Trees
(part 2 – Data Structure)

CSE 3318 – Algorithms and Data Structures
University of Texas at Arlington

Student Self-Review

- Review the theoretical lecture on trees that was covered earlier in the semester.
- Review your notes and materials on implementing trees in C.

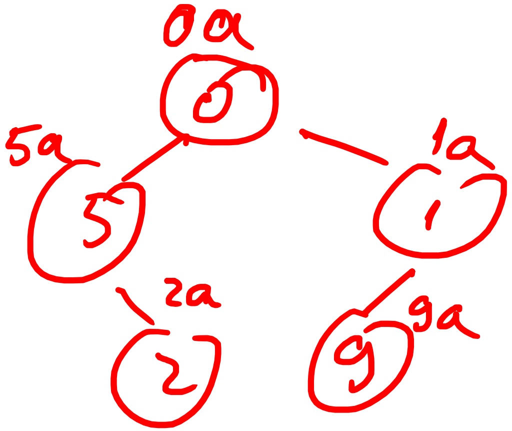
leetcode

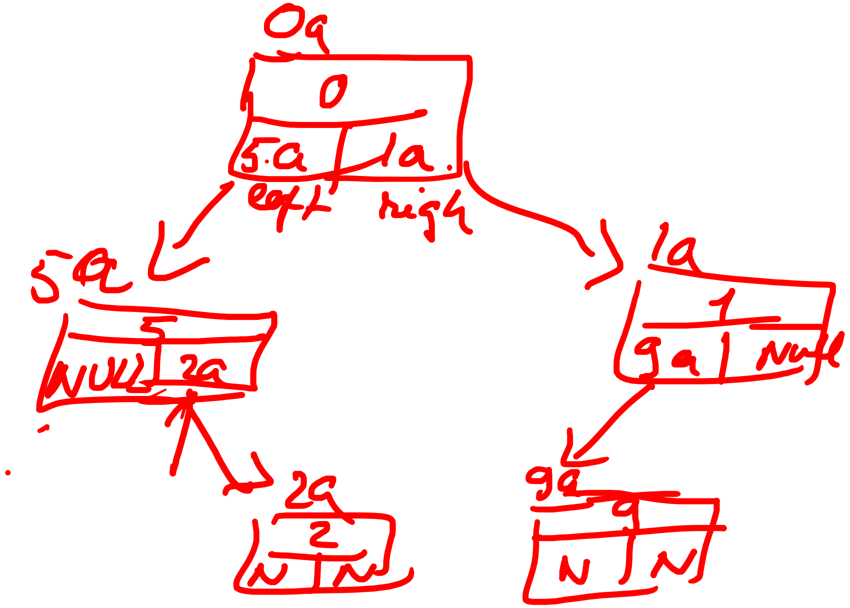
- 993. Cousins in Binary Tree (ok)
- 257. Binary Tree Paths (challenging)
- 94. Binary Tree Inorder Traversal
- 98. Validate Binary Search Tree
- Many more…

Main concepts:

- Traversals
  - Depth first: postorder, preorder, inorder (recursive)
  - Breadth first: level order (iterative), see it used in leetcode to describe trees
- Other functions: count nodes, compute height
- Analyzing time complexity for recursive functions on binary trees.
  - Uses: Tree of recursive function calls (TRC), local time complexity (TCL) and full tree and its property
- TRC - Tree of Recursive [function] Calls
  - This is a FULL tree (using the internal nodes-leaves property of full trees allows us to compute the time complexity)
  - Shown on the tree and also
  - Shown with local cost written in the nodes – we will see more of this later on
- Full tree
  - Definition: Every node has exactly 0 or 2 children (there is NO node with just 1 child)
  - Property: leaves=1+internal_nodes=> total_nodes=1+2internal_nodes (regardless of shape)
- TCL – local time complexity
  - Time complexity/work done in a function call, EXCLUDING the [work in] recursive calls
- TRC and TCL are acronyms that I created

```c
typedef struct TreeNode * TreeNodePT;
struct TreeNode {
  int data;
  TreeNodePT left;
  TreeNodePT right;
};
```





Traversing a Binary Tree

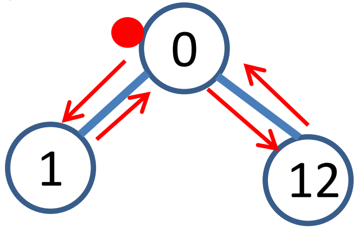
- **Traversal** - go through each node and do something with it. E.g.:
  - print it
  - change it
  - use its content to compute something
- Standard traversals

(depth-first order)

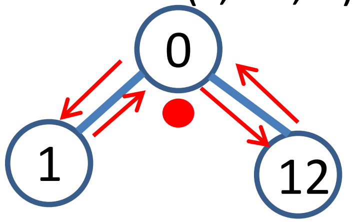
  - **Preorder (Root, Left, Right)**: visit the node, then its left subtree, then its right subtree.
  - **Inorder (Left, Root, Right)**: visit the left subtree, then the node, then the right subtree.
  - **Postorder (Left, Right, Root)**: visit the left subtree, then the right subtree, then the node.
  - **Level order**: going from tree level 0 to the last and left to right within level.

    (breadth-first order)

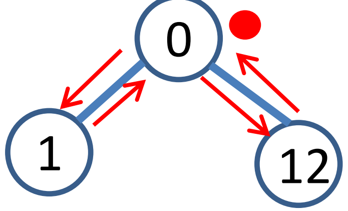
preorder (Ro, L, R)



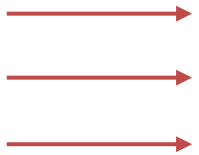
inorder (L, Ro, R)



