NANYANG TECHNOLOGICAL UNIVERSITY

CE1007/CZ1007 DATA STRUCTURES

Lecture 08: **Binary Tree Traversal**

Dr. Owen Noel Newton Fernando

College of Engineering

School of Computer Science and Engineering

OUTLINE

- Tree traversal order
  - Pre-order
  - In-order
  - Post-order
- Application examples
  - Count nodes in a binary tree
  - Find grandchild nodes
  - Calculate height of every node
- Level-by-level traversal
- Preorder traversal with a stack

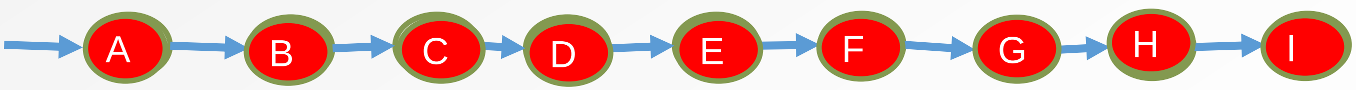
OUTLINE

- **Tree traversal order**
  - **Pre-order**
  - **In-order**
  - **Post-order**
- Application examples
  - Count nodes in a binary tree
  - Find grandchild nodes
  - Calculate height of every node
- Level-by-level traversal
- Preorder traversal with a stack

ARRAY & LINKED LIST TRAVERSAL

	[0]	[1]	[2]	[3]	[4]	[5]
Arr[i]	1	2	3	4	5	6

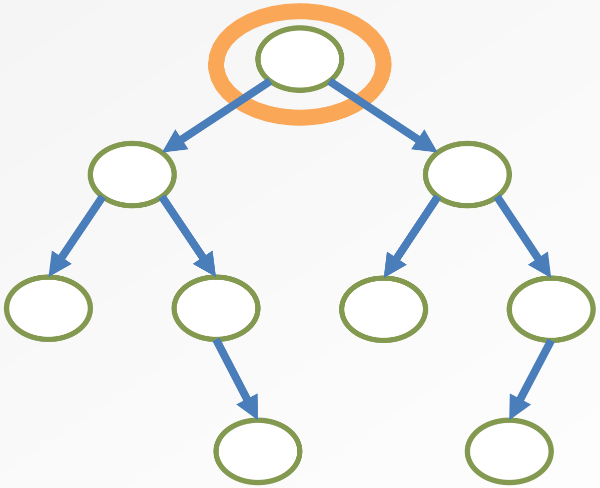
i = i + 1;

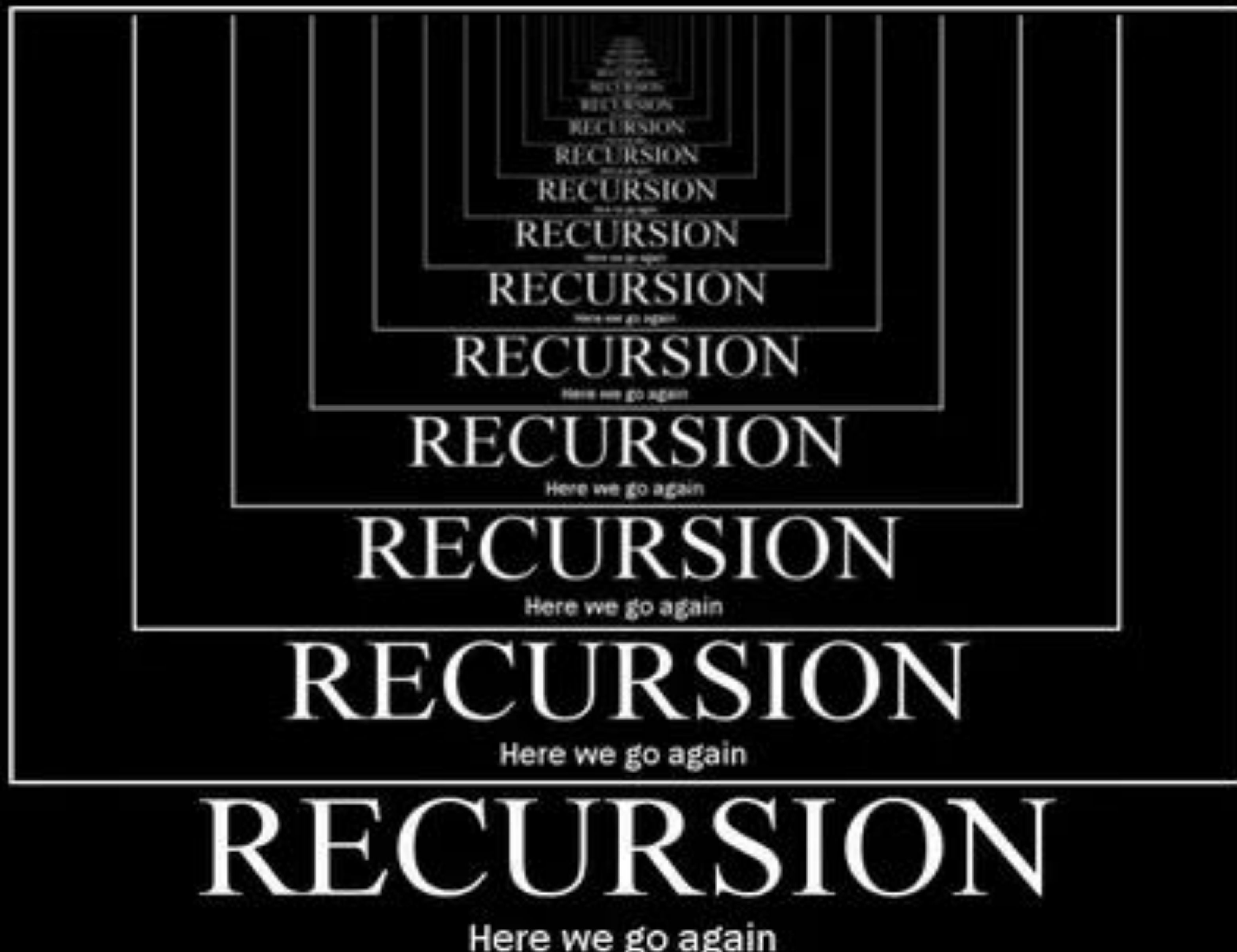


cur = cur->next ;

TREE TRAVERSAL

- Tree traversal:
  - Visiting <u>every</u> node in some systematic manner
  - Procedure should be clearly-defined and repeatable
  - Should not have repeated paths or repeated visits to nodes



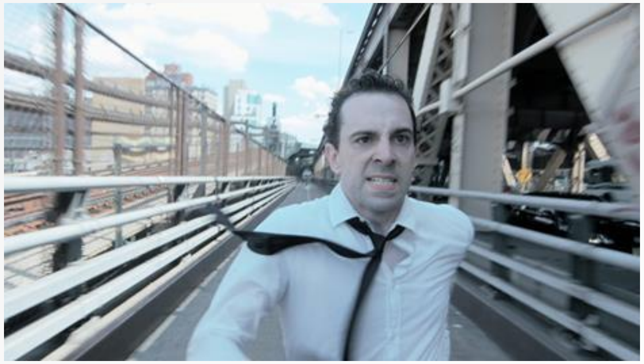

RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again
RECURSION
Here we go again

People often joke that in order to understand recursion, you must first understand recursion.

- *John D. Cook*

A RECURSION MOVIE

["RECURSION", Short Film by Sam Buntrock (US)]

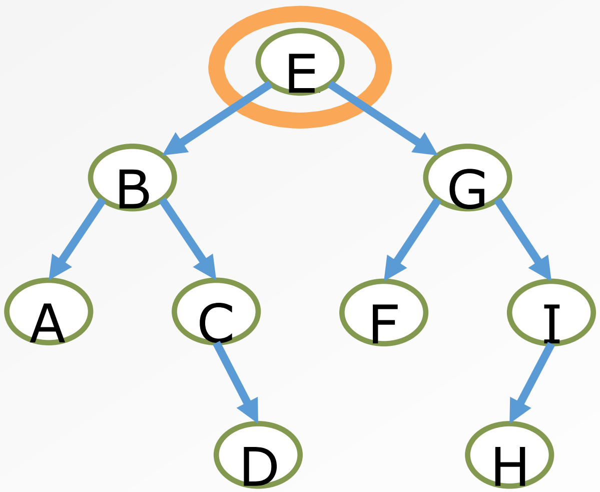


http://www.recursionshortfilm.com/about.php

'**Recursion**' is the modern-day time-travel adventure of Sherwin, a best man who travels back in time after losing a wedding ring.

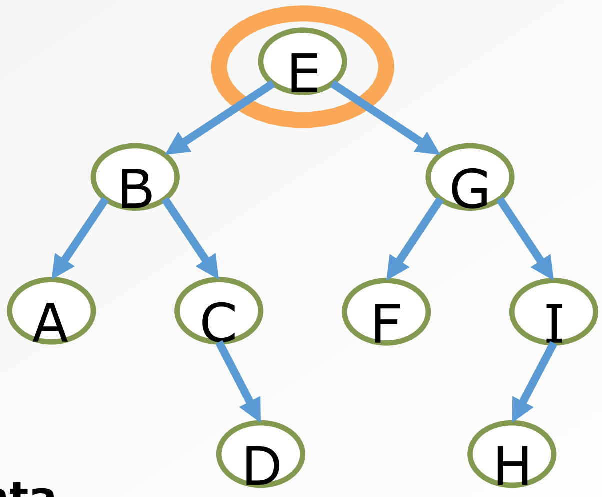
THREE "STANDARD" WAYS TO TRAVERSAL

- Pre-order
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- In-order
- Post-order



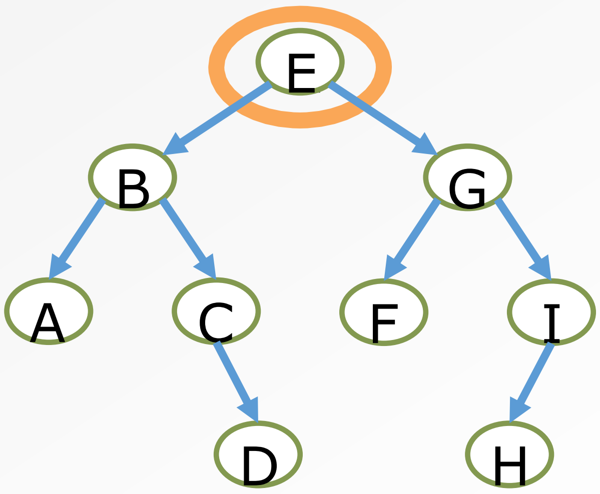
THREE “STANDARD” WAYS TO TRAVERSAL

- Pre-order
  - Process the current node’s data
  - Visit the left child subtree
  - Visit the right child subtree
- **In-order**
  - **Visit the left child subtree**
  - **Process the current node’s data**
  - **Visit the right child subtree**
- Post-order



THREE "STANDARD" WAYS TO TRAVERSAL

- Pre-order
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- In-order
  - Visit the left child subtree
  - Process the current node's data
  - Visit the right child subtree
- **Post-order**
  - **Visit the left child subtree**
  - **Visit the right child subtree**
  - **Process the current node's data**



TREE TRAVERSAL - PRINT

- Recall the TreeTraversal() template (TT) – **Pre-order** :
  - Simple task at each node: <u>print out</u> data in that node

```
void TreeTraversal(BTNode *cur){
    if (cur == NULL)
        return;

    // Do something with the current node's data

    TreeTraversal(cur->left); //Visit the left child node
    TreeTraversal(cur->right);//Visit the right child node
}
```

