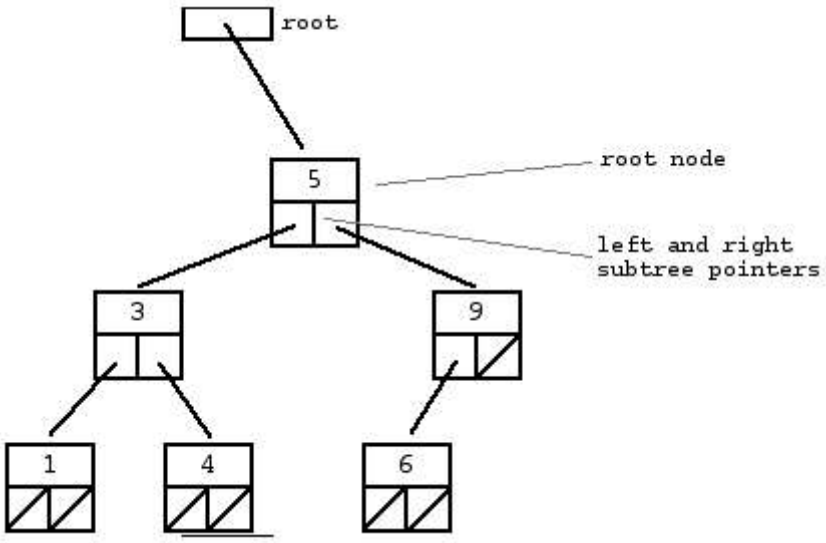
4. Binary Trees

4a. Binary Trees - Ordered or Unordered

Ordered binary tree or Binary search tree(BST) => max(LeftSubTree) <= Node/Root <= min(RightSubTree)

Note - A "binary search tree" is different from a "binary tree".



The nodes at the bottom edge of the tree have empty subtrees and are called "leaf" nodes (1, 4, 6) or external nodes while the others

are "internal" nodes (3, 5, 9).

Basically, binary search trees are fast at insert and lookup. .On average, a binary search tree algorithm can locate a node in an N node tree in

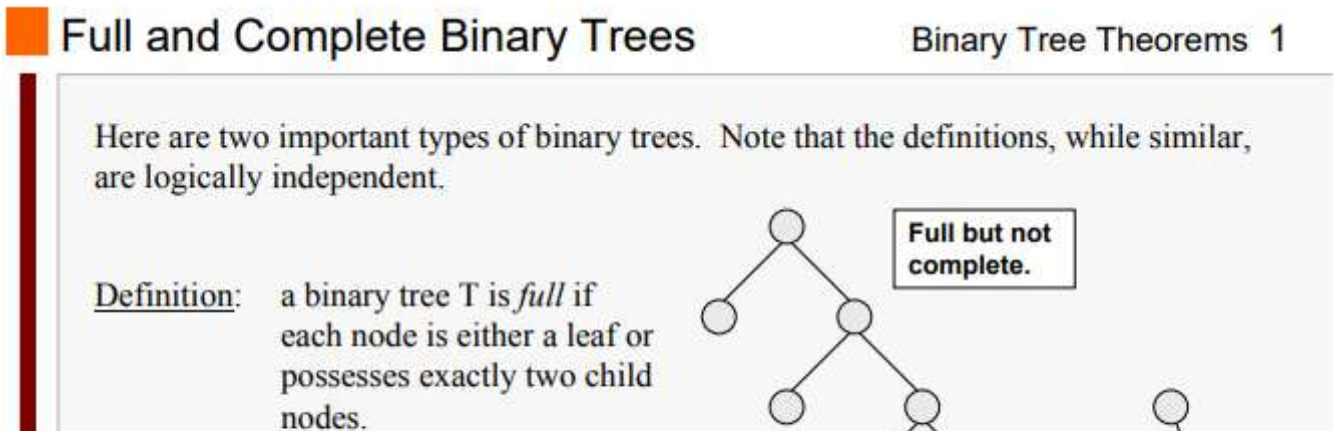
order log(N) time (log base 2). Therefore, binary search trees are good for "dictionary" problems where the code inserts and looks up

information indexed by some key. The lg(N) behavior is the average case -- it's possible for a particular tree to be much slower depending on its shape