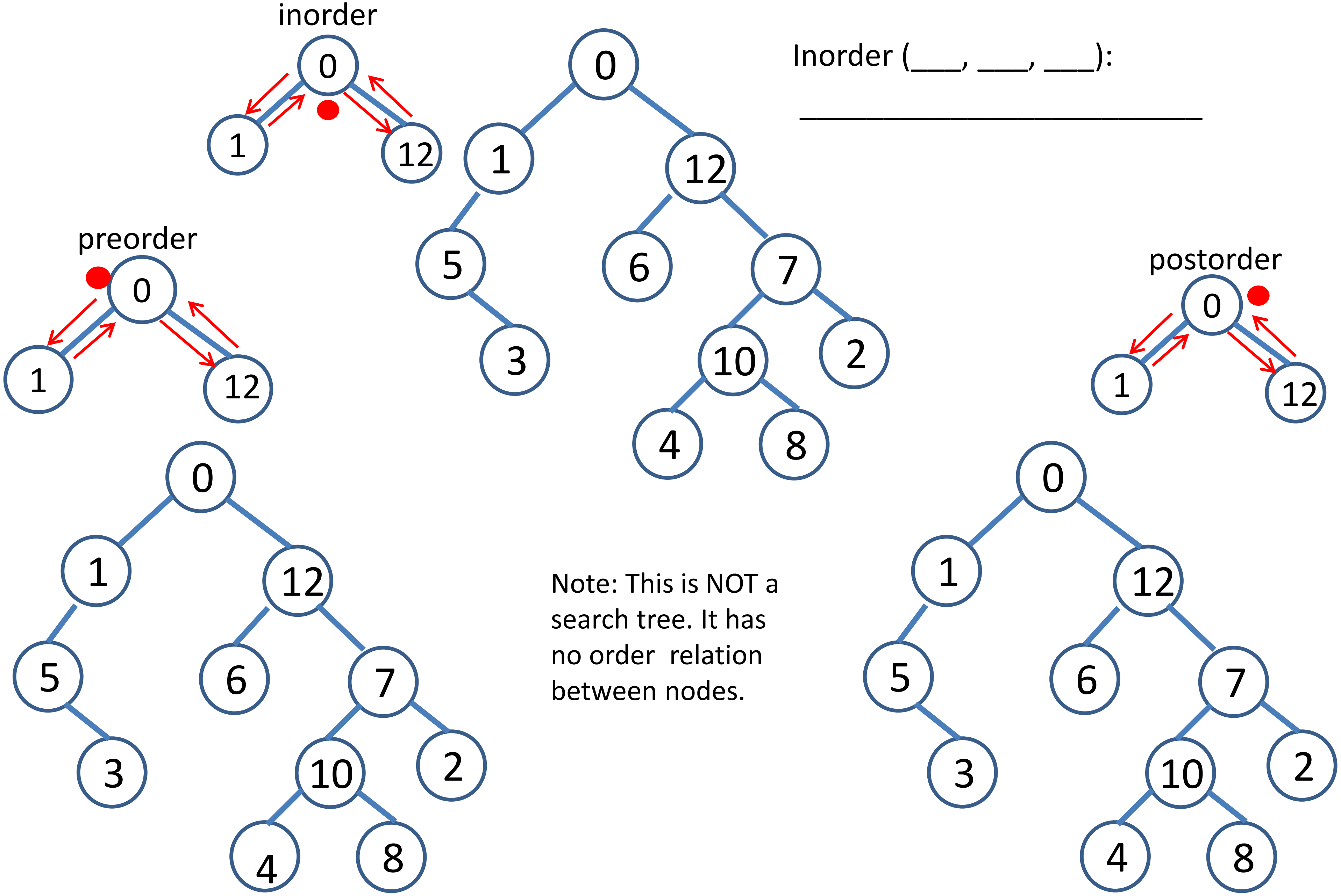
postorder (L, R, Ro)



levelorder



To remember the name for each traversal, think about where/when the root is processed with respect to its children.



Inorder (____, ____, ____):

____

Note: This is NOT a search tree. It has no order relation between nodes.

Preorder (____, ____, ____):

Postorder (____, ____, ____):

____

Recursive Tree Traversal

```
void preorder(TreeNodePT h){
base    if (h == NULL) return;
        do_something(h);printf(h->data)
        preorder (h->left);
        preorder (h->right);
}
```

```
void inorder(TreeNodePT h){
        if (h == NULL) return;
        inorder (h->left);
        do_something(h);
        inorder (h->right);
}
```

```
void post (TreeNodePT h){
        if (h == NULL) return;
        post(h->left);
        post(h->right);
        do_something(h);print(h->data)
}
```

```
typedef struct TreeNode * TreeNodePT;
struct TreeNode {
  int data;
  TreeNodePT left;
  TreeNodePT right;
};
```

Other possible fields:

- `TreeNodePT parent;`
- `int size; //`size of subtree rooted at this node. Useful for balancing trees.
- Anything else you need to add to solve a problem

For a tree with N nodes:

Time complexity: O(N)

Space complexity:

Recursive Tree Traversal - Answer

```
void preorder(TreeNodePT h){
    if (h == NULL) return;
    do_something(h);
    preorder (h->left);
    preorder (h->right);
}
```

```
void inorder(TreeNodePT h){
    if (h == NULL) return;
    inorder (h->left);
    do_something(h);
    inorder (h->right);
}
```

```
void post (TreeNodePT h){
    if (h == NULL) return;
    post(h->left);
    post(h->right);
    do_something(h);
}
```

```
typedef struct TreeNode * TreeNodePT;
struct TreeNode {
  int data;
  TreeNodePT left;
  TreeNodePT right;
};
```

Other possible fields:

- `TreeNodePT parent;`
- `int size; //`size of subtree rooted at this node. Useful for balancing trees.
- Anything else you need to add to solve a problem

For a tree with N nodes:

Time complexity: **Θ(N)** (assume do_something(h) is Θ(1))
See derivation on next page

Space complexity: **Θ(treeHeight) (O(N))** (b.c. recursion)
(if we exclude the recursion frame stack: Θ(1))