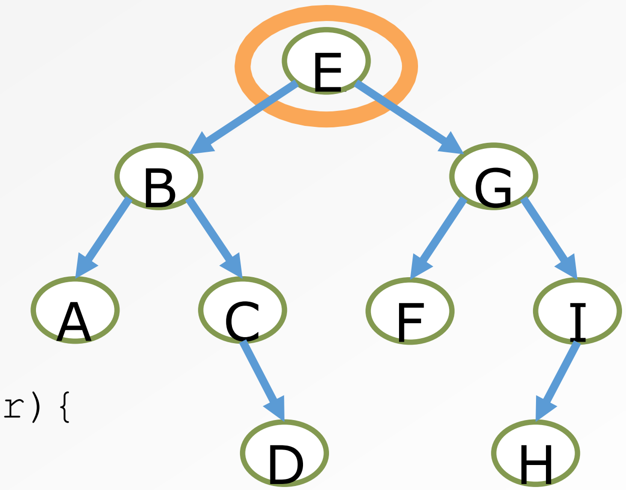
- Recall the TreeTraversal() template (TT) – **Pre-order** :
  - Simple task at each node: <u>print out</u> data in that node

```
void TreeTraversal(BTNode *cur){
    if (cur == NULL)
        return;

    printf("%c",cur->item);

    TreeTraversal(cur->left); //Visit the left child node
    TreeTraversal(cur->right);//Visit the right child node
}
```

TREE TRAVERSAL PRE-ORDER: PRINT

Output:

E B A C D G F I H

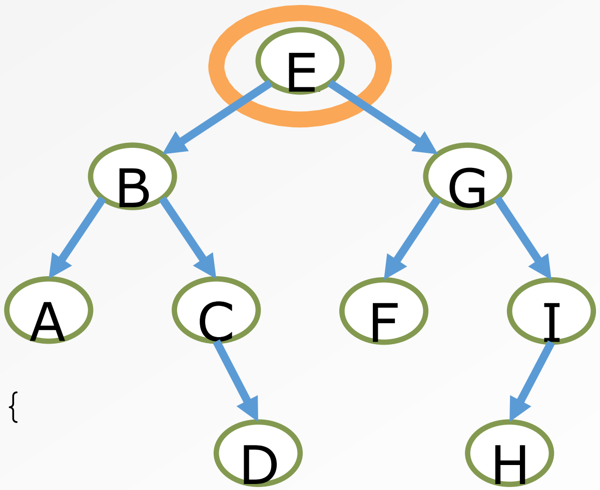



```
void TreeTraversal_pre(BTNode *cur){
    if (cur == NULL)
        return;

    printf("%c  ",cur->item);

    TreeTraversal_pre(cur->left); //Visit the left child node
    TreeTraversal_pre(cur->right);//Visit the right child node
}
```

TREE TRAVERSAL IN-ORDER: PRINT

Output:

A B C D E F G H I

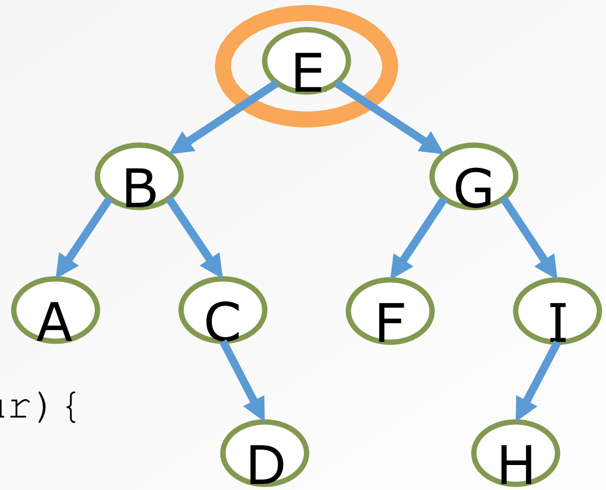



```
void TreeTraversal_in(BTNode *cur){
    if (cur == NULL)
        return;

    TreeTraversal_in(cur->left); //Visit the left child node
    printf("%c  ",cur->item);
    TreeTraversal_in(cur->right);//Visit the right child node
}
```

TREE TRAVERSAL POST-ORDER: PRINT

Output:

A D C B F H I G E




```
void TreeTraversal_post(BTNode *cur){
    if (cur == NULL)
        return;

   TreeTraversal_post(cur->left); //Visit the left child node
   TreeTraversal_post(cur->right);//Visit the right child node
   printf("%c  ",cur->item);
}
```

PRE-ORDER, IN-ORDER AND POST-ORDER

Pre-Order Traversal

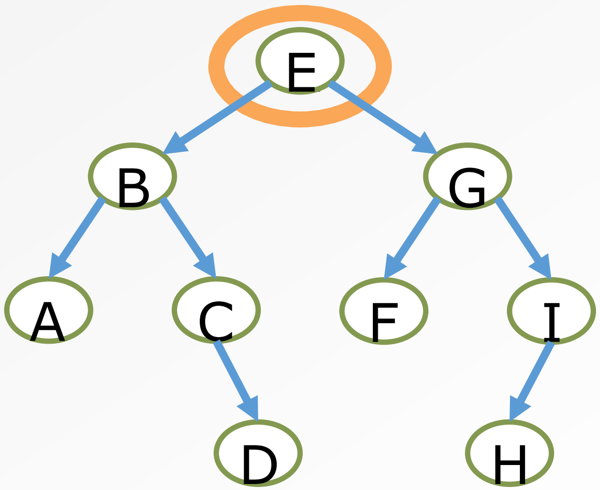
E B A C D G F I H

In-Order Traversal

A B C D E F G H I

Post-Order Traversal

A D C B F H I G E



Once we know how to traverse a Binary Tree,

we can do more based on this …

OUTLINE

- Tree traversal order
  - Pre-order
  - In-order
  - Post-order
- Application examples
  - **Count nodes in a binary tree**
  - Find grandchild nodes
  - Calculate height of every node
- Level-by-level traversal
- Preorder traversal with a stack

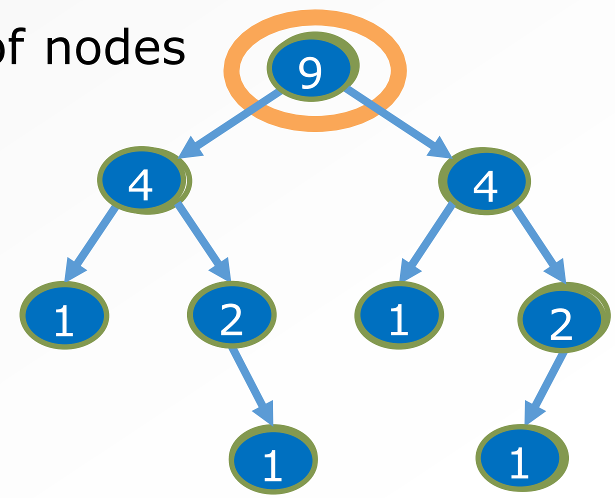
COUNT NODES IN A BINARY TREE

- Recursive definition:
  - Number of nodes in a tree

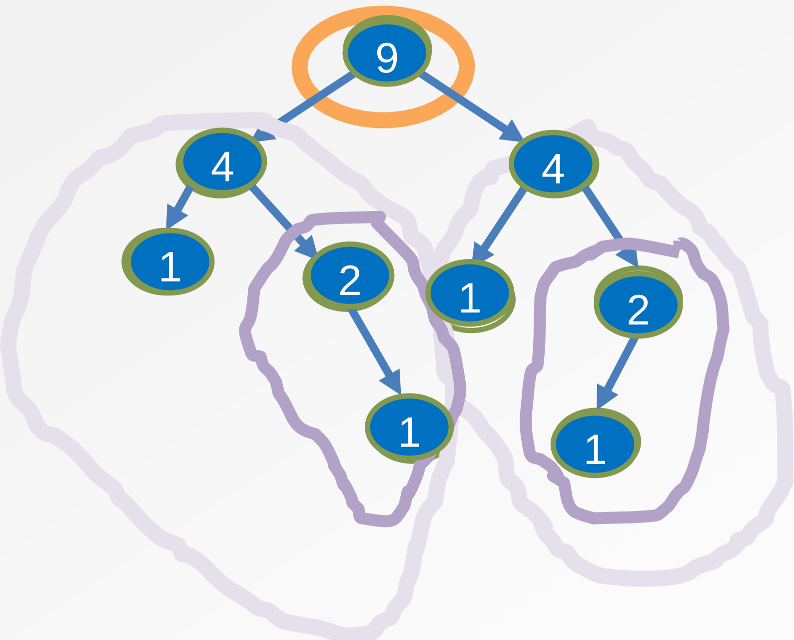
    = 1

      + number of nodes in left subtree

      + number of nodes in right subtree
- Each node returns the number of nodes in its subtree



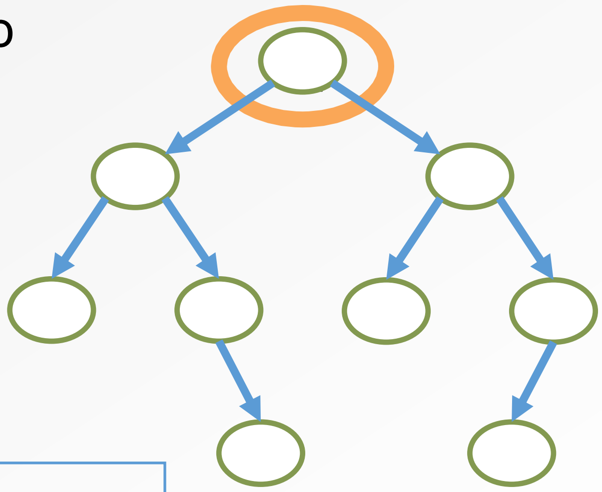
COUNT NODES IN A BINARY TREE



- Each node returns the number of nodes in its own subtree
- Leaf nodes return 1
  Information **propagates upwards** as TreeTraversal returns from visiting leaf nodes
- Which is the first/last count to be returned?

countNode()

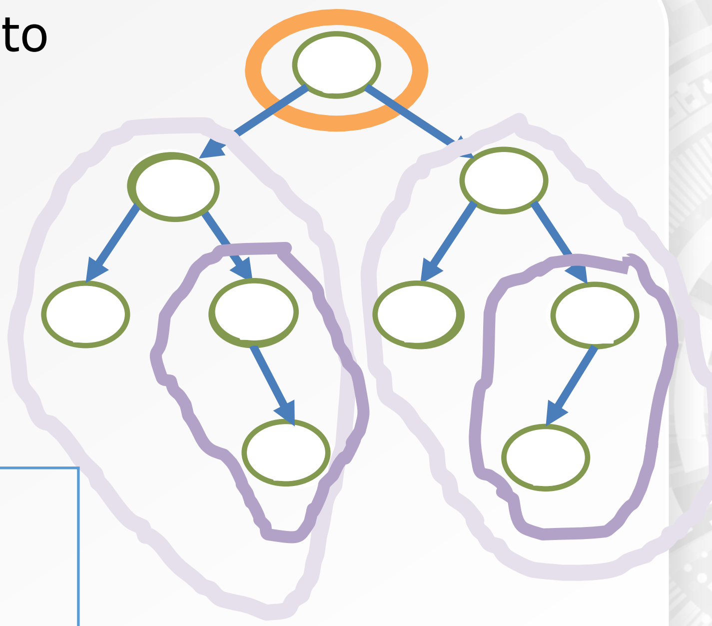
- Return the size of your subtree to your parent node
- Leaf nodes must return 1 to parent node
- Root node returns size of entire tree



```
void TreeTraversal(BTNode *cur){
    if (cur == NULL)
        return;
    //may do something with cur;
    TreeTraversal(cur->left);
    TreeTraversal(cur->right);
    //may do something with cur;
}
```

countNode()