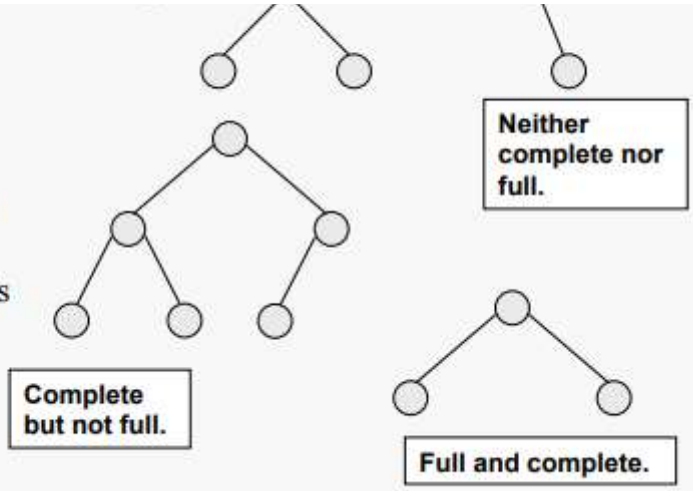
BST for lookup and insert -

Avg. O(log2(N))

Worst O(N)



Definition: a binary tree T with n levels is *complete* if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



Full Binary Tree Theorem

Binary Tree Theorems 2

Theorem: Let T be a nonempty, full binary tree Then:

(a) If T has I internal nodes, the number of leaves is L = I + 1.

(b) If T has I internal nodes, the total number of nodes is N = 2I + 1.

(c) If T has a total of N nodes, the number of internal nodes is I = (N – 1)/2.

(d) If T has a total of N nodes, the number of leaves is L = (N + 1)/2.

(e) If T has L leaves, the total number of nodes is N = 2L – 1.

(f) If T has L leaves, the number of internal nodes is I = L – 1.

Basically, this theorem says that the number of nodes N, the number of leaves L, and the number of internal nodes I are related in such a way that if you know any one of them, you can determine the other two.

Possible Implementation of tree


```
struct TreeNode
{
  Object element;
```

```
    TreeNode *firstChild;/* Pointer to linked list of children */
    TreeNode *nextSibling; /* Pointer to linked list of siblings
    */
}
```

Alternatively: a TreeNode can have pointer to parent, and a *list* (List ADT) of children (each of which is a tree). Class Tree itself can then contain one TreeNode corresponding to *root*. Many implementations possible.

Height and Size

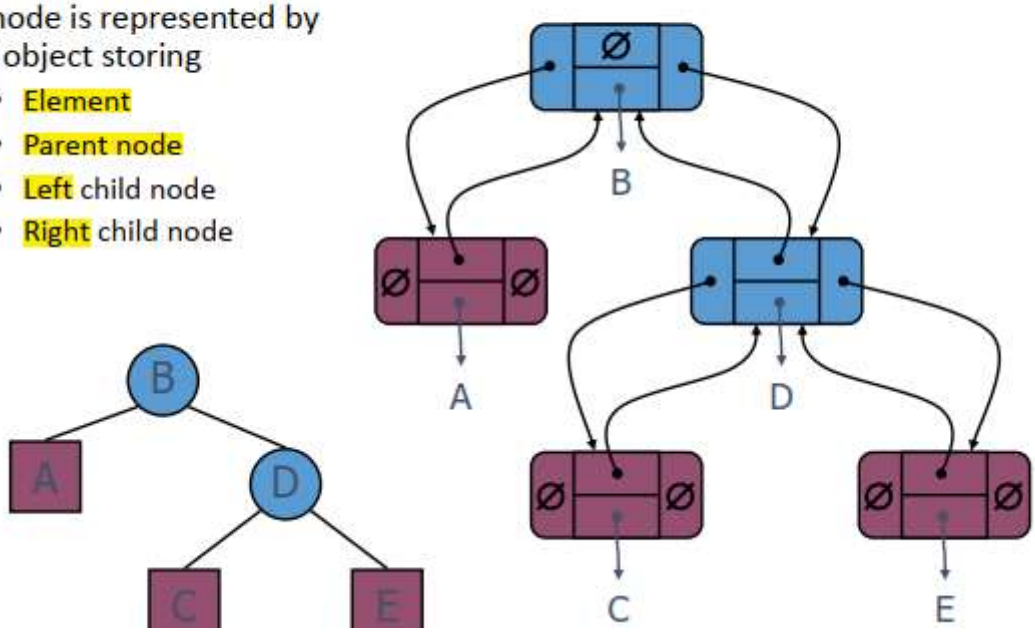
IIT Bombay

- $height(\Phi) = 0$
- $height(plant(n, L, R)) = next(max(height(L), height(R)))$
- Here *max* is the maximum function on numbers defined by
  - $max(0, n) = n$ for all n
  - $max(next(m), n) = next(m)$ if $max(m, n) = m$
    $= n$ otherwise
- $size(\Phi) = 0$
- $size(plant(n, L, R)) = next(add(size(L), size(R)))$