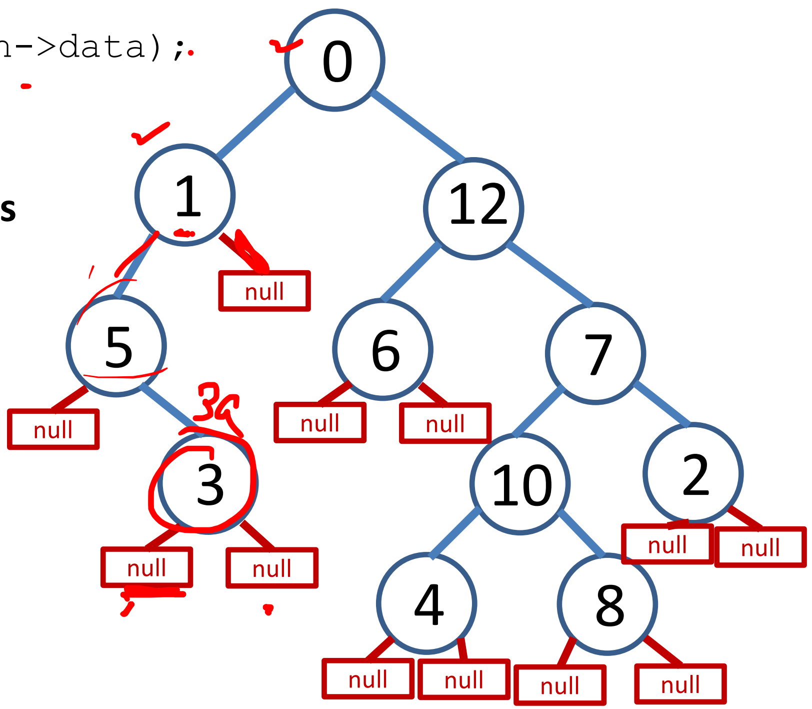
Postorder traverasal – code tracing

```
void post(TreeNodePT h){
  if (h == NULL) return;
  ~~post(h->left);~~
  ~~post(h->right);~~
  printf("%d, ", h->data);
}
```

O(1)

Let N be the number of nodes of the tree to be traversed.

E.g. here N = 11



full

post(0) [0aaa]

--- post(1)

------ post(5)

--------- post(null) return

--------- post (3))

------------ post (null)

------------ post (null)

------------ 3

--------- 5

------ post (null)

------1

---post (12)

.....

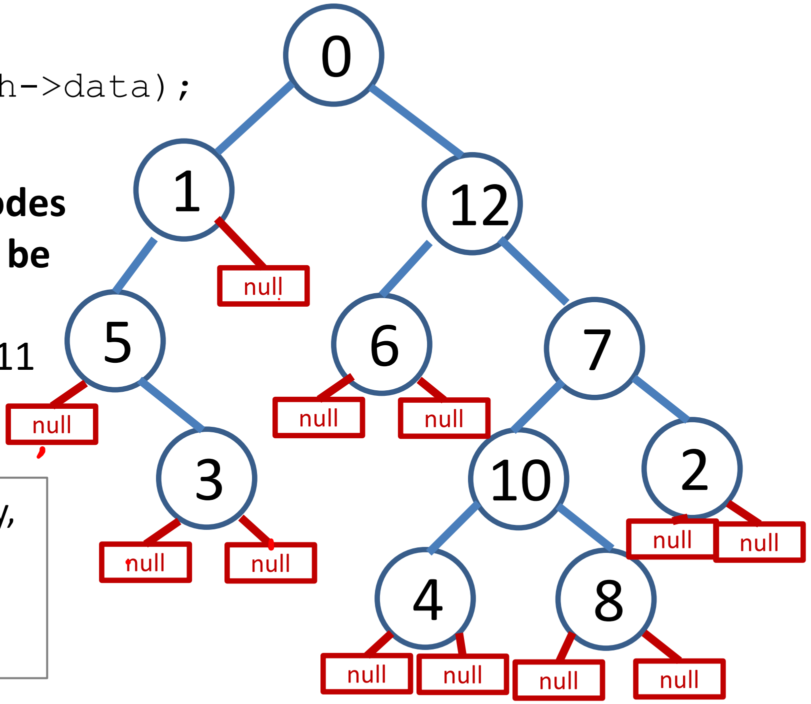
```
struct TreeNode {
  int data;
  TreeNodePT left;
  TreeNodePT right;
};
```

Postorder traverasal –
Time complexity derivation using the tree of recursive calls (TRC)

```
void post(TreeNodePT h){ // postorder
traversal
  if (h == NULL) return;
  post(h->left);
  post(h->right);
  printf("%d, ", h->data);
}
```

Let N be the number of nodes of the tree to be traversed.
E.g. here N = 11

TCL, local time complexity, is given by what instructions here?
TCL() = Θ(1)



post(0) [0aa]
--- post(1) [1aa]
------ post(5) [5aa]
--------- post(null) return
--------- post (3) [3aa]
------------ post (null)
------------ post (null)
------------ 3 printed
--------- 5 printed
------ post (null)
------1 printed
---post (12)
.....

The TREE of RECURSIVE CALLS (TRC) is a FULL tree where

- internal nodes of TRC correspond to fct calls on the original tree nodes (including leaves of the original tree) – (blue circles in picture) => N nodes
- leaves of TRC correspond to the calls for null, (red rectangles in picture)

Full tree property: leaves = internal nodes+1 => post(null) calls = 1+post(node) calls =1+N =>
=> Total recursive calls =post(node)+post(null) = N+(1+N), and each recursive call has TCL()=Θ(1) => The time complexity to traverse the entire tree (TC (root)) is total_rec_calls*TCL() = (2N+1)* Θ(1) = Θ(N)
Where TCL is the LOCAL time complexity/work done in one function call, EXCLUDING the recursive calls

Postorder traverasal –
TREE of RECURSIVE CALLS showing the LOCAL cost, c, in each node

```
void post(TreeNodePT h){ // postorder traversal
   if (h == NULL) return;
   post(h->left);
   post(h->right);
   printf("%d, ", h->data);
}
```