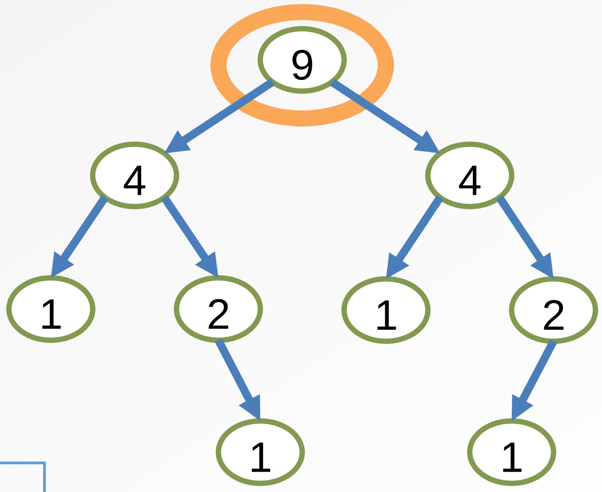
- Return the size of your subtree to your parent node
- Leaf nodes must return 1 to parent node
- Root node returns size of entire tree



```
int countNode(BTNode *cur){
    if (cur == NULL)
        return ???;

    countNode(cur->left);
    countNode(cur->right);
    ??? //sum and get total;
}
```

countNode()

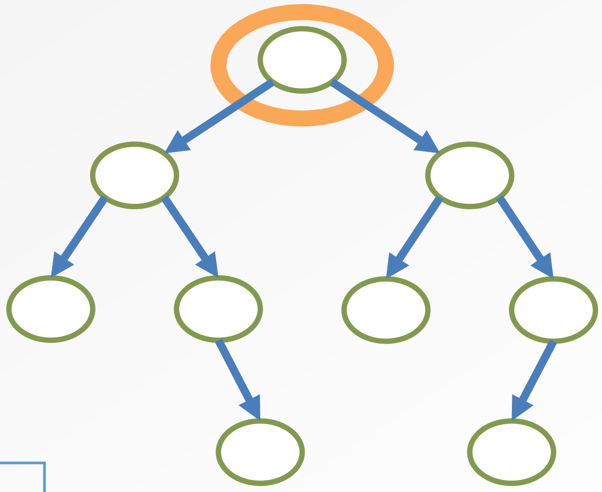
- Leaf nodes must return 1
    - "Null" nodes should return 0
- Leaf node returns 1 + 0 + 0




```
int countNode(BTNode *cur){
    if (cur == NULL)
        return 0;

l = countNode(cur->left);
r = countNode(cur->right);
    return l+r+1;
}
```

countNode()

- Leaf nodes must return 1
  - "Null" nodes should return 0
- Leaf node returns 1 + 0 + 0



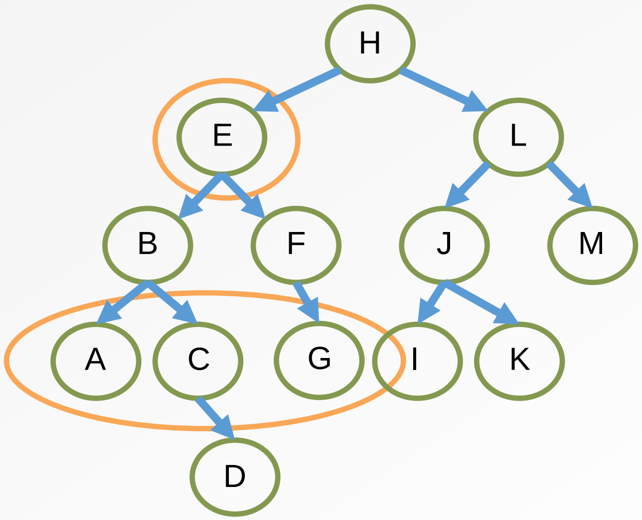
```
int countNode(BTNode *cur){
    if (cur == NULL)
        return 0;

    return (countNode(cur->left)
            + countNode(cur->right)
            + 1);
}
```

OUTLINE

- Tree traversal order
  - Pre-order
  - In-order
  - Post-order
- Application examples
  - Count nodes in a binary tree
  - **Find grandchild nodes**
  - Calculate height of every node
- Level-by-level traversal
- Preorder traversal with a stack

FIND GRANDCHILDREN

- Given a node X, find all the nodes that are X's grandchildren
- Given node E, we should return grandchild nodes A, C, and G
- What if we want to find k-level grandchildren?
  - **Need a way to keep track of how many levels down we've gone**

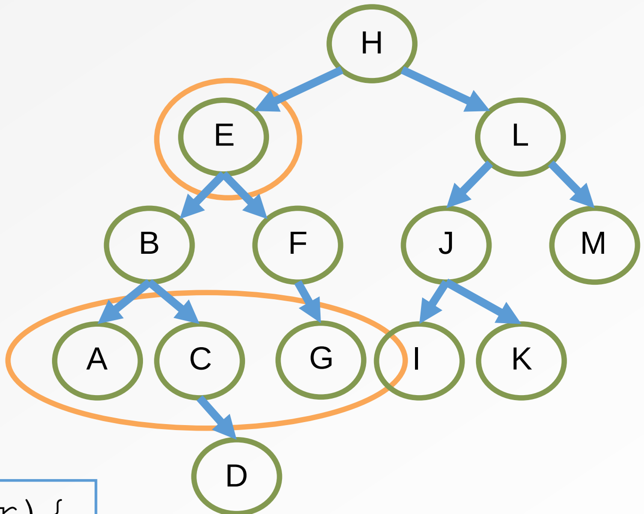


X->left->left
X->left->right
X->right->left
X->right->right

2-level grandchildren

FIND GRANDCHILDREN

- We want to go down k "levels"
- Use a counter to track how far down we've gone
- At each TreeTraversal(child), increment counter




```
void TreeTraversal(BTNode *cur){

    if (cur == NULL)
        return;

    // check counter

    TreeTraversal(cur->left);
    TreeTraversal(cur->right);
}
```

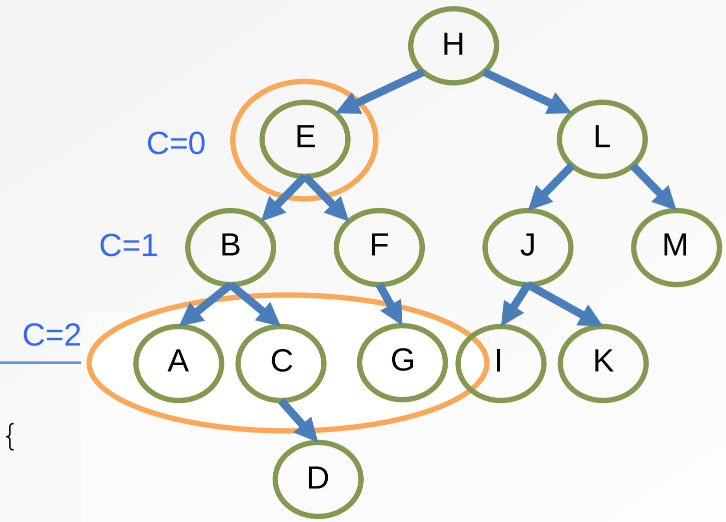
Do something with the current node's data

Visit the left child node

Visit the right child node

FIND GRANDCHILDREN

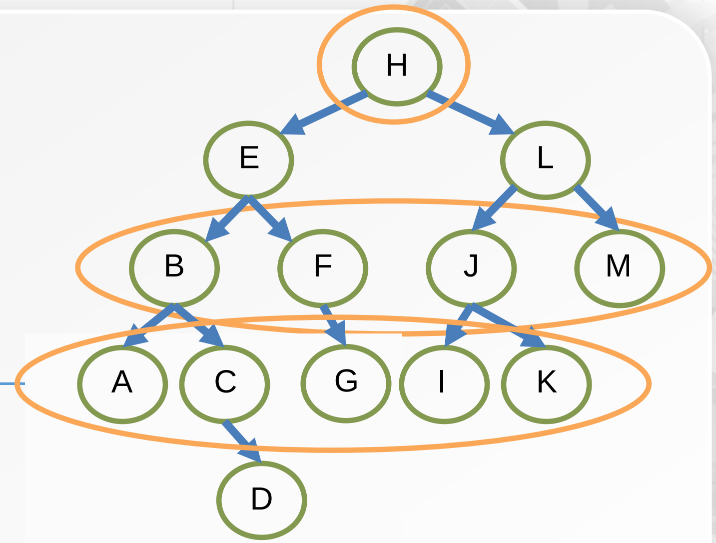
```
void main( ){ …
    if (X = null)return;
    findgrandchildren(X,0);
}
```




```
void findgrandchildren(
             BTNode *cur, int c){
    if (cur == NULL) return;

    if (c == k){
       printf("%d ", cur->item);
       return;
    }
    if (c < k){
       findgrandchildren(cur->left, c+1);
       findgrandchildren(cur->right, c+1); }
}
```

FIND GRANDCHILDREN

```
void main( ){ …
    if (X = null)return;
    findgrandchildren(X,0);
}
```




```
void findgrandchildren(
             BTNode *cur, int c){

    if (cur == NULL) return;

    if (c == k){
       printf("%d ", cur->item);
       return;
    }
    if (c < k){
       findgrandchildren(cur->left, c+1);
       findgrandchildren(cur->right, c+1); }
}
```

if k=2, we call

findgrandchildren(H,0),

what is the output?

How about k=3?

How about

findgrandchildren(H,1)?

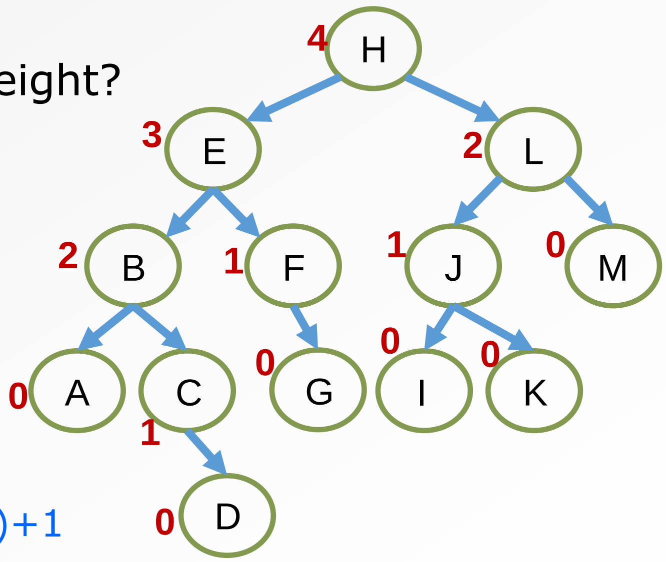
OUTLINE

- Tree traversal order
  - Pre-order
  - In-order
  - Post-order
- Application examples
  - Count nodes in a binary tree
  - Find grandchild nodes
  - **Calculate height of every node**
- Level-by-level traversal
- Preorder traversal with a stack

CALCULATE HEIGHT OF EVERY NODE

- Height of a node = number of links from that node to the deepest leaf node
- How does each node calculate its height?
  - What is the height of node D, C, H?
- We found:
  - leaf.height= 0
  - Non-leaf node X

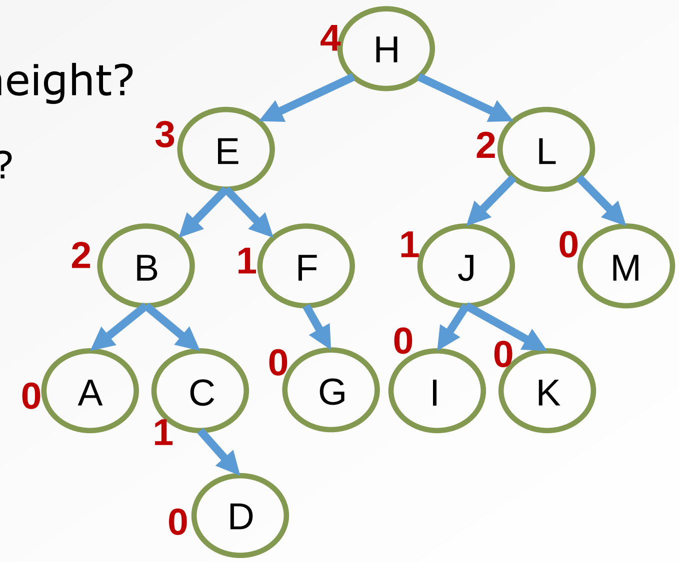
X.height=max(X.left.height, X.right.height)+1



- Does information propagate upwards or downwards?