Data Structure for Binary Trees

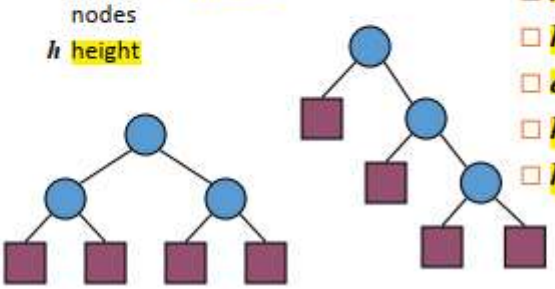
IIT Bombay

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node



Properties of Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes or leaves
  - i number of internal nodes
  - h height



- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

Special Binary Trees

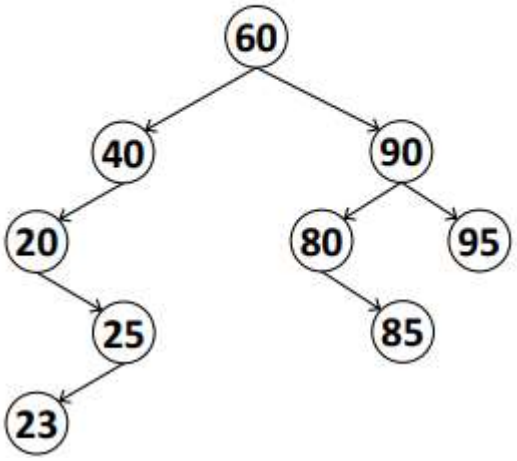
- Some special types of binary trees are particularly useful.
- Full binary trees
  - $full(\Phi) = true$
  - $full(plant(n, L, R)) = false$ if $height(L) \neq height(R)$
    $= full(L)$ and $full(R)$ otherwise
- A full binary tree of height h has size $2^h - 1$
- For every node, the left and right subtree have the same size.

Complete Binary Trees

- $complete(\Phi)$ = true
- $complete(plant(n, L, R)) = full(L)$ and $complete(R)$
  if $height(L) = height(R)$
  $= complete(L)$ and $full(R)$
  if $height(L) = next(height(R))$
  $= false$ otherwise

Binary Tree

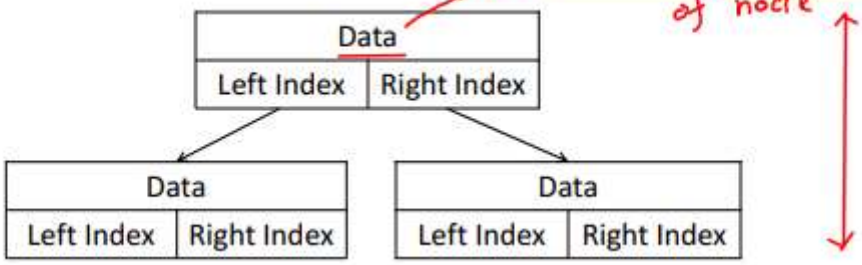
- Nodes
- Parent
  - Left Child
  - Right Child
  - Can have either left, right, or both
- Leaf Node
  - Does not have left and right child



Represented Internally as

- Each node in the tree is represented with
  - Data: The element
  - leftIndex: Index of the left child
  - rightIndex: Index of the right child

could also include height of node



Basic Operations

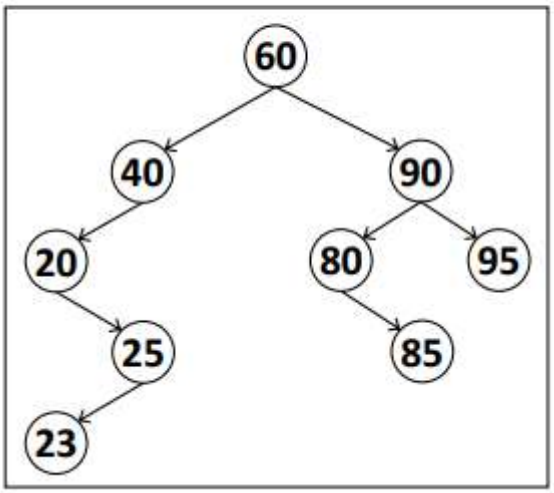
- Making a node (Root)
- Inserting
  - Insert Left node
  - Insert Right node
- Traversing
  - Pre-Order
  - In-Order
  - Post-Order