$(2N+1)O(1)$

$O(N)$

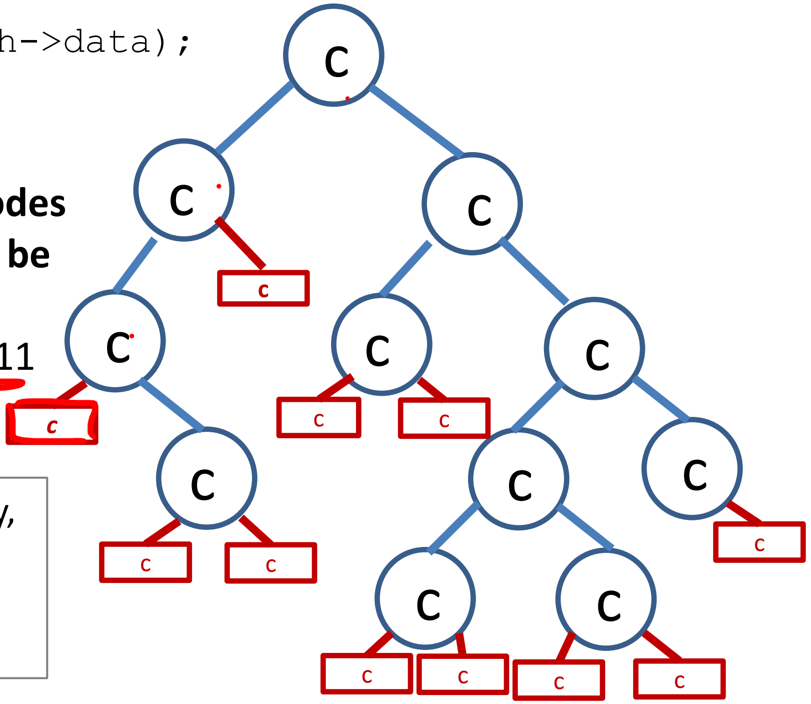
Let N be the number of nodes of the tree to be traversed.

E.g. here N = 11

x internal → x+1 leaves

TCL, local time complexity, is given by what instructions here?

TCL() = Θ(1) = c



TCL(null) and TCL(non_null_node) are both constant, but not necessarily the same actual constant. (NOTE that this is TCL, LOCAL TC, not the entire TC for that function call.)

HOWEVER we will ***use the same constant, c***, for the local cost for recursive cases (blue nodes) and base case (red nodes. (Imagive you pick the larger of the two actual local costs.)

The TREE of RECURSIVE CALLS (TRC) is a FULL tree where

- internal nodes of TRC correspond to fct calls on the original tree nodes (including leaves of the original tree) – (blue circles in picture) => N nodes
- leaves of TRC correspond to the calls for null, (red rectangles in picture)

Full tree property: leaves = internal nodes+1 => post(null) calls = 1+post(node) calls =1+N =>

=> Total recursive calls =post(node)+post(null) = N+(1+N), and each recursive call has **TCL()=Θ(1) =c =>** The time complexity to traverse the entire tree (TC (root)) is total_rec_calls*TC$_{local}$() **= (2N+1)* c = 2cN+c= Θ(N)**

Where TCL is the LOCAL time complexity/work done in one function call, EXCLUDING the recursive calls

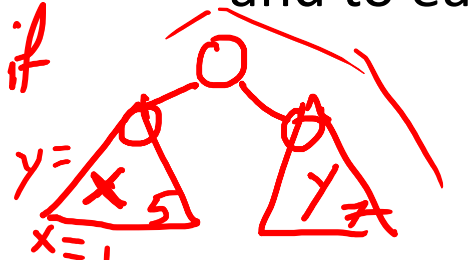
void count2(TreeNodePT t, int * cf)

Class Practice

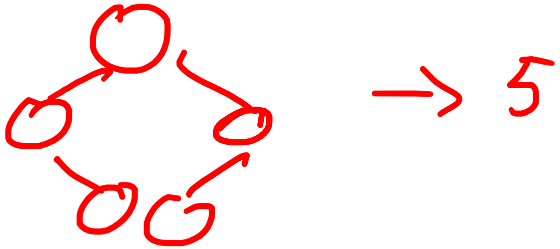
- Write the following (recursive or not) functions, in class:
  - Count the number of nodes in a tree
  - Compute the height of a tree
    - height of a leaf is 0 (node with no child)
  - Level-order traversal – discuss/implement
  - Print the tree in a tree-like shape – discuss/implement

```
int count (TreeNodePT tree){
if (tree == NULL) return 0;
int L = count(tree->left);
int R = count(tree->right);
return 1+L+R;
}
```

- Which functions are "similar" to the traversals discussed previously and to each other?



return X+Y+1



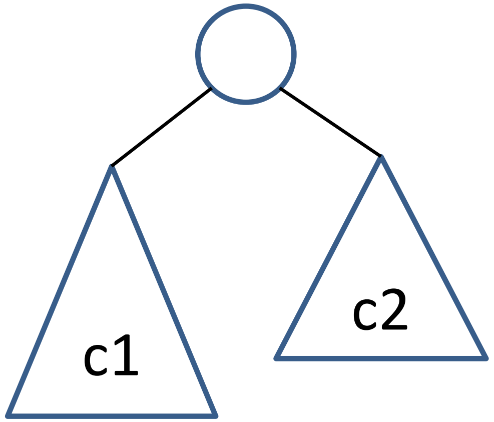
0 → 1

NULL → 0

Recursive Examples

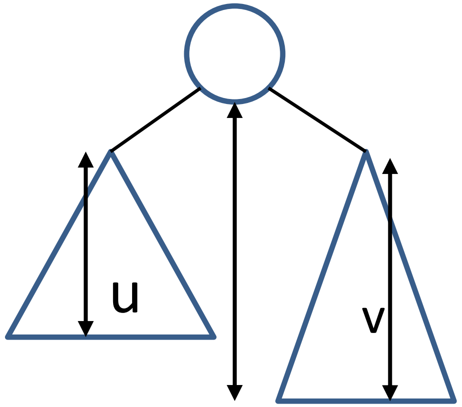
Count the number of nodes in the tree

```
int count(TreeNodePT h){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1;
}
```



Compute the height of the tree

```
int height(TreeNodePT h){
    if (h == NULL) return -1;
    int u = height(h->left);
    int v = height(h->right);
    if (u > v)
          return u+1;
    else
          return v+1;
}
```