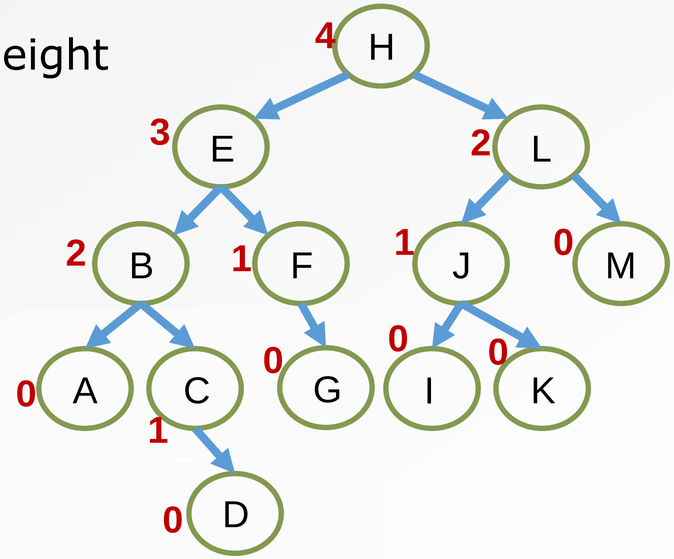
CALCULATE HEIGHT OF EVERY NODE

- Height of a node = number of links from that node to the deepest leaf node
- How does each node calculate its height?
  - What is the height of node D, C, H?
- Go through entire tree: calculate and store height of each node in the item field



CALCULATE HEIGHT OF EVERY NODE

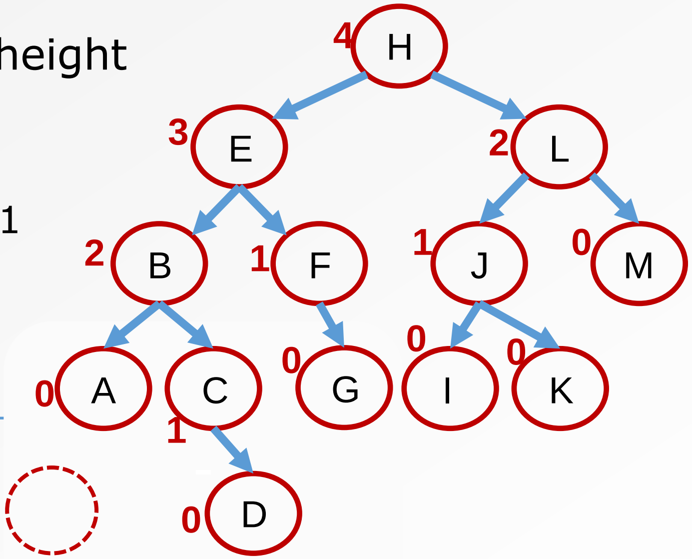
- We want each node to report its height
  - Leaf node must report 0




```
int TreeTraversal(BTNode *cur){

    if(cur == NULL)
        return   ;

    int l = TreeTraversal(cur->left);
    int r = TreeTraversal(cur->right);

    // do something here. Max( left, right)?

    return  ;

}
```

CALCULATE HEIGHT OF EVERY NODE

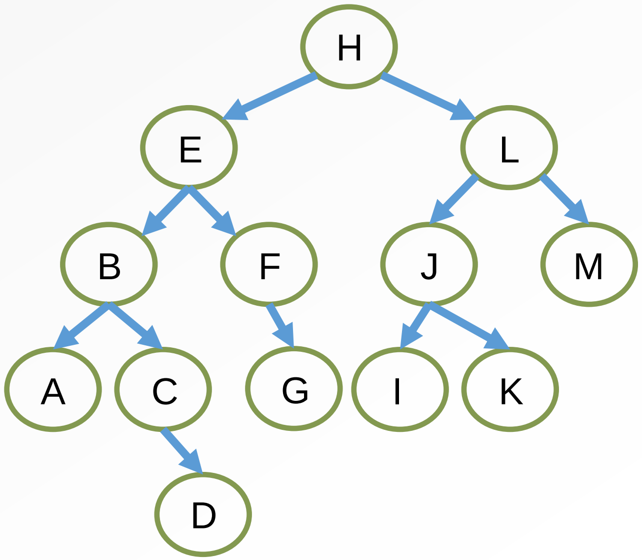
- We want each node to report its height
  - Leaf node must report 0
  - At "null" condition, must report -1




```
int TreeTraversal(BTNode *cur){

    if(cur == NULL)
        return -1;

    int l = TreeTraversal(cur->left);
    int r = TreeTraversal(cur->right);

    int c = max (l, r) + 1;

    return c;

}
```

QUESTIONS

- Does the tree traversal order matter?
- Depth of a node = number of links from that node to the root node. How does each node calculate its depth?

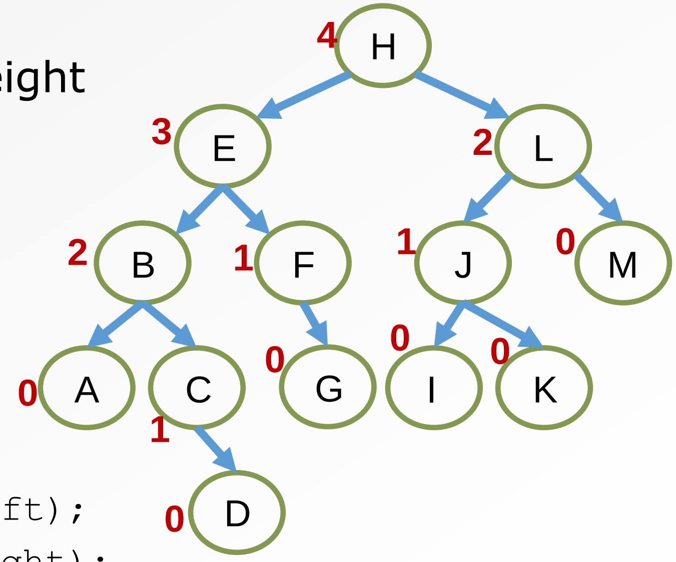


CALCULATE HEIGHT OF EVERY NODE

- Height of a node = number of links from that node to the deepest leaf node

- We want each node to report its height
  - Leaf node must report 0
  - At "null" condition, must report -1

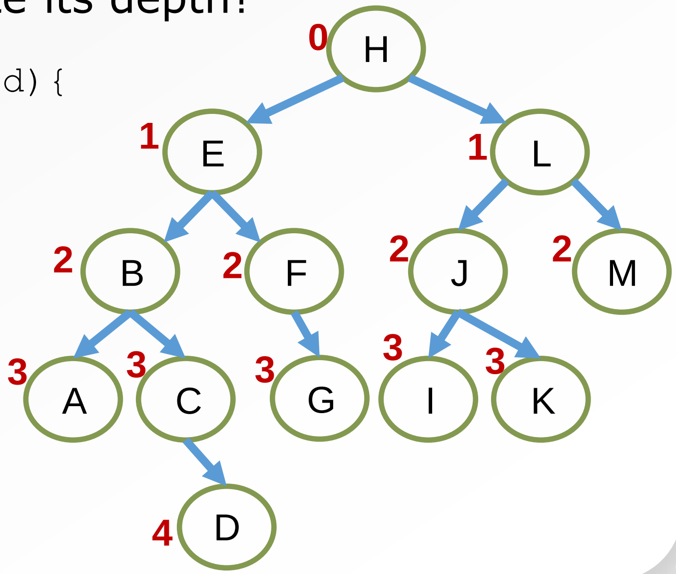
```
int TreeTraversal(BTNode *cur){
    if(cur == NULL)
        return -1;

    int l = TreeTraversal(cur->left);
    int r = TreeTraversal(cur->right);

    int c = max (l, r) + 1;

    return c;
}
```



QUESTIONS

- Does the tree traversal order matter?
- Height of a node = number of links from that node to the deepest leaf node
- Depth of a node = number of links from that node to the root node. How does each node calculate its depth?

```
void TreeTraversal(BTNode *cur, int d){
    if(cur == NULL)
        return;
    //print cur->item and d;
    TreeTraversal(cur->left, d+1);
    TreeTraversal(cur->right, d+1);
    return;
}
```



OUTLINE

- Tree traversal order
  - Pre-order
  - In-order
  - Post-order
- Application examples
  - Count nodes in a binary tree
  - Find grandchild nodes
  - Calculate height of every node
- **Level-by-level traversal**
- Preorder traversal with a stack

LEVEL-BY-LEVEL: BREADTH-FIRST SEARCH

Depth-first search

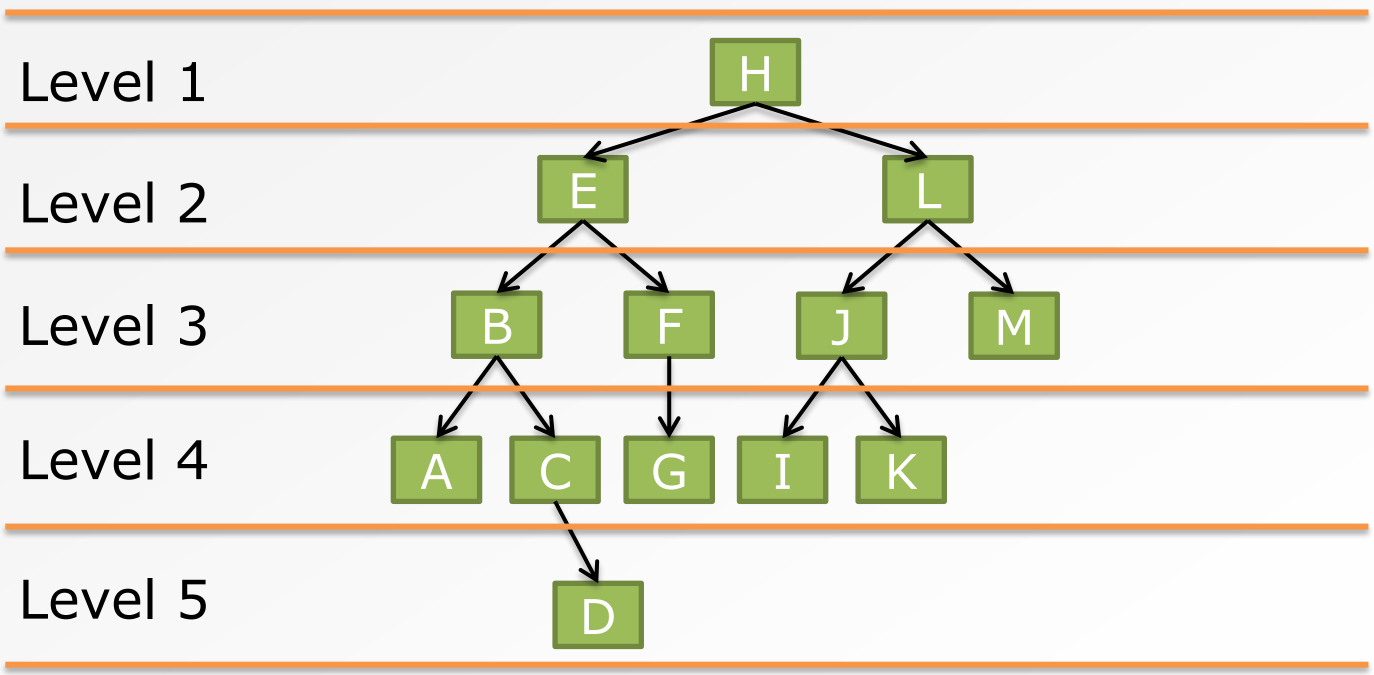
begins at the root and explores as far as possible along each branch
before backtracking