Pre-Order (Inorder sorts the list)

IIT Bombay

Pre-Order
Display Root
Traverse Left recursively
Traverse Right recursively

60, 40, 20, 25, 23, 90, 80, 85, 95

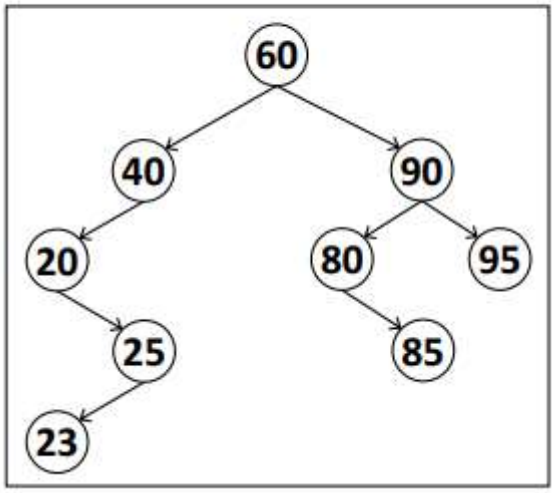


In-Order

IIT Bombay

In-Order
Traverse Left recursively
Display Root
Traverse Right recursively

20, 23, 25, 40, 60, 80, 85, 90, 95

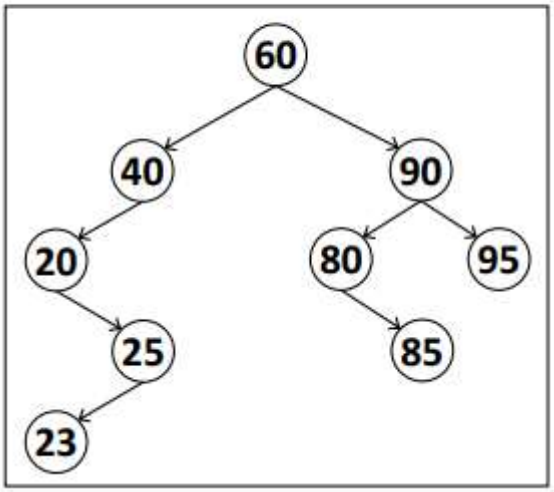


Post-Order

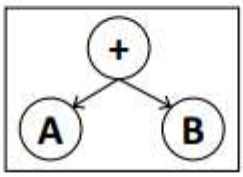
IIT Bombay

Post-Order
Traverse Left recursively
Traverse Right recursively
Display Root

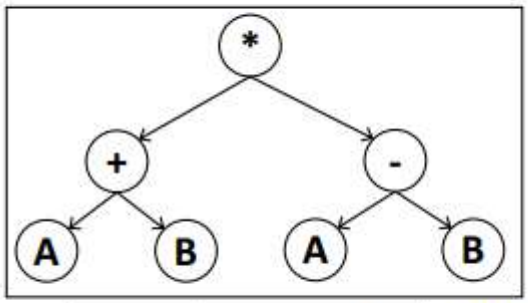
23, 25, 20, 40, 85, 80, 95, 90, 60



Applications



Infix	A+B
Prefix	+AB
Postfix	AB+



Infix	(A+B) * (A-B)
Prefix	* +AB - AB
Postfix	AB+ AB- *

Best Expression

Data Structures

- **Structure 'Node'**: Represent each node as
  - Data
  - Left Index
  - Right Index
- **Vector 'bt'**: Represents binary tree or could use a list
  - It is of type structure 'Node'

Efficiency of Binary Search Trees

- A binary search tree allows many operations on sets to be performed efficiently.
- The time for find, insert, delete depends on the height of the tree rather than the size.
- Most binary trees have much smaller height compared to their size.
- More efficient than using sequences for representing sets.

Sorting

- An inorder traversal on a binary search tree will order the nodes of the tree in increasing order of their values.
- Gives a way of sorting values of any type T.

- Start with an empty tree.
- Insert the values in the tree.
- Do an inorder traversal of the tree.

- Used to compare two sets, test for equality of sets represented by binary search trees.