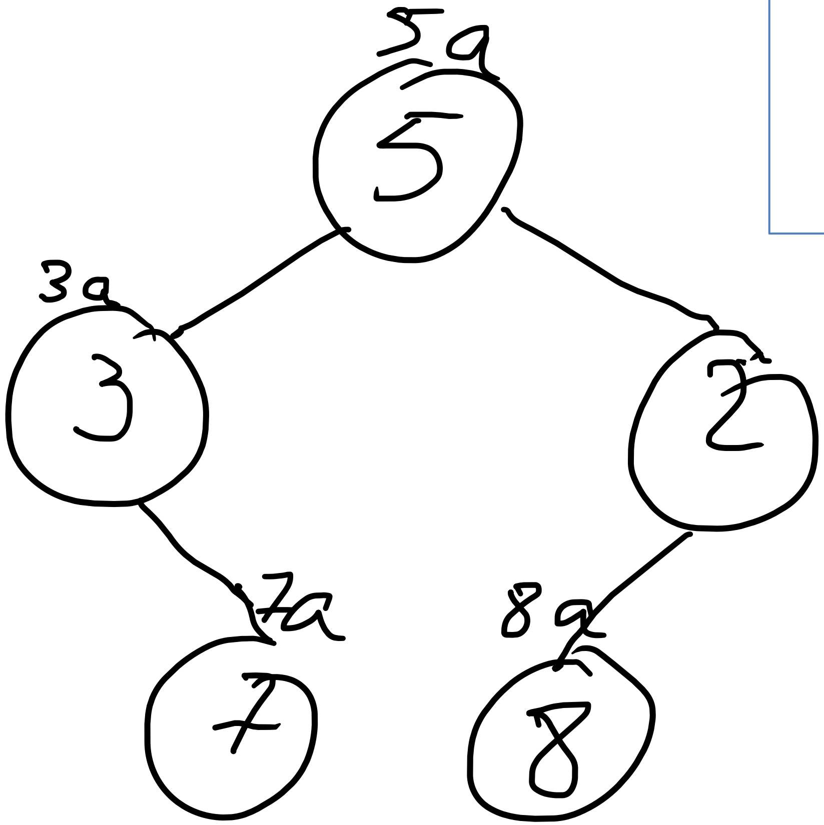


Can you write the `countTwo` used below?

It does not return the count, but modifies the argument `ct`.

```
int ct = 0;
countTwo(root,&ct);
printf("ct=%d",ct);// gives correct count
```

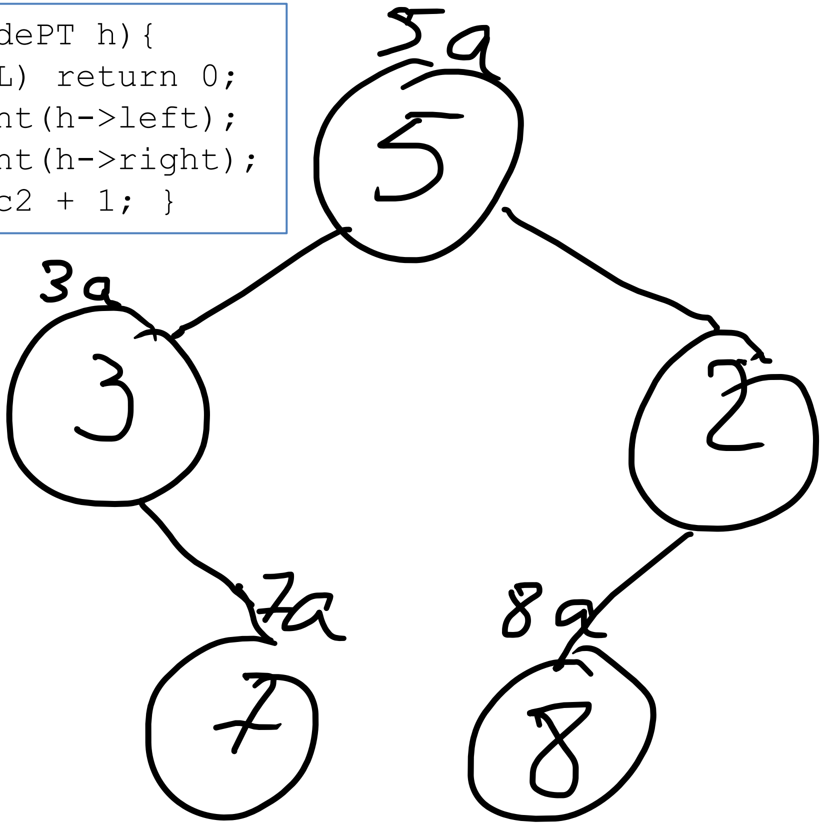
```
int count(TreeNodePT h){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1; }
```



```
int count(TreeNodePT h){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1; }
```

```
int count(TreeNodePT h){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1; }
```

```
int count(TreeNodePT h){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1; }
```



```
int count(TreeNodePT h){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1; }
```

```
int count(______){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1; }
```

```
int count(______){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1; }
```

```
int count(______){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1; }
```

```
int count(______){
    if (h == NULL) return 0;
    int c1 = count(h->left);
    int c2 = count(h->right);
    return c1 + c2 + 1; }
```

Recursive Examples: print tree

Prints the contents of each node

How will the output look like?

What type of tree traversal is this?

Note how the pass-by-value works for the depth variable: the correct depth is passed for each node, and even after returning from the recursive call(s), it remains correct for the current node. E.g. after the call show(x->left, depth+1) depth is still the depth of this node. We want pass-by-value, NOT pass-by-reference for depth.

```
typedef struct node * nodePT;
struct TreeNode {
  int data;
  nodePT left;
  nodePT right;
};
```

```
void print_node(int val, int depth) {
    int i;
    for (i = 0; i < depth ; i++)
        printf("  ");
    printf("%d\n", val);
}

void show(TreeNodePT x, int depth) {
    if (x == NULL) {
        printnode(-1, depth);
        return;
    }
    print_node(x->data, depth);
    show(x->left, depth+1);
    show(x->right, depth+1);
}
```