E.g. the post-order traversal

Breadth-first search

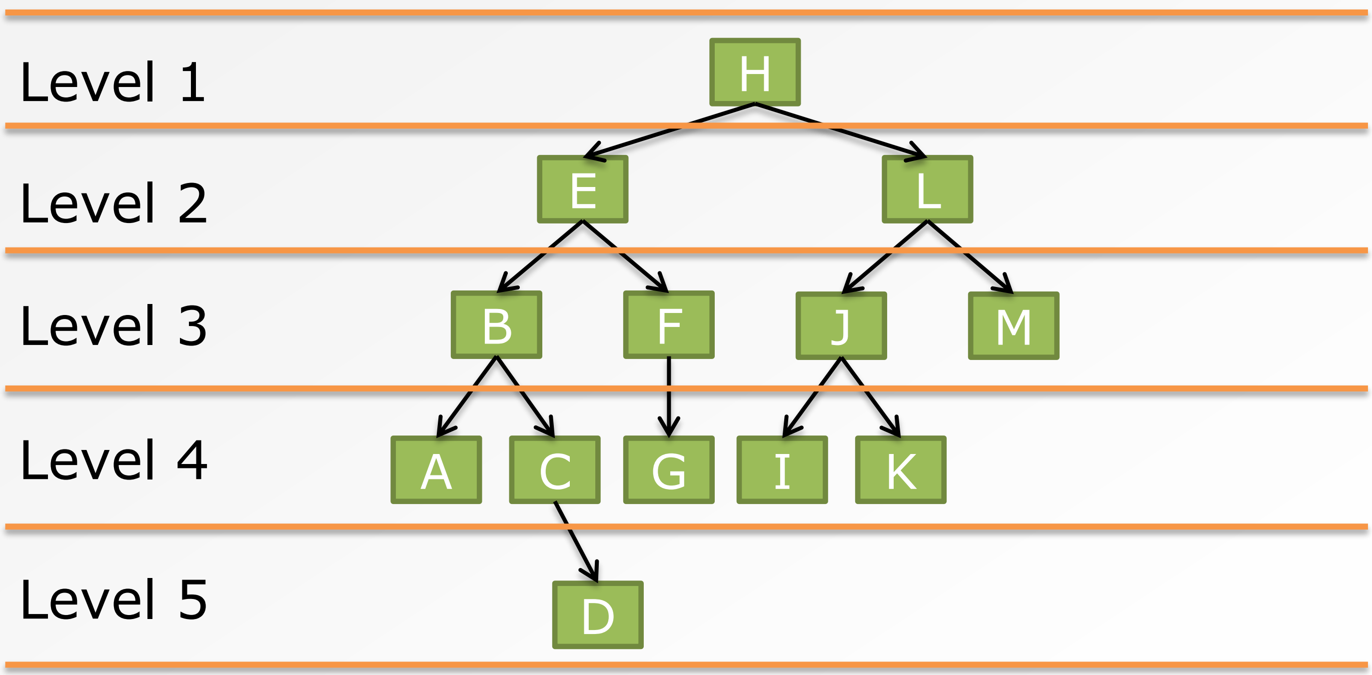
begins at a root node and inspects all its children nodes. Then for each of those children nodes in turn, it inspects their children nodes, and so on.

LEVEL-BY-LEVEL TREE TRAVERSAL



LEVEL-BY-LEVEL TREE TRAVERSAL

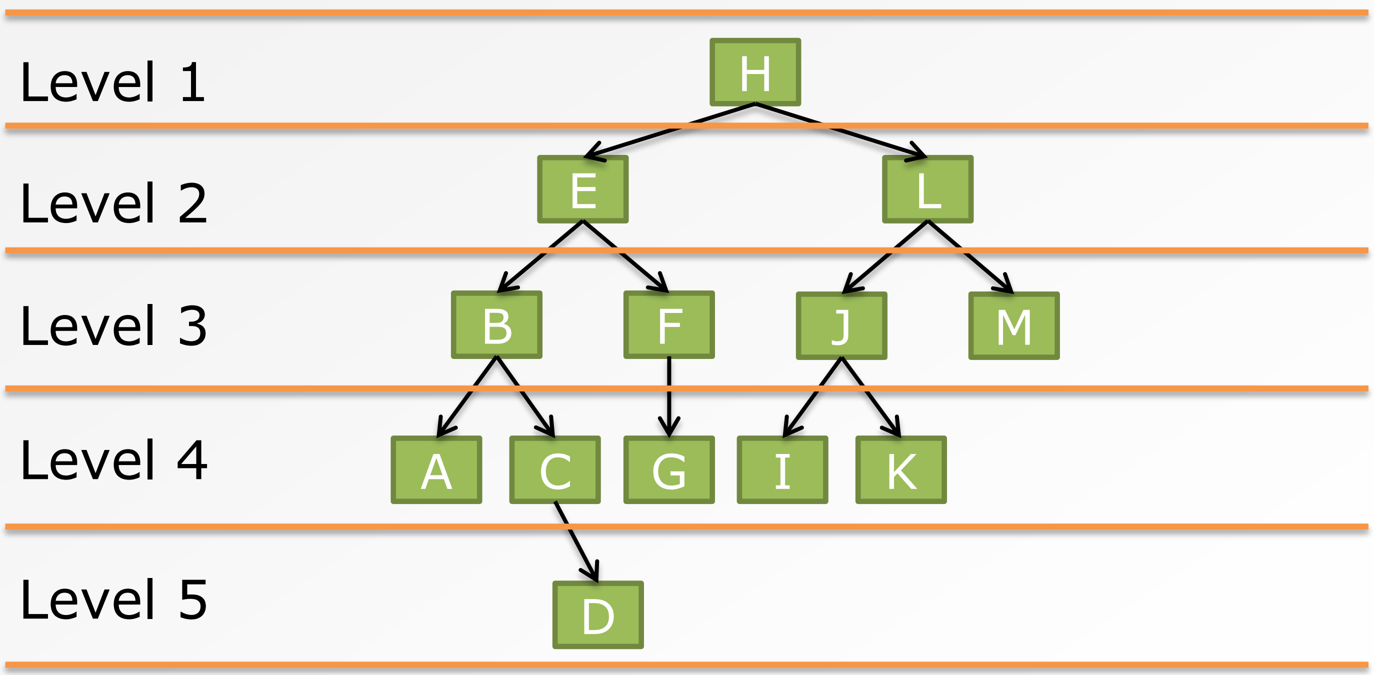
- Hint: Make use of another data structure



Nodes stored in order accessed in tree…

LEVEL-BY-LEVEL TREE TRAVERSAL

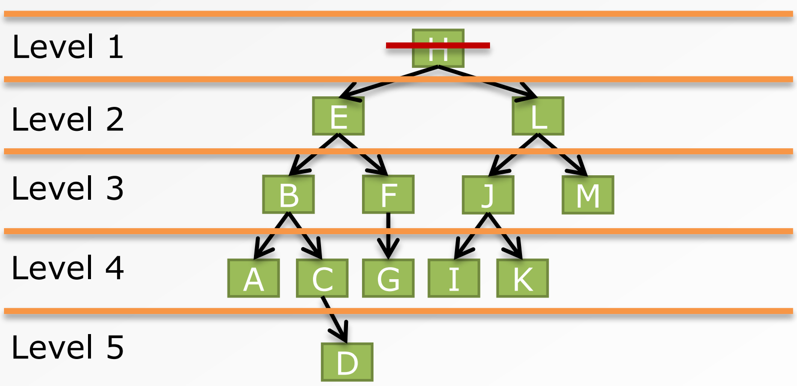
- Use a queue! Root node should be first