// The most important traversal is inorder traversal
// 1. An inorder traversal on a binary search tree will order the nodes
// of the tree in increasing order of their values.
// 2. An inorder traversal gives the true form of arithmetic expressions

Leetcode notes

Level-order Traversal - Introduction

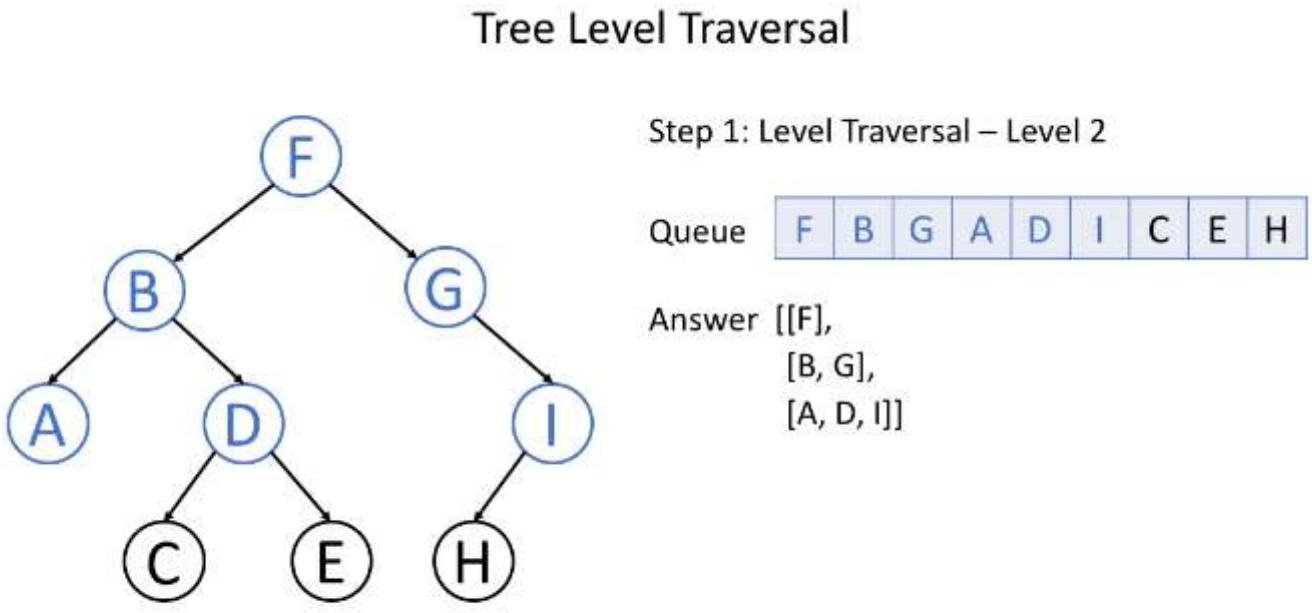
Level-order traversal is to traverse the tree level by level.

Level order traversal of a tree is breadth-first traversal for the tree.

Breadth-First Search is an algorithm to traverse or search in data structures like a tree or a graph. The algorithm starts with a root node and visit the node itself first. Then traverse its neighbors, traverse its second level neighbors, traverse its third level neighbors, so on and so forth.

When we do breadth-first search in a tree, the order of the nodes we visited is in level order.

Here is an example of level-order traversal:



Typically, we use a queue to help us to do BFS. If you are not so familiar with the queue, you can find more information about that in another card coming soon.

Algo for BFS using a queue

```
queue of node* as Q
Q.push(root)
while(Q is not empty){
   node* current = Q.front()
   print current value
   enqueue both children of the node
      if(current->left!=NULL) Q.push(current->left)
      if(current->right!=NULL) Q.push(current->right)
   dequeue the front node or Q.pop()
   }

// 15.  Binary Tree Level Order Traversal - There are two methods of level order traversal or breadth first search approach.
//      (Both are important)
// 15a. Without using a queue. But using two functions 1. maxDepth() and 2. printOneLevel(). # Really IMPORTANT (Learn It)
//      This is useful when we need to keep a track of all nodes at a particular level
// I.   maxDepth-1 gives the height of the given binary tree i.e. levels 0, 1, 2, 3, ..., n for tree of height n.
// II.  printOneLevel() function prints all the nodes in that level.
void printOneLevel(node* root, int level){      // The root and the level at which all the nodes are to be printed are passed
    if(root==NULL) return;
    if(level==0) cout<<root->data<<" ";         // Print only the nodes of a given level
    else if(level>0){                           // If the level > 0 => not reached the required level yet
        printOneLevel(root->left, level-1);     // Prints both the left and right nodes in a given level
        printOneLevel(root->right, level-1);
    }
}

void printBFSorLOT(node* root){
    if(root==NULL) cout<<"The given binary tree is empty\n";
    else {                                      // We cannot write a function
```

```
for maxHeight directly but use the fact h = mD - 1
        int height = maxDepth(root)-1;          // As max depth calculates
the max number of nodes in a path
        for(int i=0; i<=height; i++){           // while height is the max
number of connections in a path i.e. maxDepth - 1
            cout<<"[ ";
            printOneLevel(root, i);             // Prints all the nodes for
the given level i which itself varies from 0 to height
            cout<<"]";                          // thus prints all the nodes
in BFS or LOT.
        }
    }
}

// 15b. Using a queue(STL) to print all the nodes in BFS/LOT. Not useful in
tracking all the nodes in a particular level.
void printBFSusingQ(node* root){                // # Really IMPORTANT (Learn
It)
    if(root==NULL){
        cout<<"The given binary tree is empty \n";
        return;
    }
    queue<node*> Q;     // Need #include<queue> or can create our own class
for queue using three functions push(), front() & pop()
    Q.push(root);
    while(!Q.empty()){
        node* curr = Q.front();
        cout<<curr->data<<" ";
        if(curr->left!=NULL) Q.push(curr->left);
        if(curr->right!=NULL) Q.push(curr->right);
        Q.pop();
    }
}
```

Solve Tree Problems Recursively

- "Top-down" Solution
- "Bottom-up" Solution
- Conclusion

In previous sections, we have introduced how to solve tree traversal problems recursively. Recursion is one of

In previous sections, we have introduced how to solve tree traversal problems recursively. Recursion is one of the most powerful and frequently used techniques for solving tree problems.

As we know, a tree can be defined recursively as a node(the root node) that includes a value and a list of references to children nodes. Recursion is one of the natural features of a tree. Therefore, many tree problems can be solved recursively. For each recursive function call, we only focus on the problem for the current node and call the function recursively to solve its children.

Typically, we can solve a tree problem recursively using a top-down approach or using a bottom-up approach.

"Top-down" Solution

"Top-down" means that in each recursive call, we will visit the node first to come up with some values, and pass these values to its children when calling the function recursively. So the "top-down" solution can be considered as a kind of **preorder** traversal. To be specific, the recursive function top_down(root, params) works like this:

```
1. return specific value for null node
2. update the answer if needed                      // answer <-- params
3. left_ans = top_down(root.left, left_params)      // left_params <-- root.val, params
4. right_ans = top_down(root.right, right_params)   // right_params <-- root.val, params
5. return the answer if needed                      // answer <-- left_ans, right_ans
```

For instance, consider this problem: given a binary tree, find its maximum depth.

We know that the depth of the root node is 1. For each node, if we know its depth, we will know the depth of its children. Therefore, if we pass the depth of the node as a parameter when calling the function recursively, all the nodes will know their depth. And for leaf nodes, we can use the depth to update the final answer. Here is the pseudocode for the recursive function maximum_depth(root, depth):

```
1. return if root is null
2. if root is a leaf node:
3.       answer = max(answer, depth)         // update the answer if needed
4. maximum_depth(root.left, depth + 1)       // call the function recursively for left child
5. maximum_depth(root.right, depth + 1)      // call the function recursively for right chil
```

```
int answer;                     // don't forget to initialize answer before
call maximum_depth
void maximum_depth(TreeNode* root, int depth) {
    if (!root) {
        return;
    }
```

```
    if (!root->left && !root->right) {
        answer = max(answer, depth);
    }
    maximum_depth(root->left, depth + 1);
    maximum_depth(root->right, depth + 1);
}
```

"Bottom-up" Solution

"Bottom-up" is another recursive solution. In each recursive call, we will firstly call the function recursively for all the children nodes and then come up with the answer according to the returned values and the value of the current node itself. This process can be regarded as a kind of **postorder** traversal. Typically, a "bottom-up" recursive function bottom_up(root) will be something like this:

```
1. return specific value for null node
2. left_ans = bottom_up(root.left)          // call function recursively for left child
3. right_ans = bottom_up(root.right)        // call function recursively for right child
4. return answers                           // answer <-- left_ans, right_ans, root.val
```

Let's go on discussing the question about maximum depth but using a different way of thinking: for a single node of the tree, what will be the maximum depth x of the subtree rooted at itself?

If we know the maximum depth l of the subtree rooted at its **left** child and the maximum depth r of the subtree