 3rd Edition.