Nodes stored in order accessed in tree

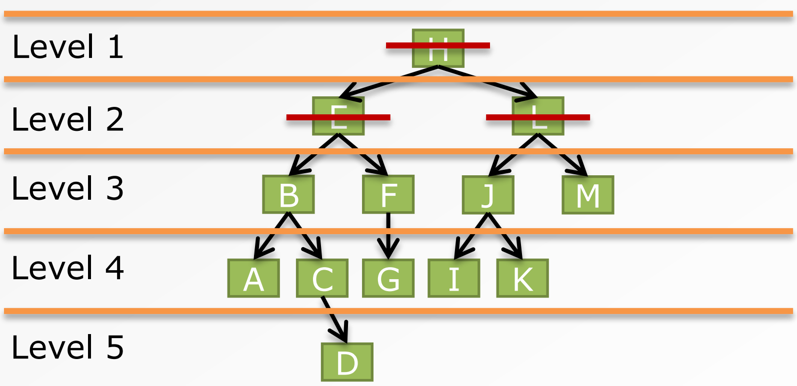
LEVEL-BY-LEVEL TREE TRAVERSAL

- Enqueue the root, H

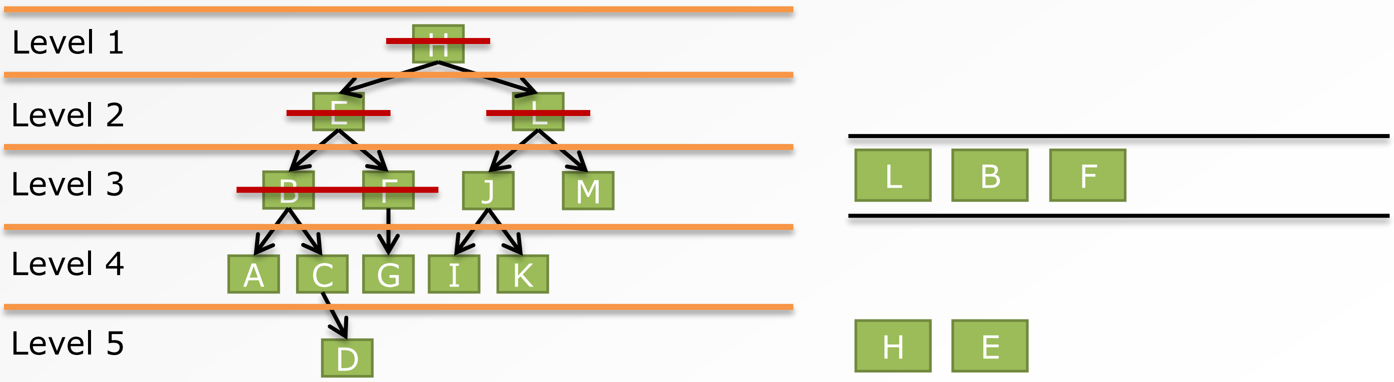


LEVEL-BY-LEVEL TREE TRAVERSAL

- Enqueue the root, H
- Dequeue H, and enqueue H's children

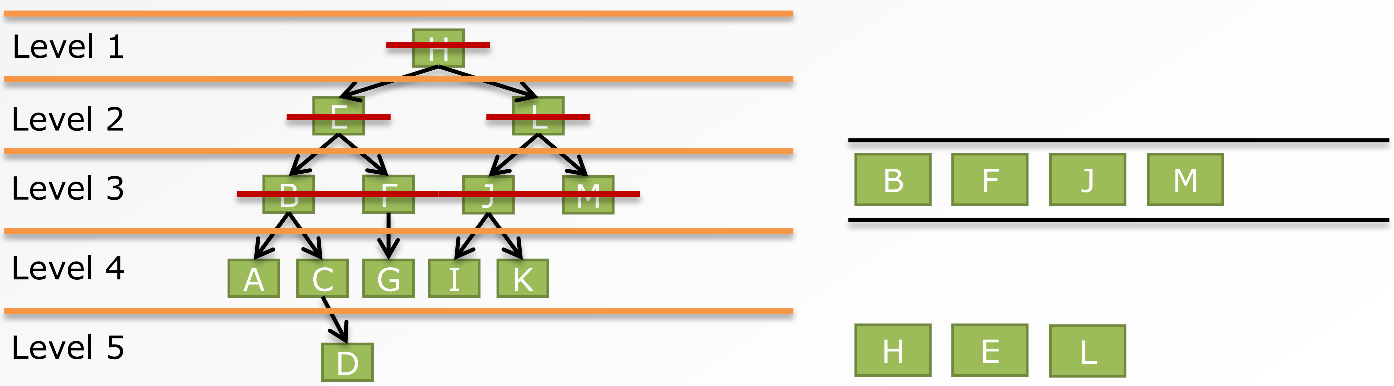


- Enqueue the root, H
- Dequeue H, and enqueue H’s children
- Dequeue E, and enqueue E’s children



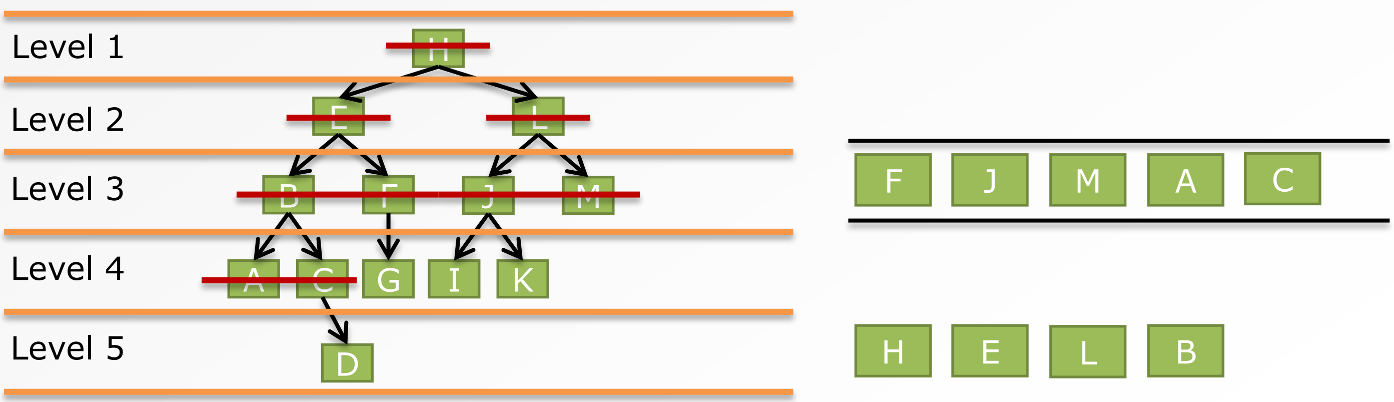
LEVEL-BY-LEVEL TREE TRAVERSAL

- Enqueue the root, H
- Dequeue H, and enqueue H's children
- Dequeue E, and enqueue E's children
- Dequeue L, and enqueue L's children



LEVEL-BY-LEVEL TREE TRAVERSAL

- Enqueue the root, H
- Dequeue H, and enqueue H's children
- Dequeue E, and enqueue E's children
- Dequeue L, and enqueue L's children
- Dequeue B, and enqueue B's children



OUTLINE

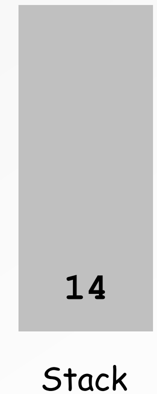
- Tree traversal order
  - Pre-order
  - In-order
  - Post-order
- Application examples
  - Count nodes in a binary tree
  - Find grandchild nodes
  - Calculate height of every node
- Level-by-level traversal
- **Preorder traversal with a stack**

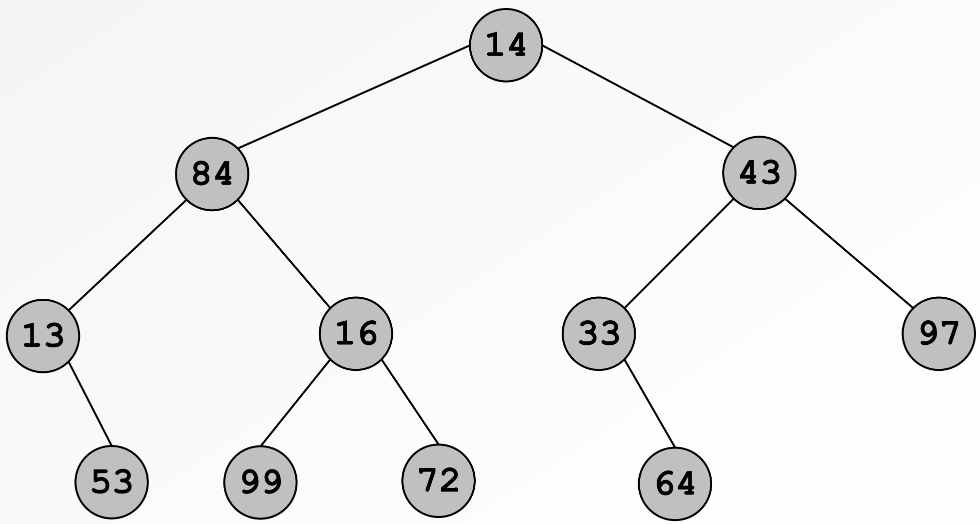
PREORDER TRAVERSAL WITH A STACK

Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children





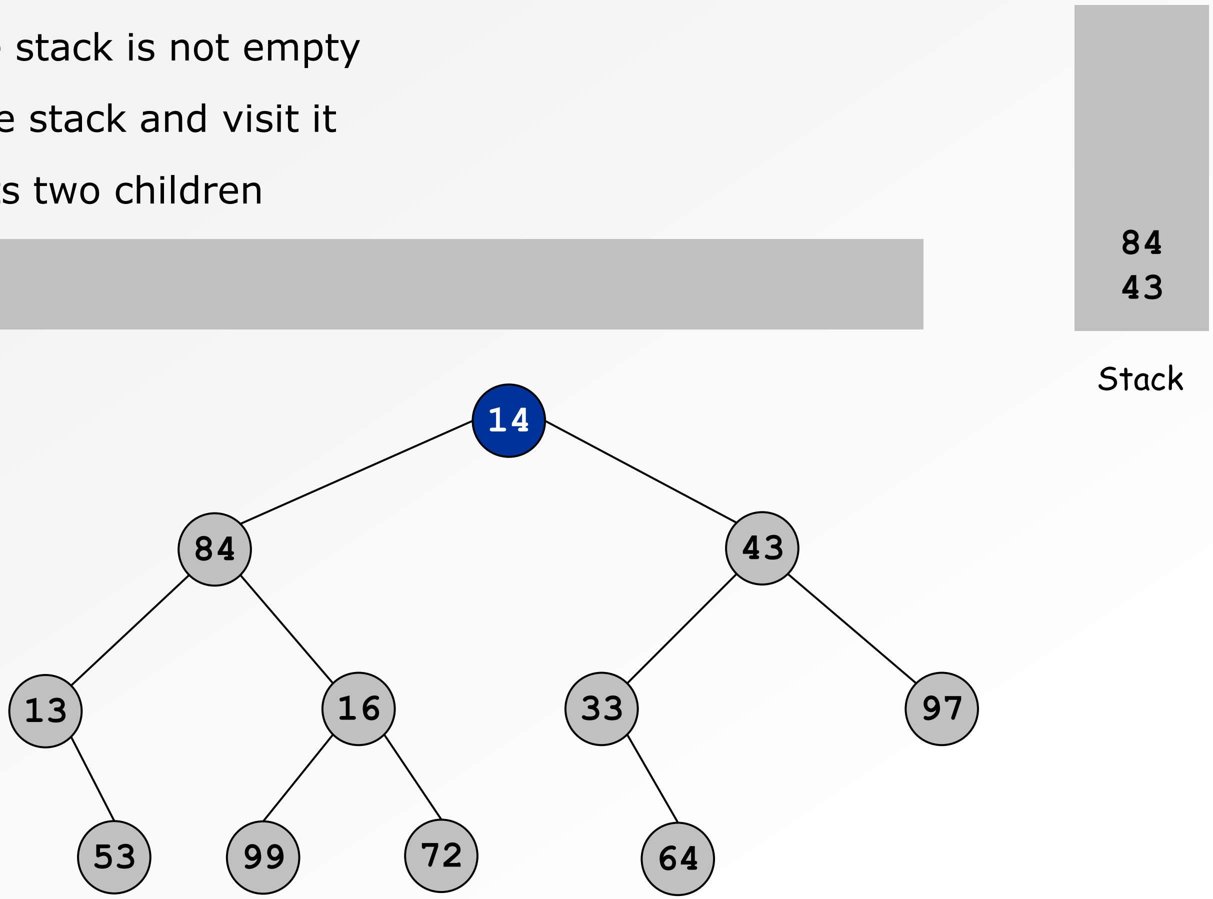
PREORDER TRAVERSAL WITH A STACK

Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

14



PREORDER TRAVERSAL WITH A STACK

Push the root onto the stack.

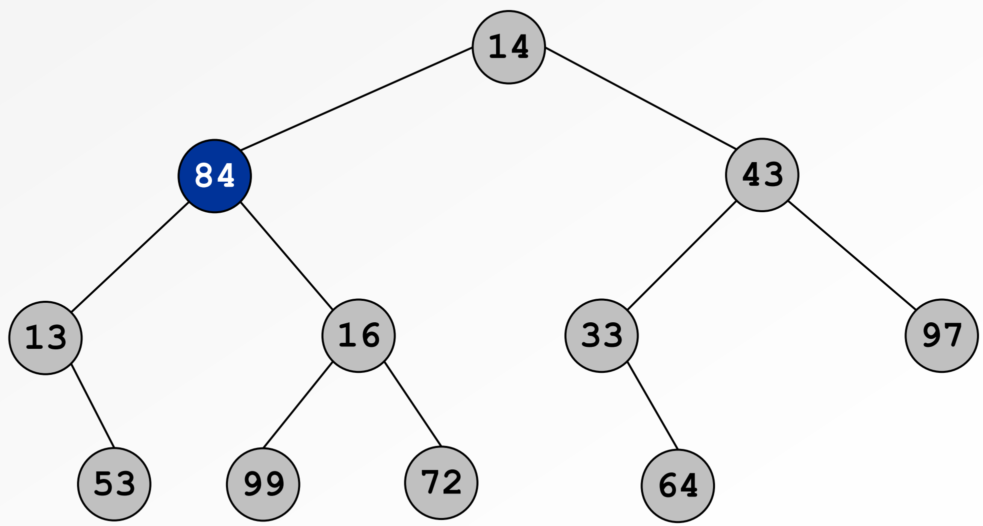
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84

13
16
43

Stack



PREORDER TRAVERSAL WITH A STACK

Push the root onto the stack.