Use leetcode to run my code: cousins in binary tree

Leetcode problem: 257. Binary Tree Paths

Return **all paths from the root to leaves.**

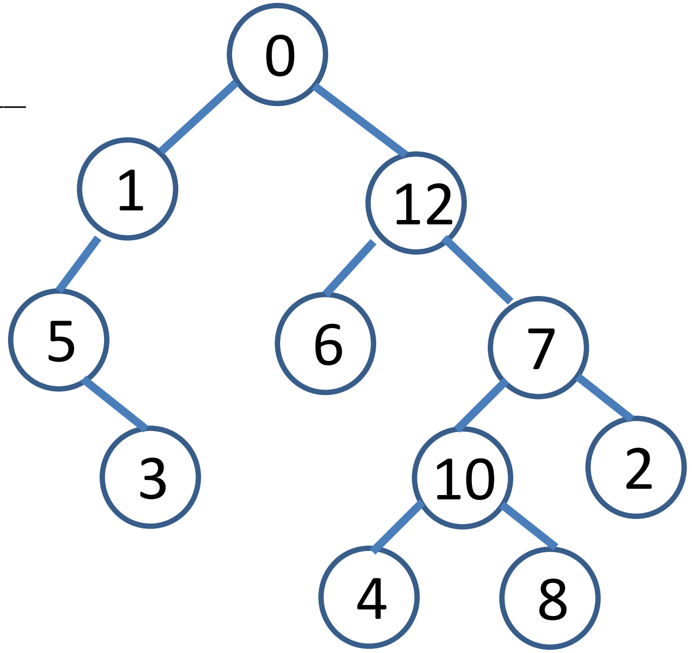
Store one path? Store all paths?

Time & Space complexity?

Easier: Print all paths from the root to leaves.

Level-Order Traversal

```
// Adapted from Sedgewick //Time: _____   Space: _____
void traverse(TreeNodePT h) {
    Queue Q = new_Queue();
    put(Q,h);
    while (!empty(Q)) {
        h = get(Q);//removes and returns first node from Q
        printItem(h->data);
        if (h->left != NULL) put(Q,h->left);
        if (h->right != NULL) put(Q,h->right);
      }
  }
```



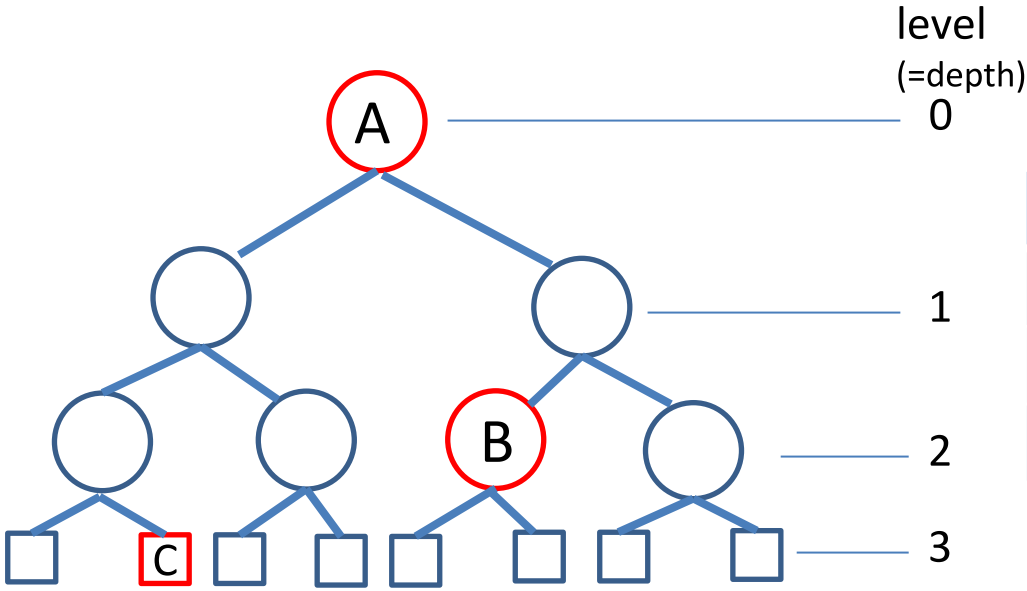
Queue:_________________________

Print:____________________________________

```
// Adapted from Sedgewick
// Same code, but uses a function argument      void
traverse(TreeNodePT h, void (*visit)(TreeNodePT)) {
    Queue Q = new_Queue();
    put(Q,h);
    while (!empty(Q)) {
        h = get(Q); //gets first node
        (*visit)(h);
        if (h->left != NULL) put(Q,h->left);
        if (h->right != NULL) put(Q,h->right);
      }
  }
```

Terminology

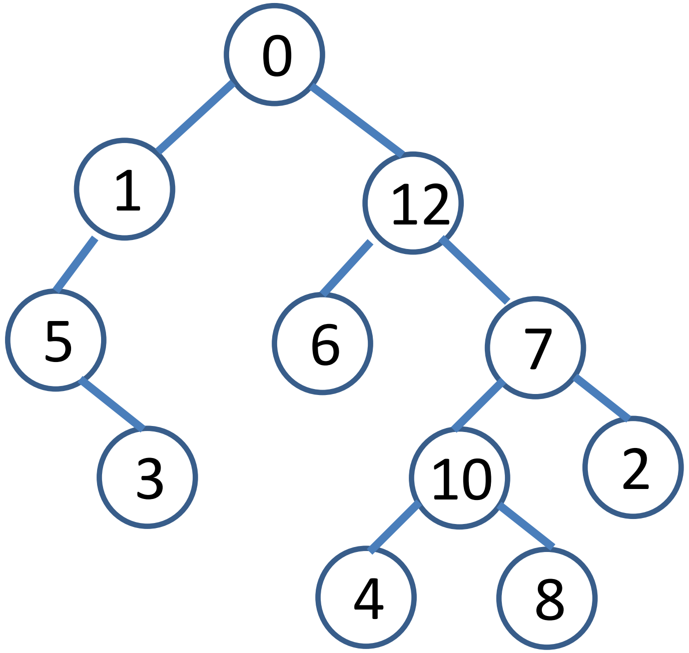
- The ***level*** of the root is defined to be 0.
- The ***level*** of each node is defined to be 1+ the level of its parent.
- The ***depth*** of a node is the number of edges from the root to the node.
  (It is equal to the level of that node.)
- The ***height*** of a node is the number of edges *from the node to the deepest leaf*.
  (Treat that node as the root of a small tree => height = depth_of_deepest leaf)



Node	level	depth	height
A	0	0	3
B	2	2	1
C	3	3	0

Practice:
- Give the level, depth and height for each of the red nodes.
- How many nodes are on each level? __________1, 2, 4, 8_______________