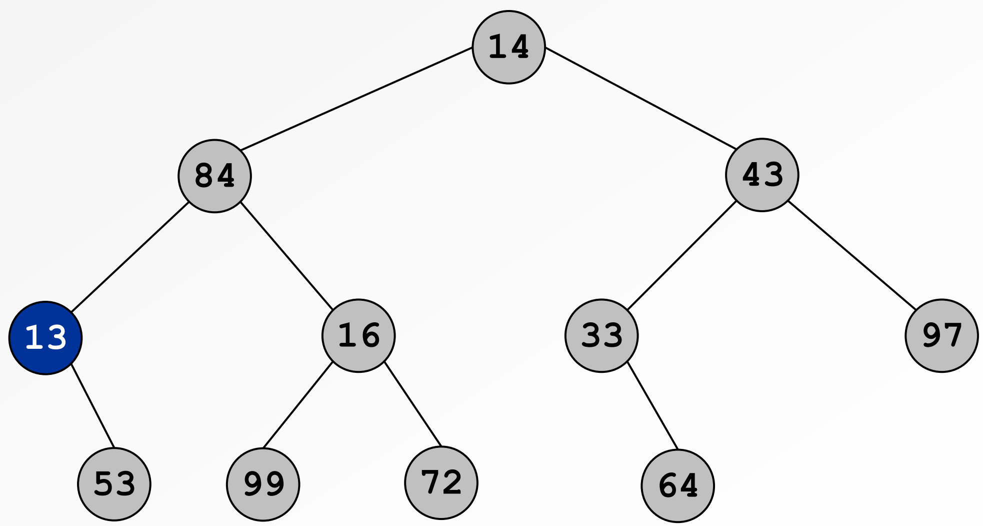
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13

53
16
43

Stack



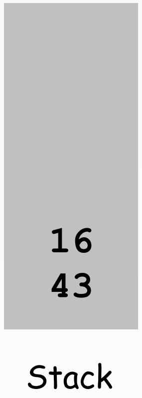
PREORDER TRAVERSAL WITH A STACK

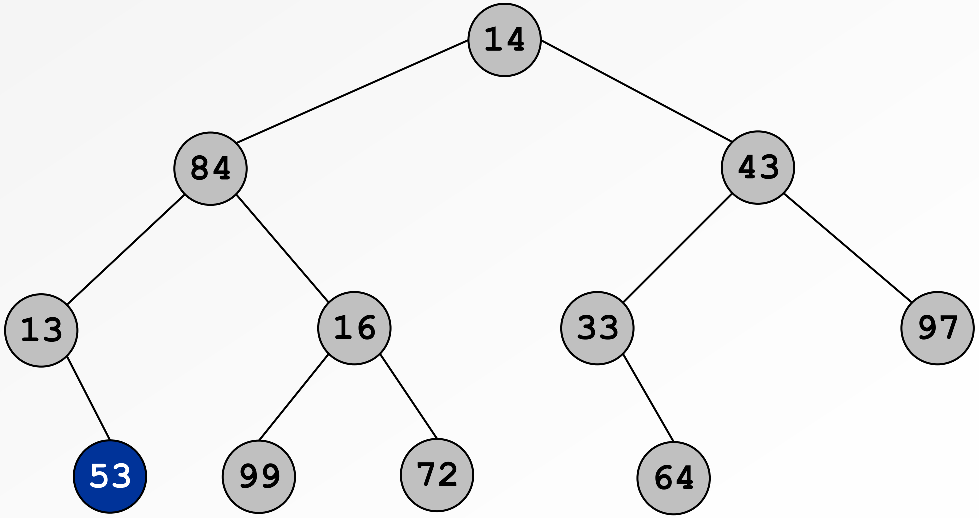
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

`14 84 13 53`





PREORDER TRAVERSAL WITH A STACK

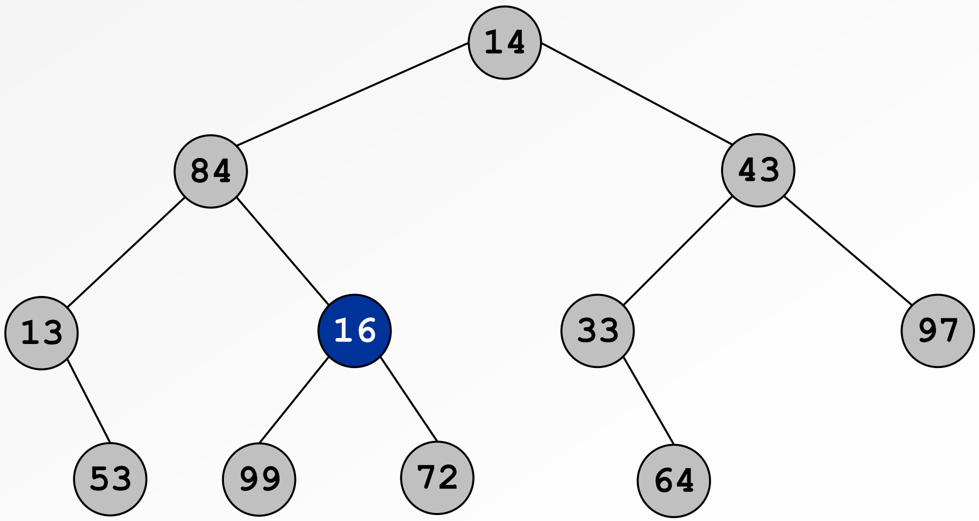
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16

Stack
99
72
43



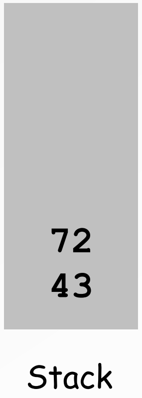
PREORDER TRAVERSAL WITH A STACK

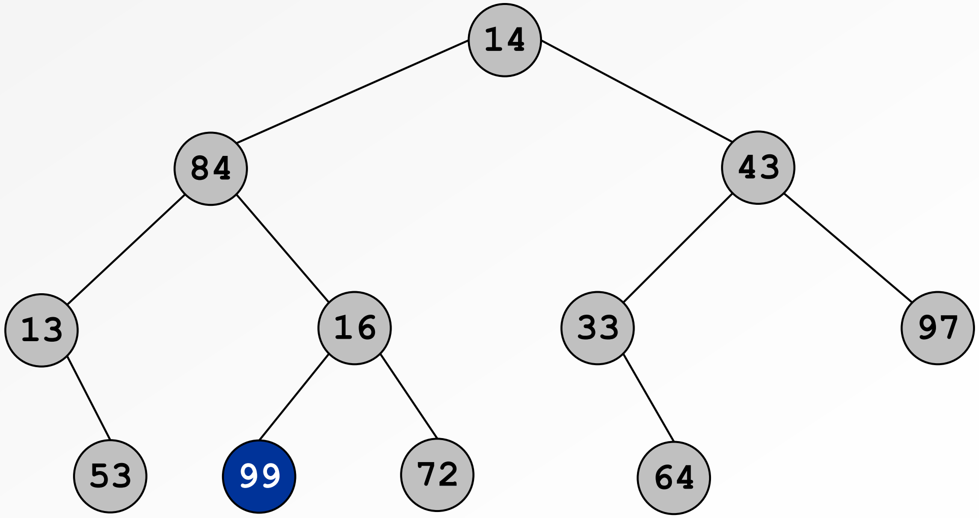
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

```
14 84 13 53 16 99
```



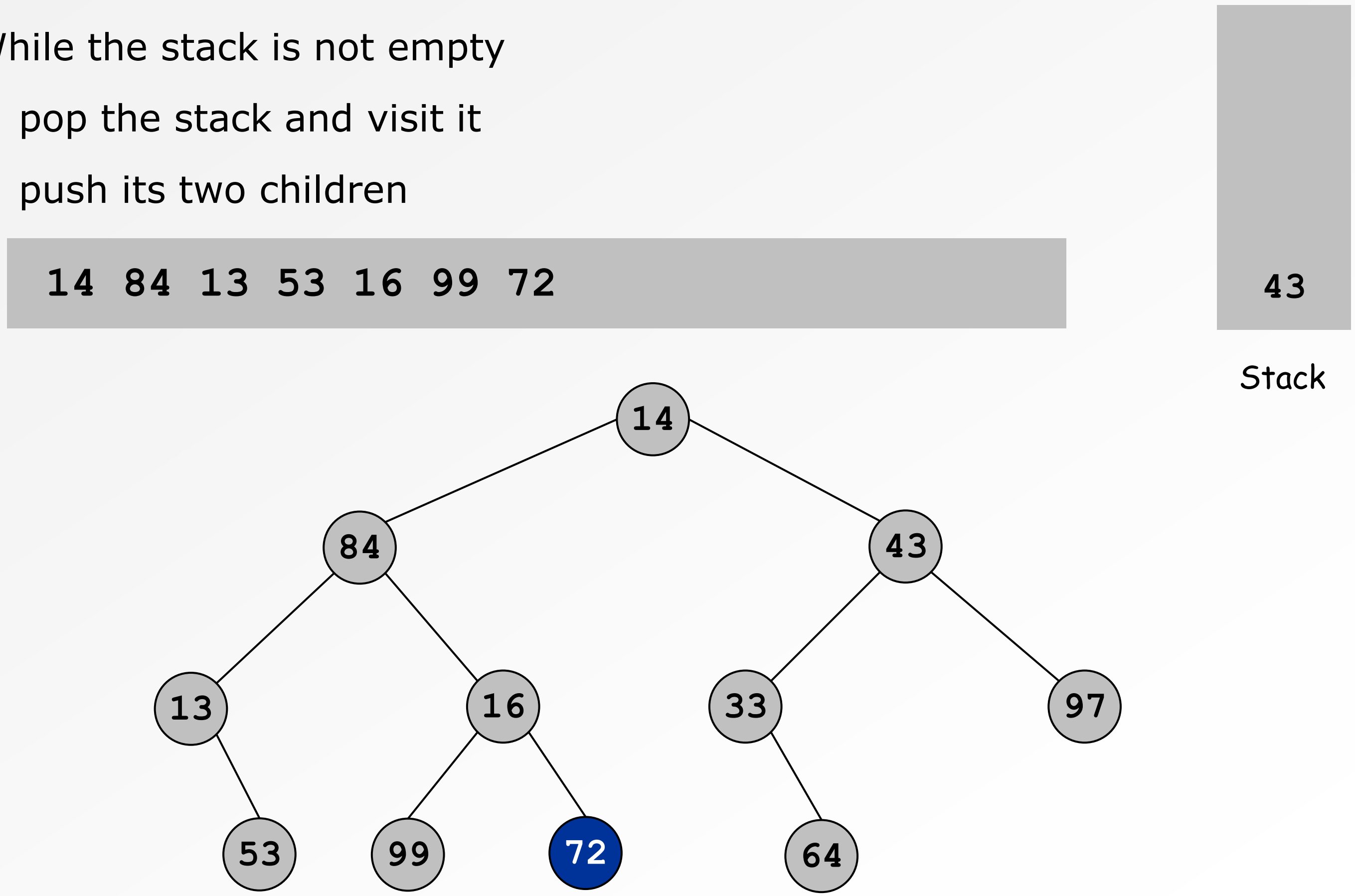


PREORDER TRAVERSAL WITH A STACK

Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children



PREORDER TRAVERSAL WITH A STACK

Push the root onto the stack.

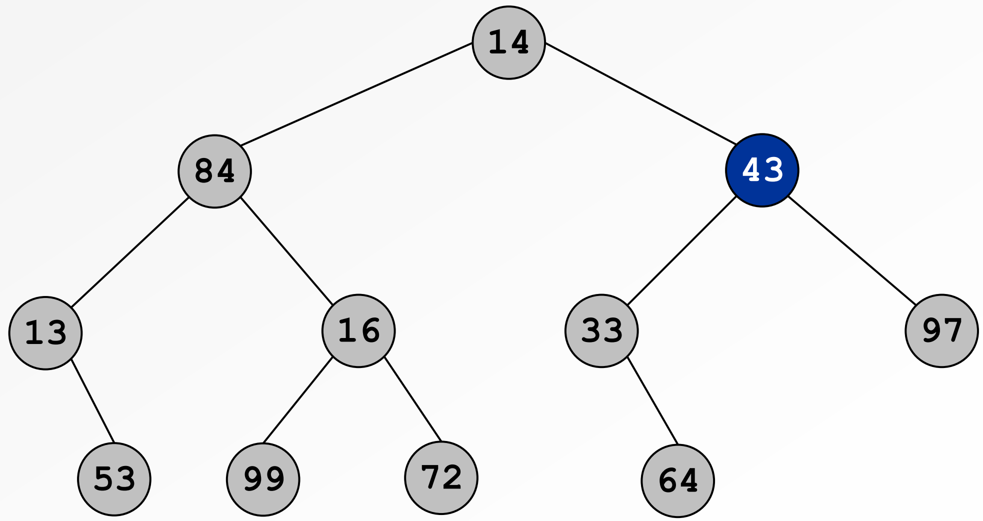
While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99 72 43

33
97

Stack



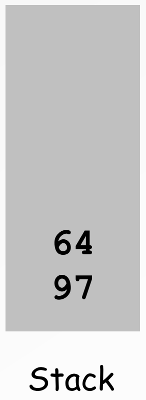
PREORDER TRAVERSAL WITH A STACK

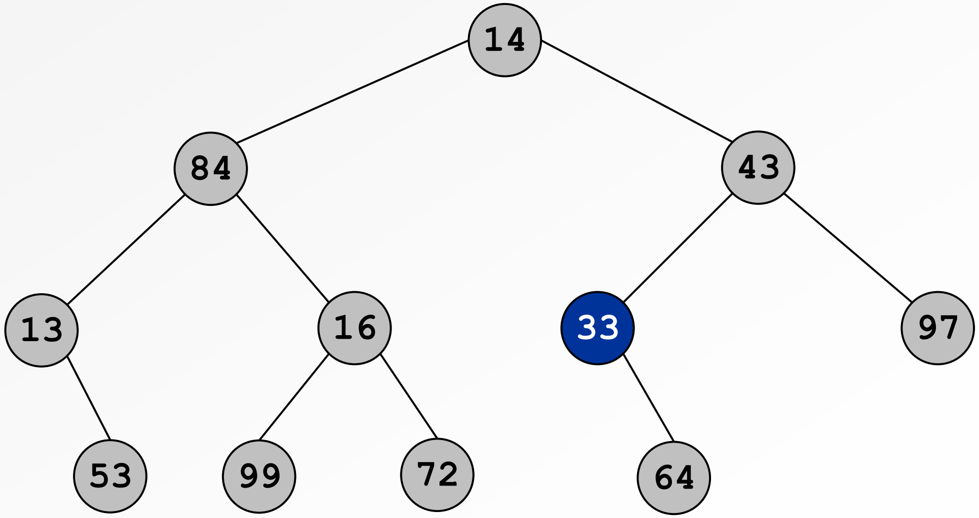
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99 72 43 33





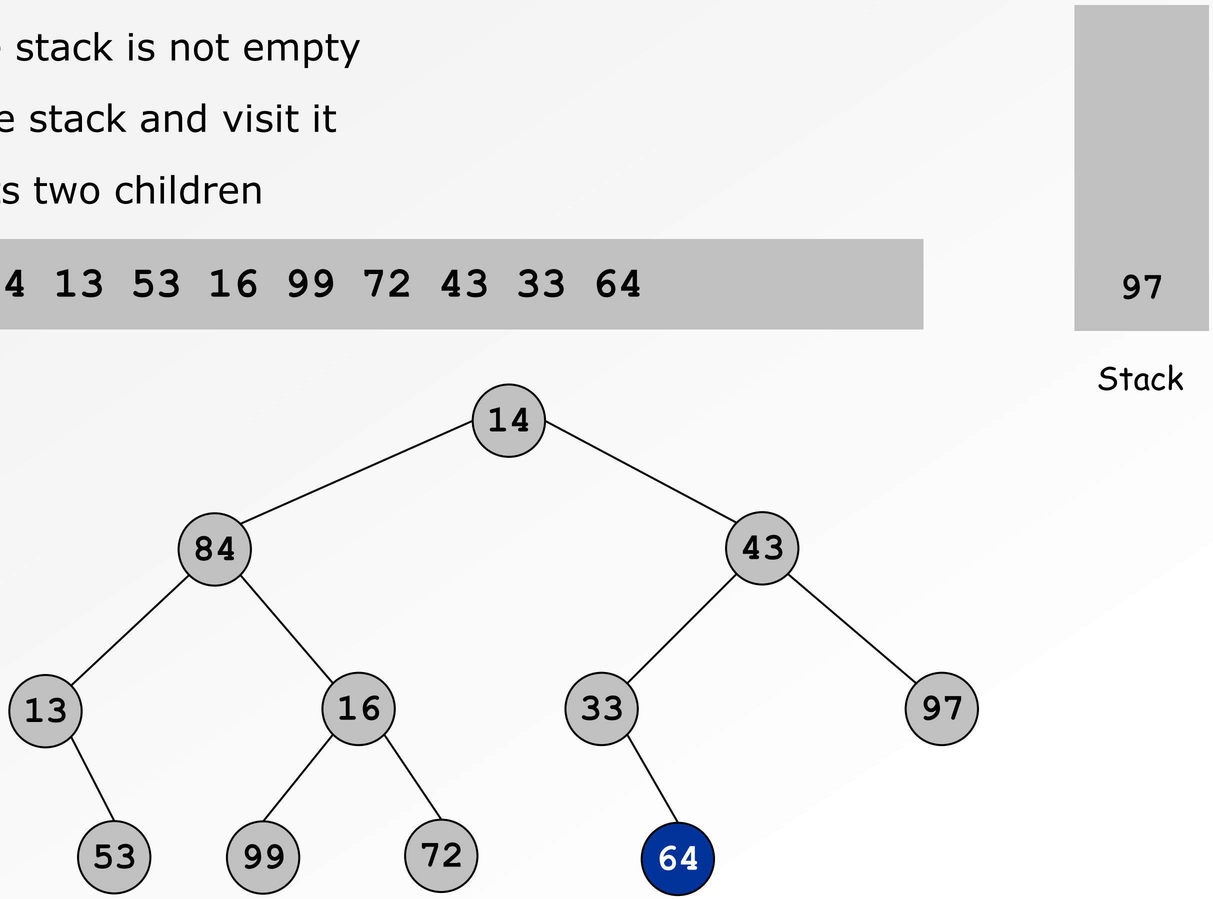
PREORDER TRAVERSAL WITH A STACK

Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99 72 43 33 64



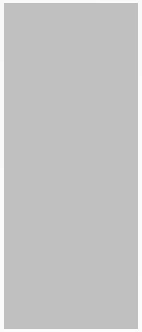
PREORDER TRAVERSAL WITH A STACK

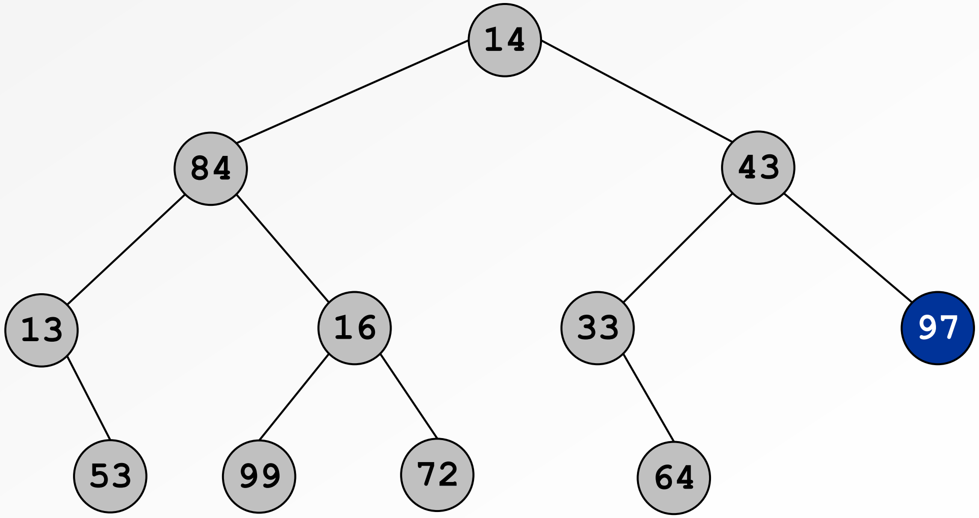
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

`14 84 13 53 16 99 72 43 33 64 97`





PREORDER TRAVERSAL WITH A STACK

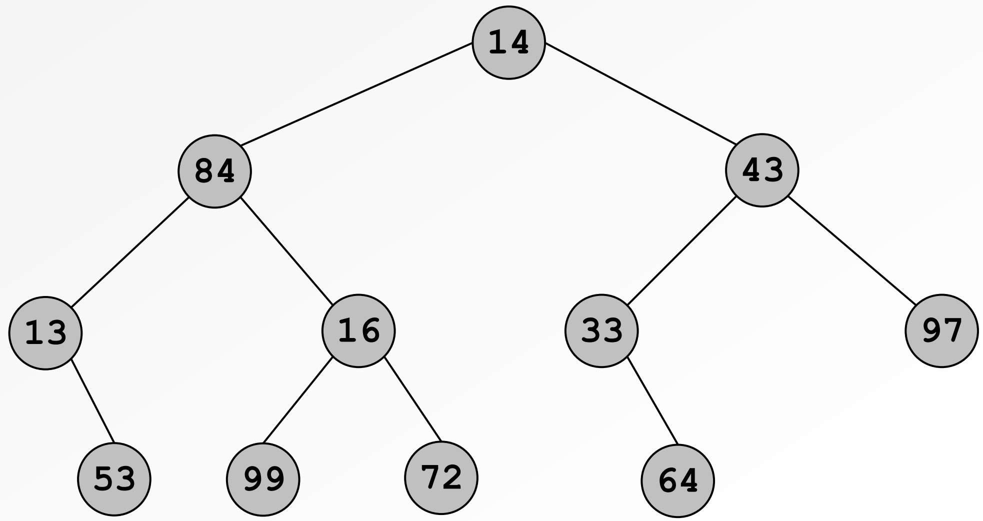
Push the root onto the stack.

While the stack is not empty

- pop the stack and visit it
- push its two children

14 84 13 53 16 99 72 43 33 64 97

Stack



YOU SHOULD BE ABLE TO

- Binary tree Traverse:
  - Pre-order
  - In-order
  - Post-order
- Write recursive binary tree functions using the TreeTraversal template as a starting point
- Based on the traversal of the binary tree, do a lot of things: print, count numbers, count height/depth, find grandchildren,…, etc.