Practice



Node	level	depth	height
0			
6			
7			

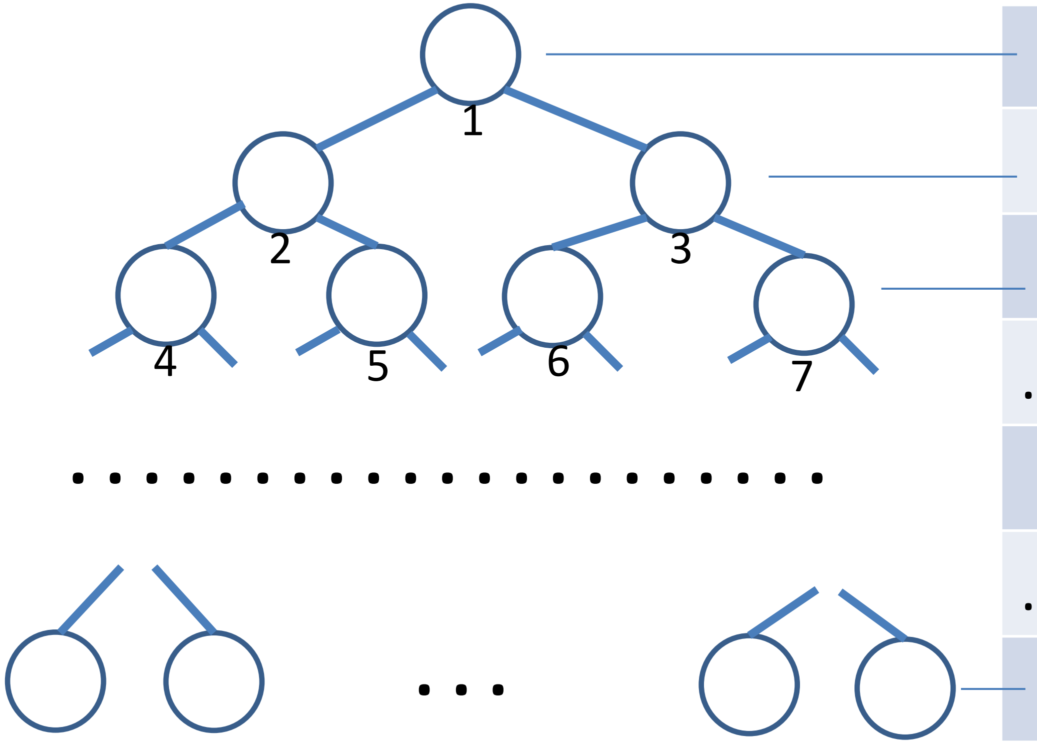
Complete Binary Trees

In the other direction: number of nodes = $2^{\text{height}+1}$ -1

A **Complete binary tree** with N nodes has:

- $\lfloor \lg N \rfloor$ +1 levels
- Height : $\lfloor \lg N \rfloor$
- $\lceil N/2 \rceil$ leaves (half the nodes are on the last level)
- $\lfloor N/2 \rfloor$ internal nodes (half the nodes are internal)

$$\sum_{k=0}^{L} 2^k = 2^{L+1} - 1$$

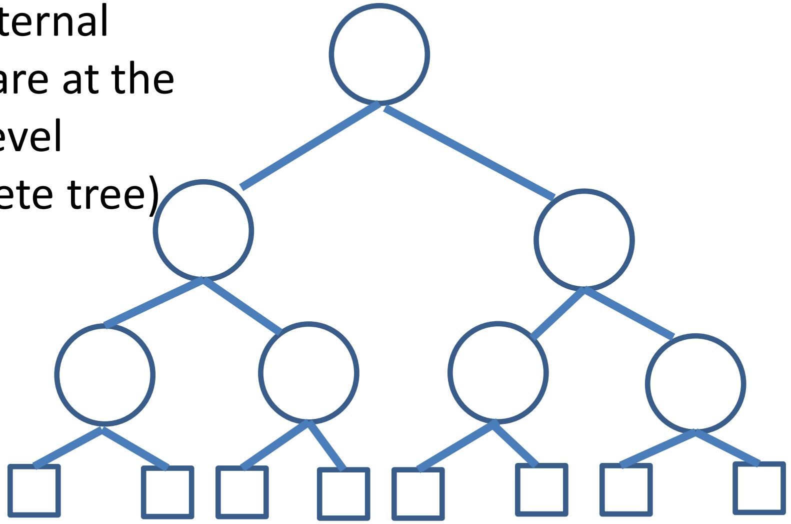


Level	Nodes per level	Sum of nodes from root up to this level
0	2^0 (=1)	$2^1 – 1$ (=1)
1	2^1 (=2)	$2^2 – 1$ (=3)
2	2^2 (=4)	$2^3 – 1$ (=7)
…	…	
i	2^i	$2^{i+1} – 1$
…	…	
L	2^L	$2^{L+1} – 1$

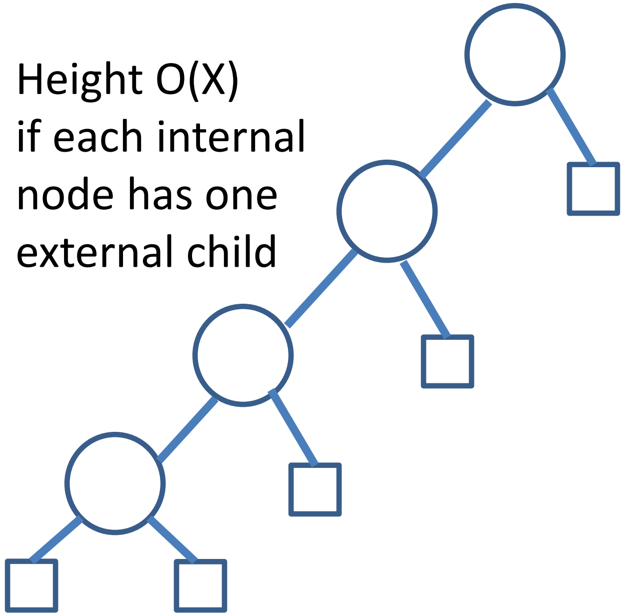
Properties of Full Trees

- **Full** binary tree : every node has exactly 0 or 2 children. No nodes with only 1 child.
- A **full** binary tree with X internal nodes has:
  - X+1 external nodes.
  - 2X edges (links).
  - N = 2X+1 (total number of nodes)
  - height at least lg X and at most X:

Height O(lgX)
if all external
nodes are at the
same level
(complete tree)

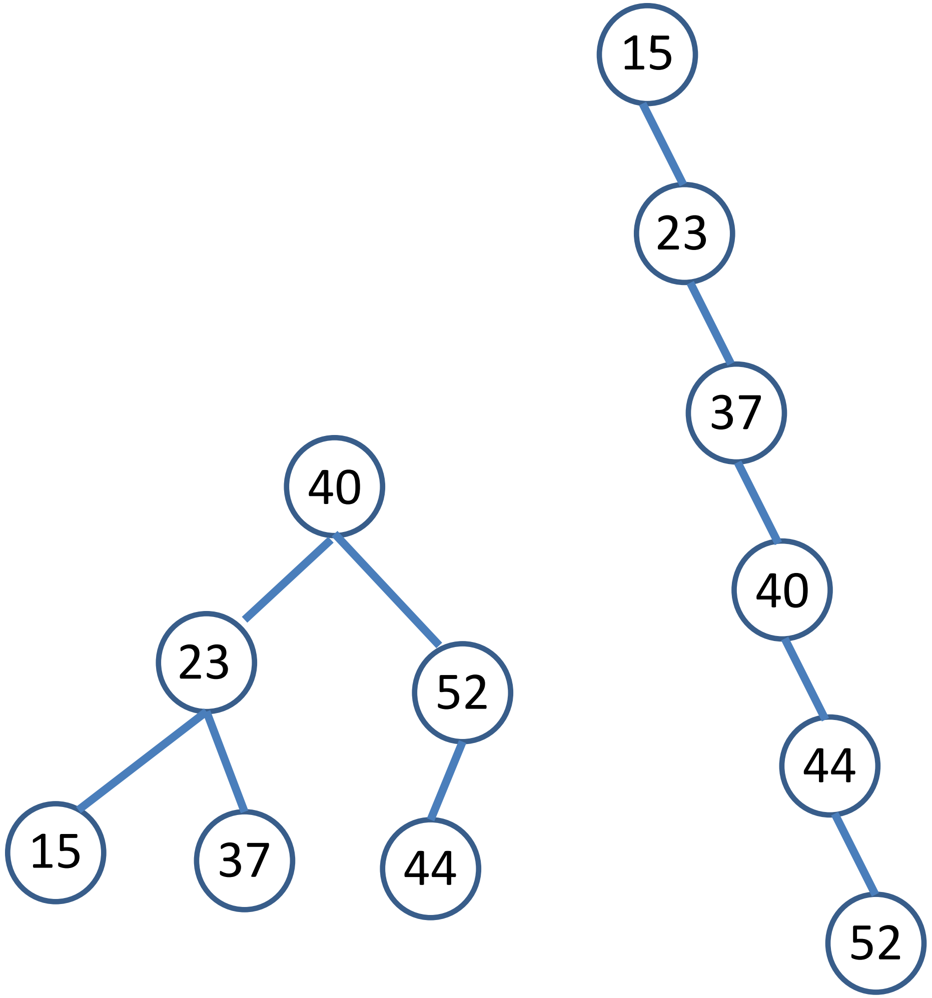


Height O(X)
if each internal
node has one
external child



Binary tree shape and height

- A binary tree with N nodes has height
  - at least Θ(lgN)
  - at most Θ(N)
- We CANNOT assume any binary tree with N nodes has height Θ(lgN)
  - discuss best and worst case
  - use worst case in final/general TC
- Some special trees will maintain the Θ(lgN) height by preserving some properties of the tree over insertion and deletion.



General Trees

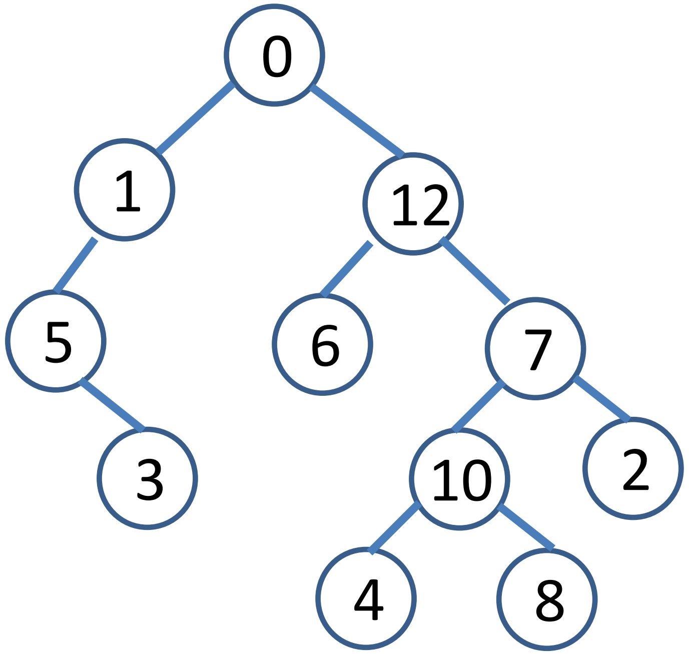
- In a general tree, a node can have any number of children.
- Left-child - right-sibling implementation
  - Draw tree and show example
  - (There is a one-to-one correspondence between ordered trees and binary trees)

EXTRA – Recursive and Iterative Preorder Traversal (Sedgewick)

```
// Visit function is an argument to traversal
void travGen(TreeNodePT h, void (*visit)(TreeNodePT))  {
    if (h == NULL) return;
    (*visit)(h);
    travGen (h->left, visit);
    travGen (h->right, visit);
}
```

Passing functions as arguments to functions using function pointers

```
void traverse(TreeNodePT h)  { // recursive
    if (h == NULL) return;
    printNode(h);
    traverse(h->left, visit);
    traverse(h->right, visit);
  }
-----
void traverse(TreeNodePT h) {  // iterative
    Stack S = newStack(max);
    push(S,h);
    while (!isEmpty(S)) {
        h = pop(S);
        printNode(h);
        if (h->right!= NULL)  push(S,h->right);
        if (h->left != NULL)  push(S,h->left);
      }
  }
```



Stack:

Print: