CS 214 – DATA STRUCTURES FALL 2015

Lecture 7 – Linked Lists

Group **B & C**

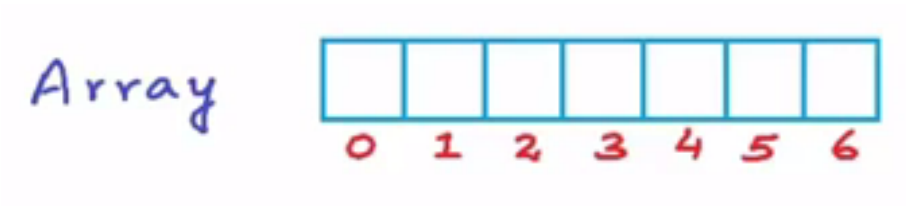
November 13th 2017

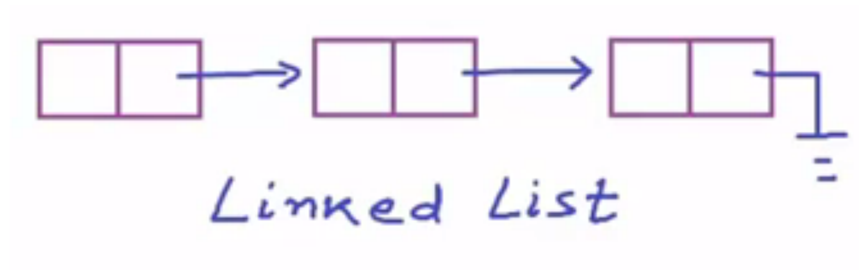
Dr. Mai Hamdalla

mai@fci.helwan.edu.eg

TREES

Linear Data Structures



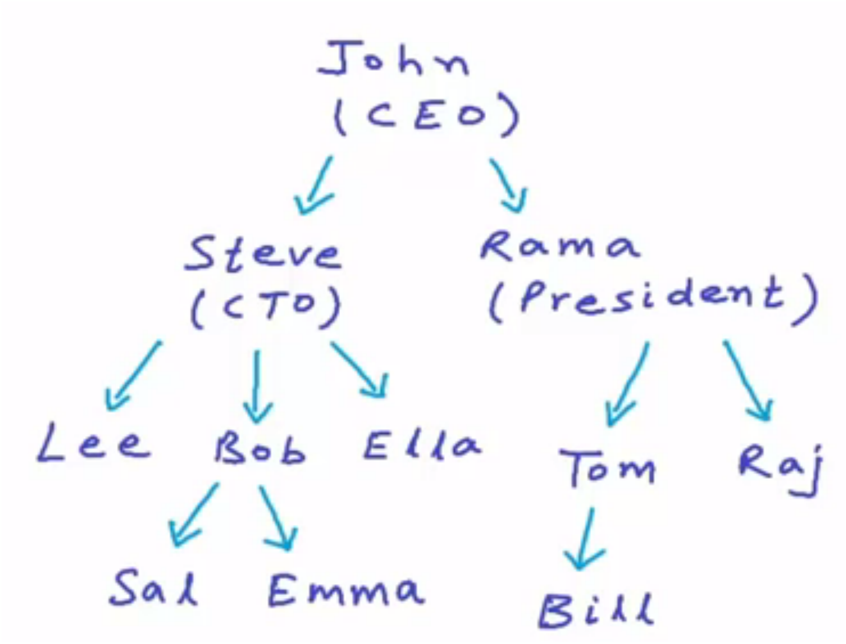


Tree ADT

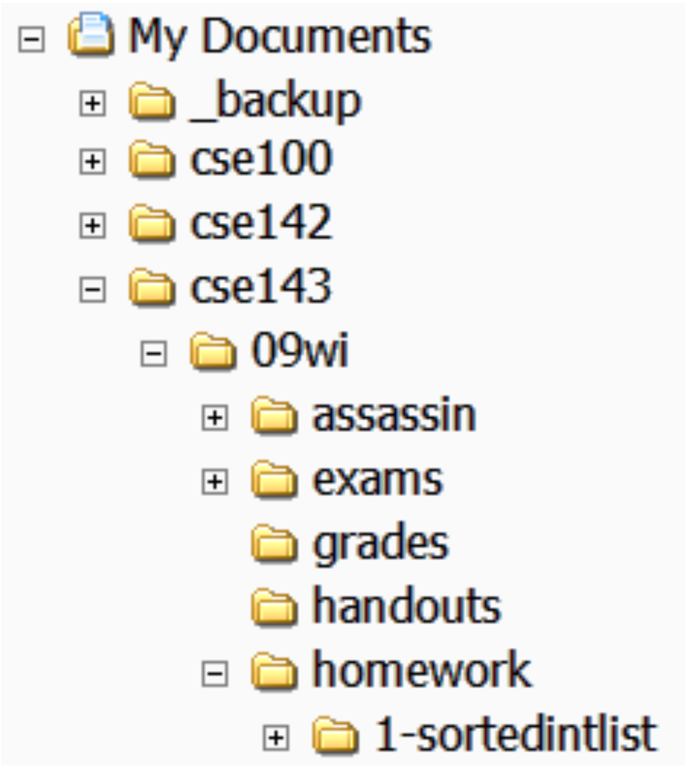
- A non-linear, directed, acyclic structure of linked nodes.
  - *directed*: Has one-way links between nodes.
  - *acyclic*: No path wraps back around to the same node twice.

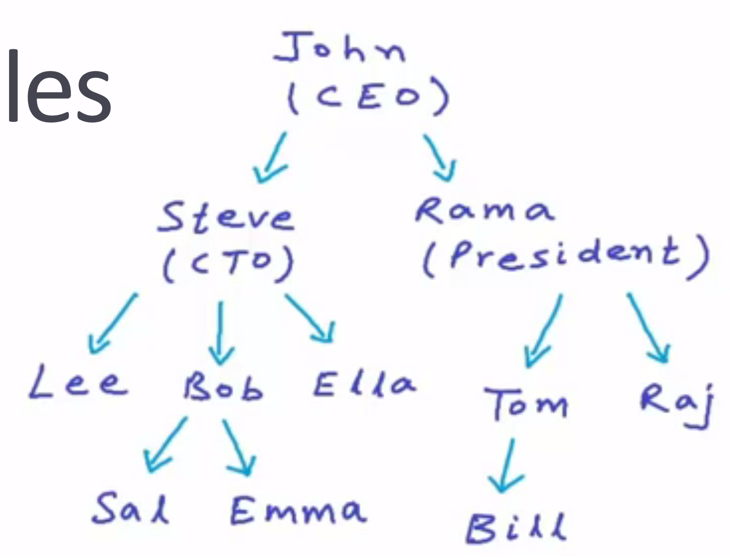
- **binary tree**: One where each node has at most two children.

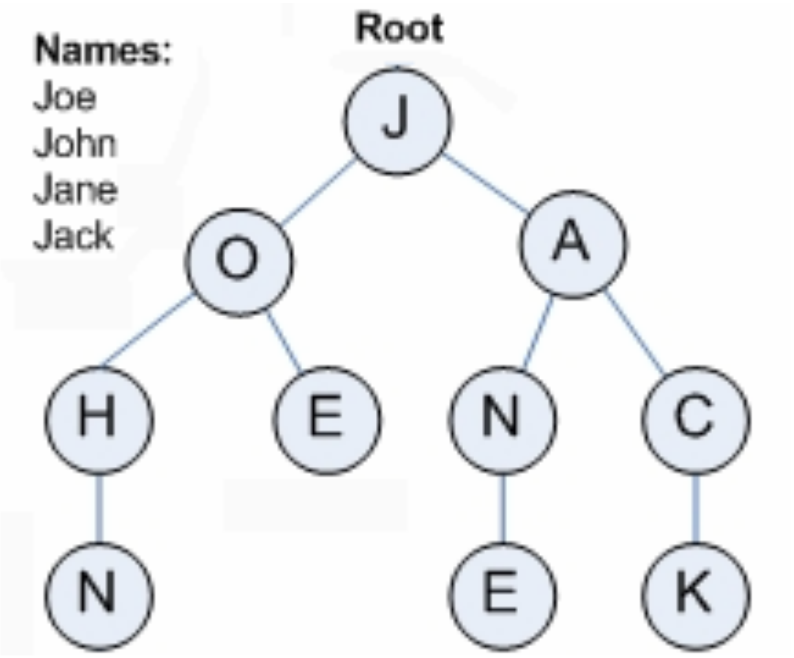
Tree Structures



Tree Structures - Examples





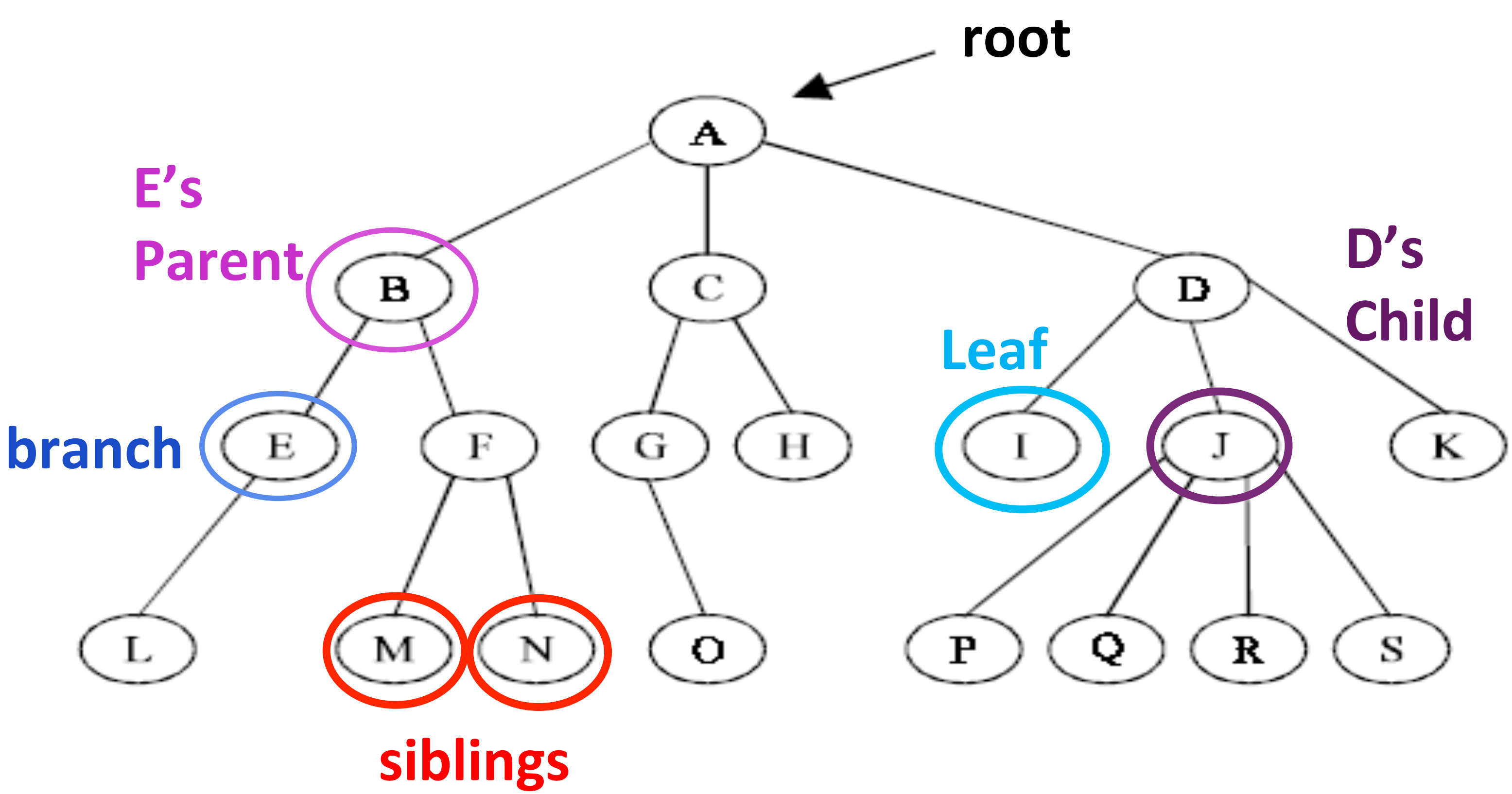


Terminology (I)

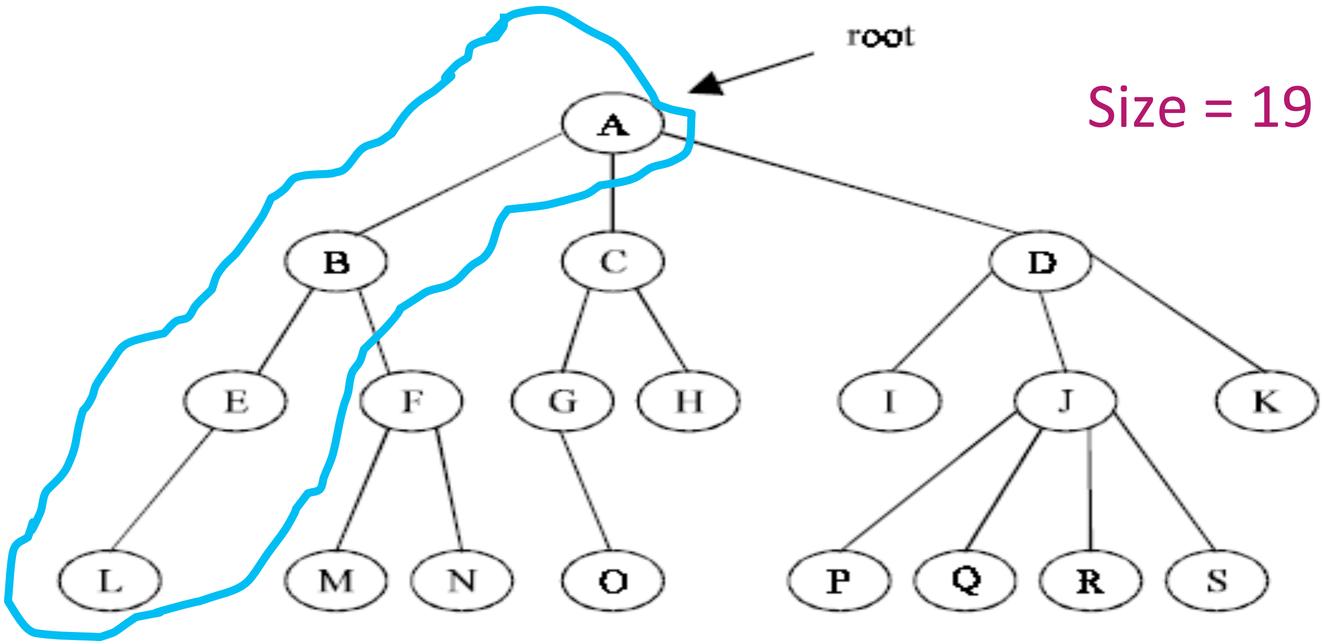
- **Node**: an object containing a data value and children links.
  - **root**: the topmost node of a tree.
  - **leaf**: a node that has no children.
  - **branch**: any internal node; neither the root nor a leaf

  - **parent**: a node that has at least one descendant.
  - **child**: a node that has an ancestor.
  - **sibling**: a node with a common parent node.

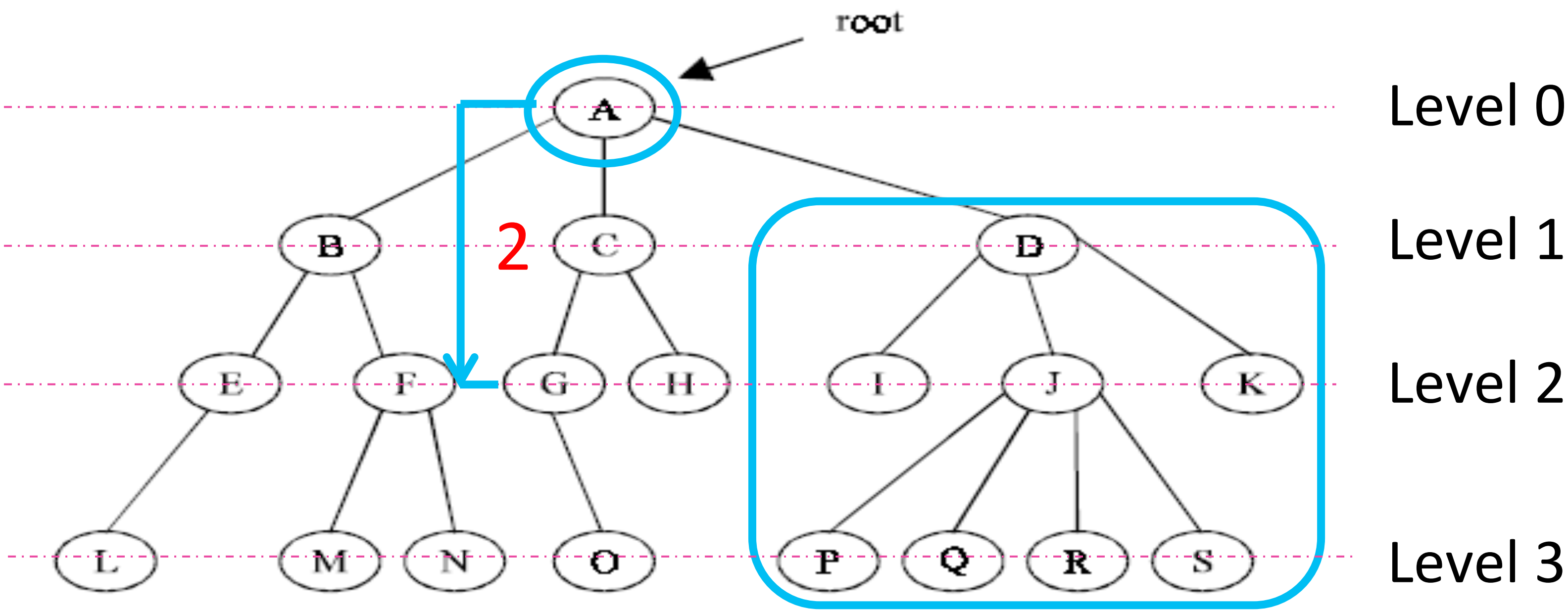
Terminology (II)



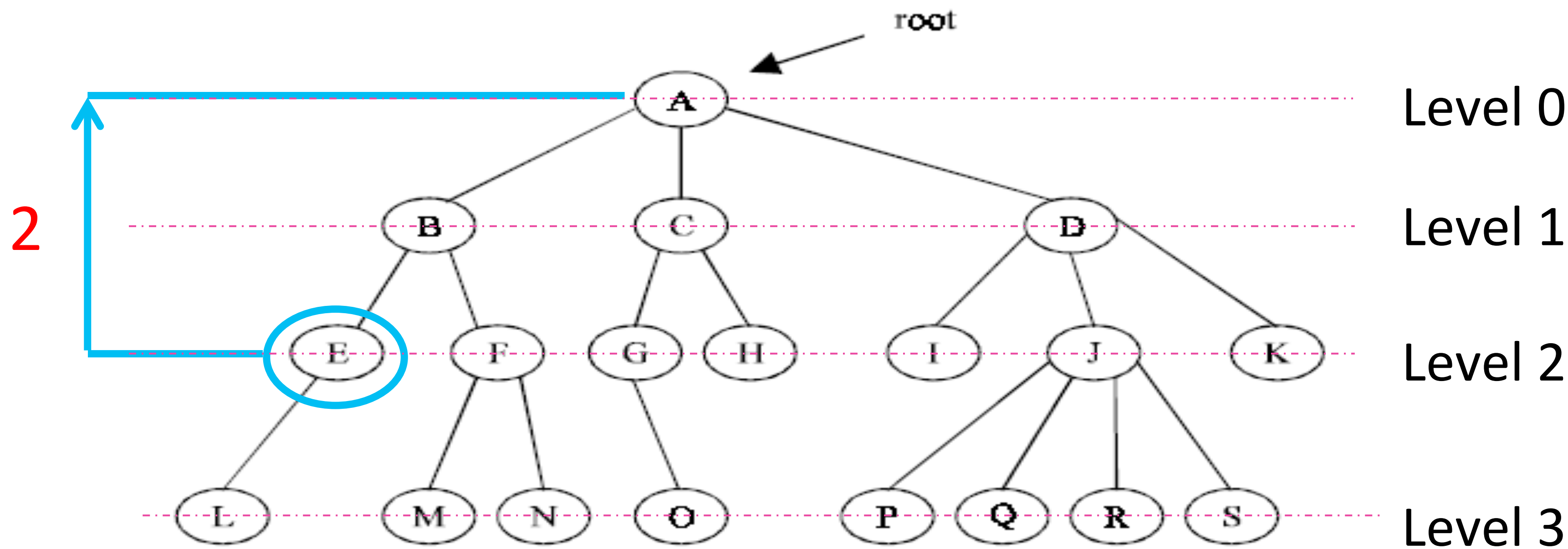
Terminology (III)



- **Path**: a sequence of edges from one node to another.
- **Size**: the number of nodes in a tree.



- **Subtree**: the smaller tree of nodes within a tree.
- **Height**: length of the longest path from the root to any node.
- **Level** : length of the path from a root to a given node

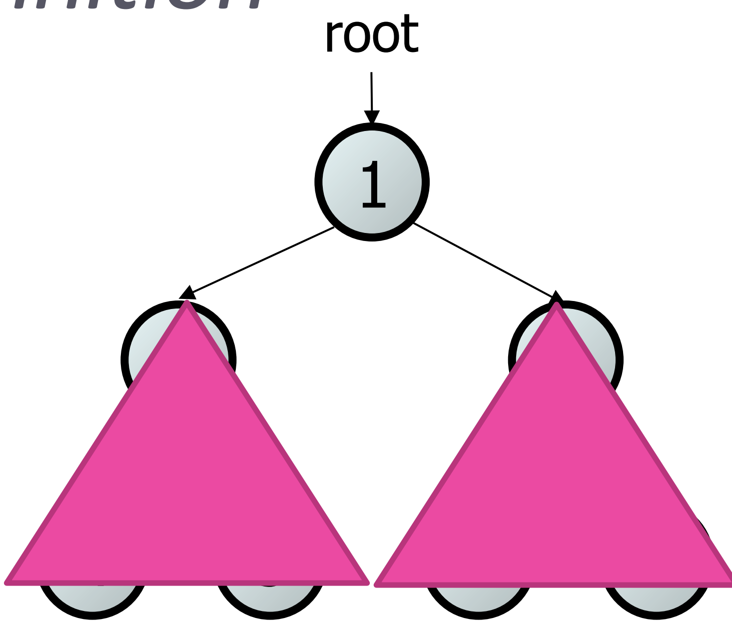


- **Depth of a node:** is its distance from the root.
  - **A** is at depth zero.
  - **E** is at depth 2.
- **Depth of a tree:** is the depth of its deepest node.
  - This tree has depth 3

BINARY TREE

Binary Tree - *Recursive definition*

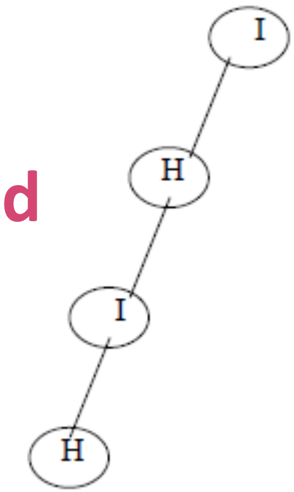
- A tree is either:
  - empty (null), or
  - a **root** node (vertex) that contains:
    - **data**,
    - a **left** subtree, and
    - a **right** subtree.

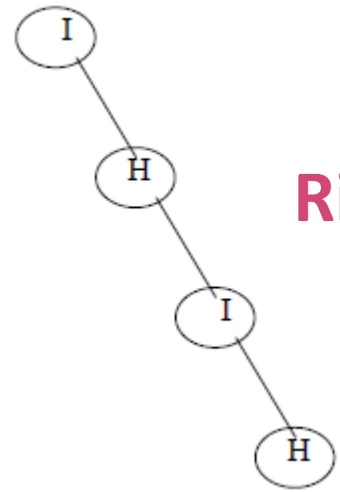


The left and/or right subtree could be empty.

Binary Trees

Left Skewed

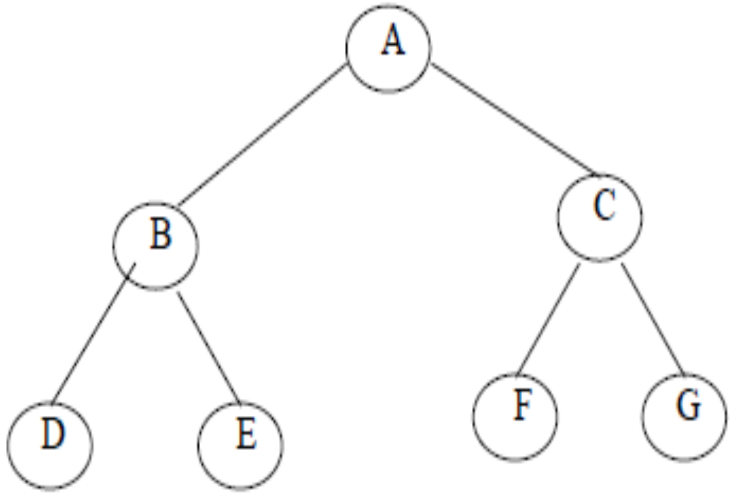


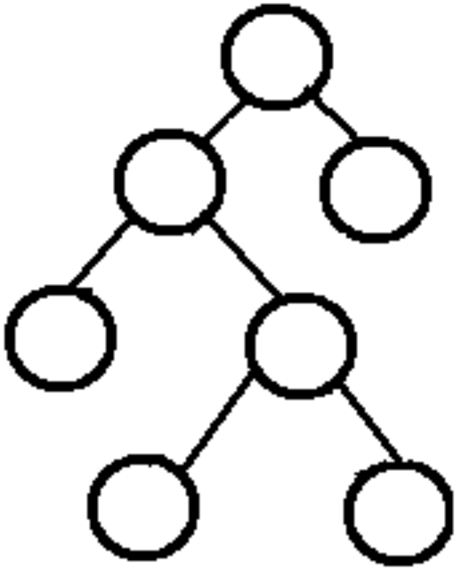


Right Skewed

- If a binary tree has **only right sub trees**, then it is called right skewed binary tree.
- If a binary tree has only left sub trees, then it is called left skewed binary tree.

Full Binary Tree

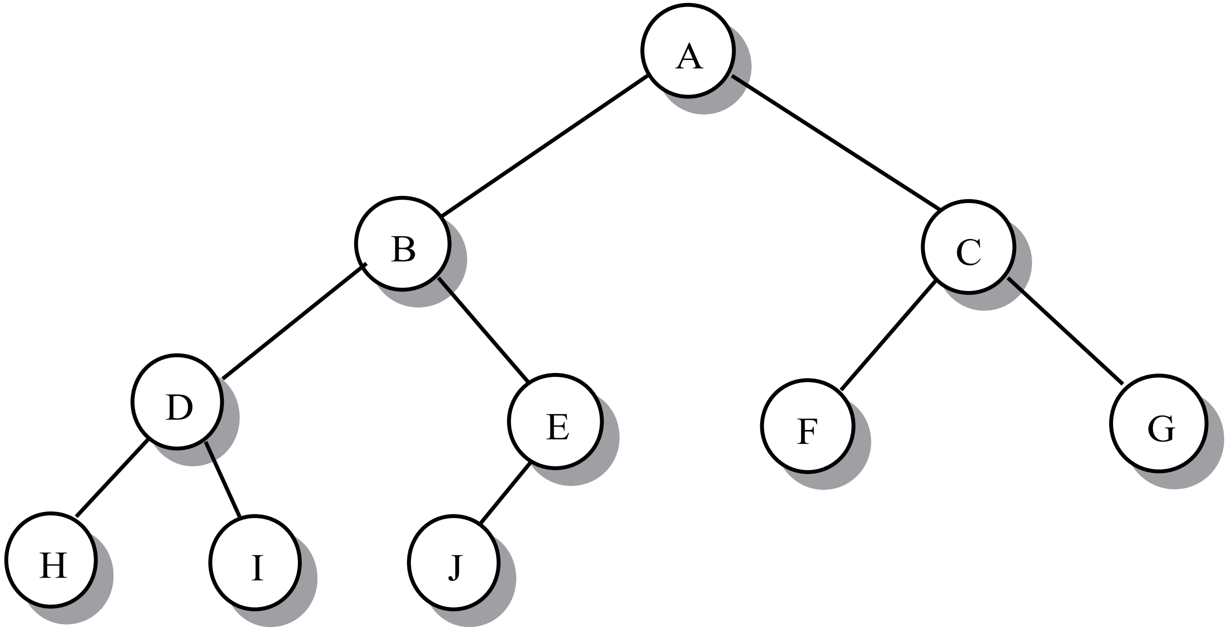




- A binary tree in which every node other than the leaves has **<u>exactly</u>** two children.

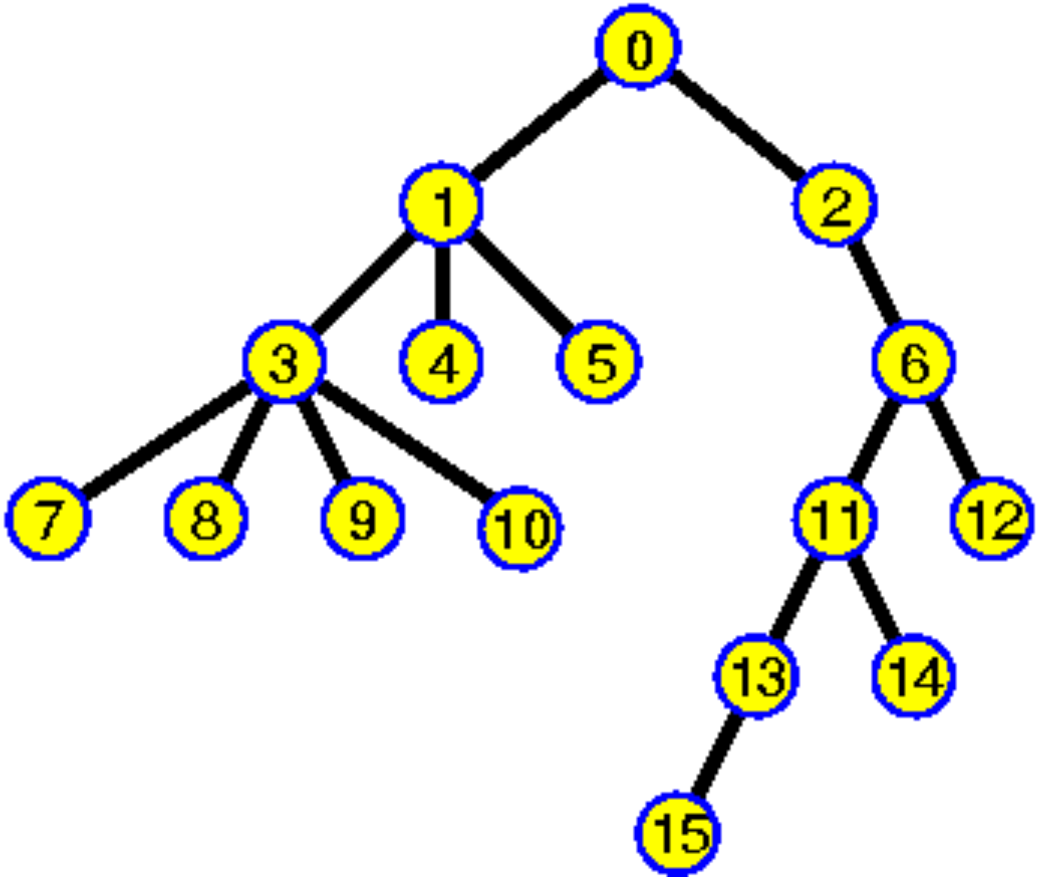
Complete Binary Tree

- A binary tree in which every level, except possibly the last, is **completely filled**, and all nodes are as far left as possible.



Complete Tree (Depth 3)

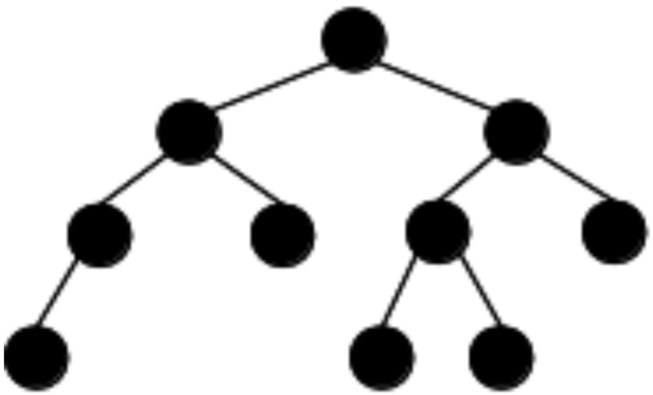
TREE DEFINITIONS



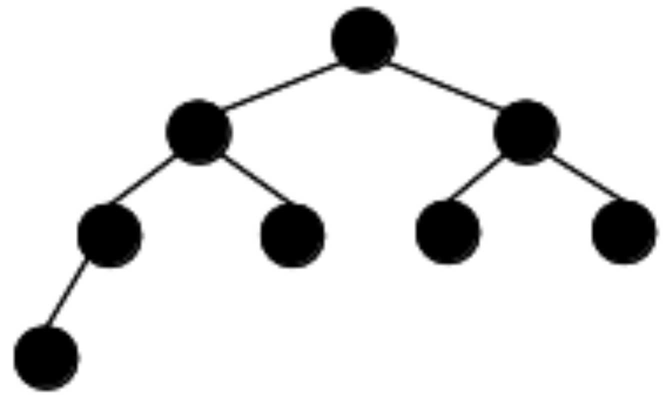
Tree has 16 nodes
Tree has degree 4
Tree has depth 5
Node 0 is the root
Node 1 is internal
Node 4 is a leaf
4 is a child of 1
1 is the parent of 4
0 is grandparent of 4
3, 4 and 5 are siblings

Selected Samples of Binary Trees (I)

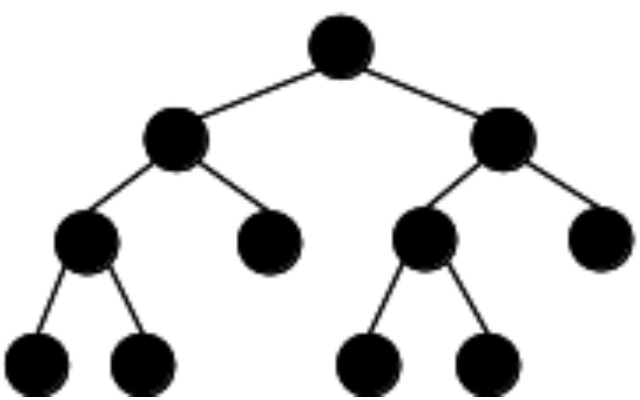
Neither complete nor full



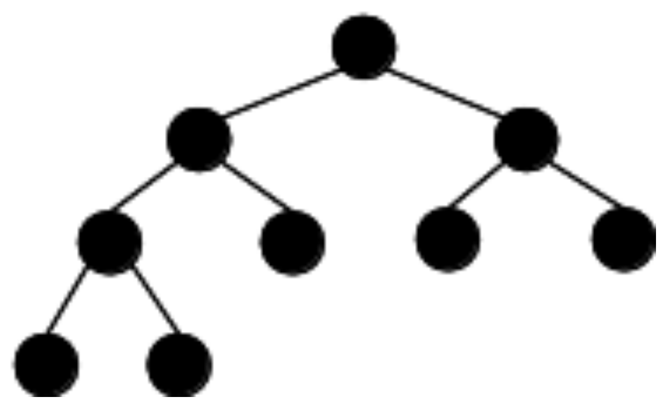
Complete but not full



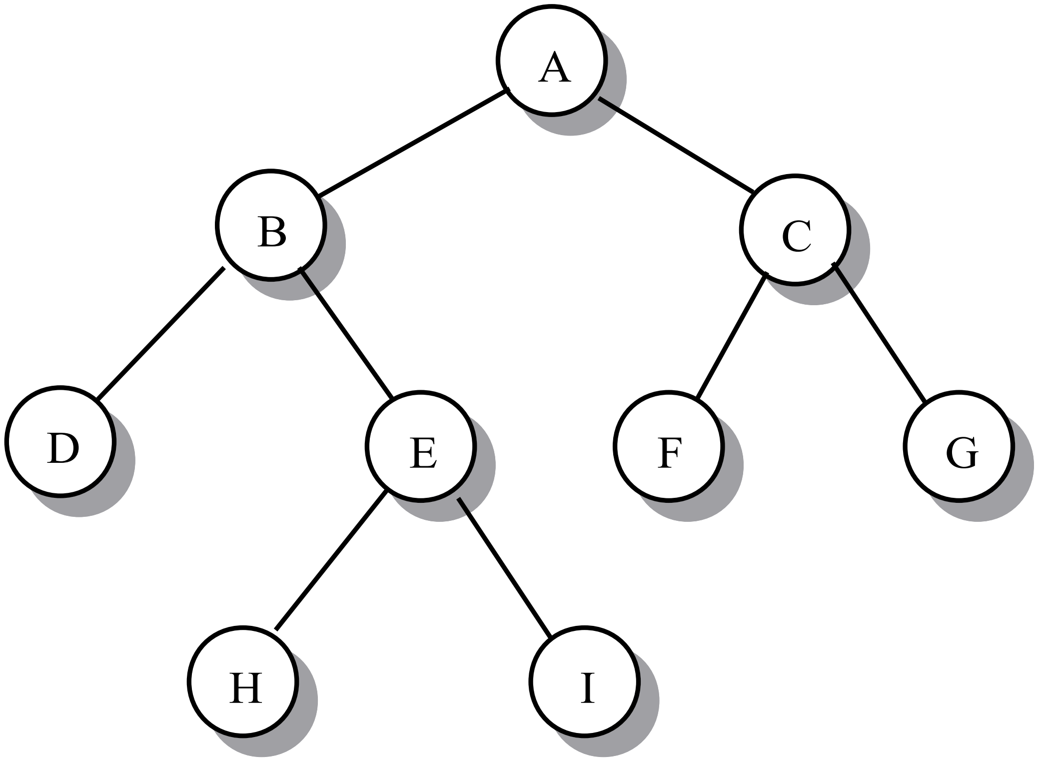
Full but not complete



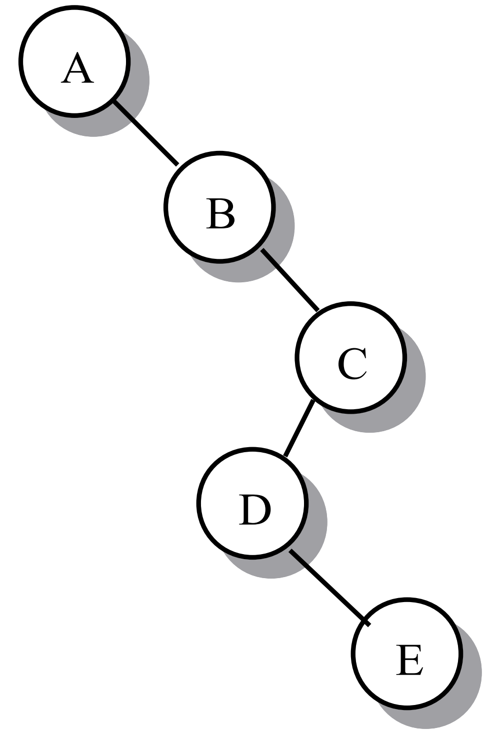
Complete and full



Selected Samples of Binary Trees (II)



Tree A
Size 9 Depth 3



Tree B
Size 5 Depth 4

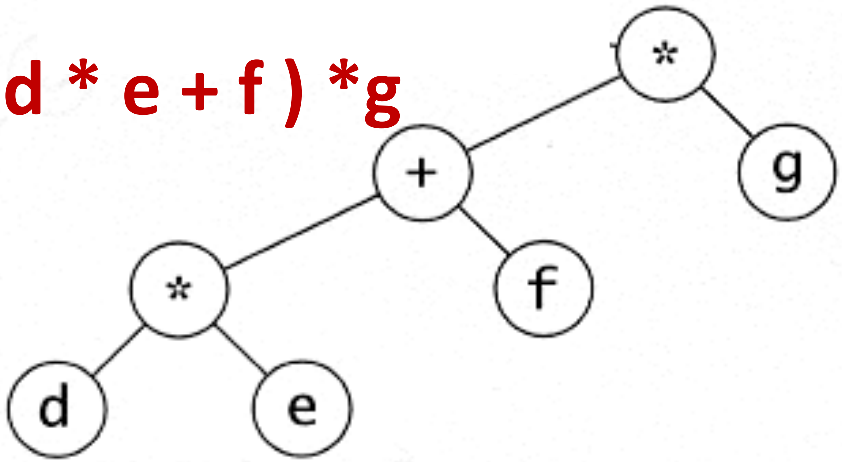
EXPRESSION TREES

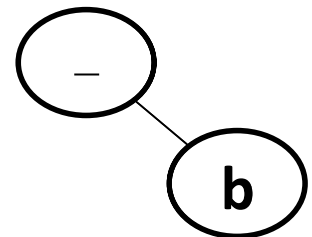
Example: Expression Trees

- It is a binary tree contains an arithmetic expression with some operators and operands.
- Leaves are **operands** (constants or variables)
- The internal nodes contain **operators**
- For each node contains an operator, its left subtree gives the left operand, and its right subtree gives the right operand.
- Great importance in syntactical analysis and parsing, along with the validity of expressions

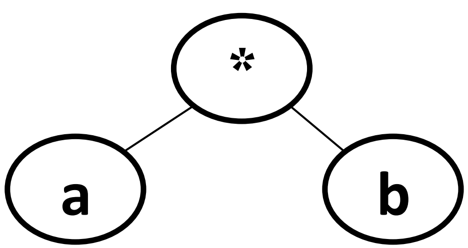
Expression Trees

(d * e + f) *g

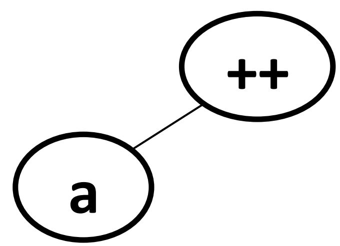




- b



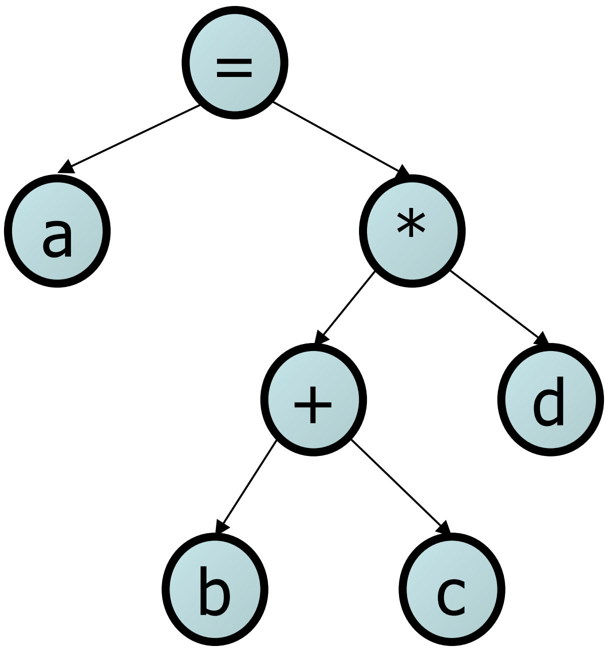
a* b



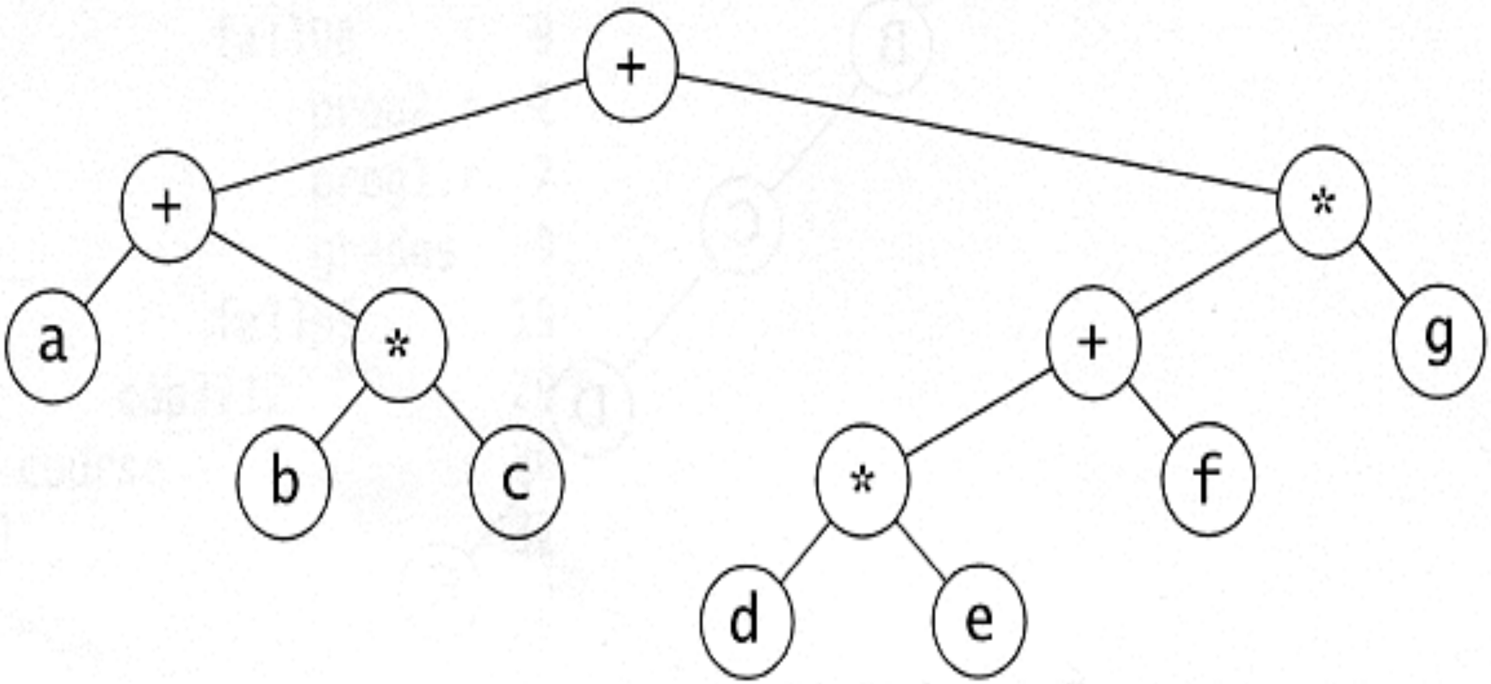
a++

Expression Trees

- a = (b + c) * d



Expression Trees



Expression tree for (a + b * c) + ((d * e + f) * g)

BINARY TREE TRAVERSAL

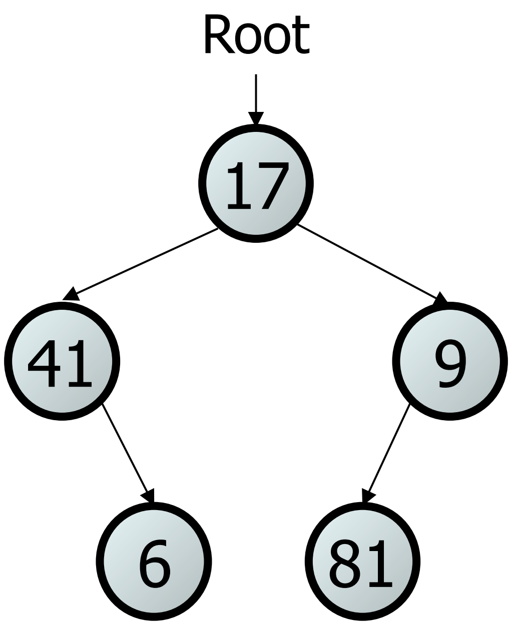
Tree Traversals

- An examination of the elements of a tree.
- Common orderings for traversals:
  - **pre-order**: process root node, then its left subtrees followed by its right subtree
  - **in-order**: process left subtree, then root node, then right subtree
  - **post-order**: process left/right subtrees, then root node

Traversal example

- in-order (LVR):

```
41
6
17
81
9
```



Traversal example

- pre-order (VLR):

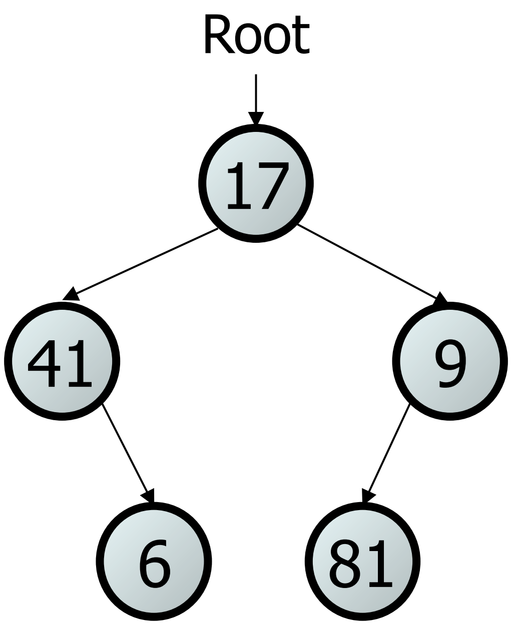
  17

  41

  6

  9

  81



Traversal example

- post-order (LRV):

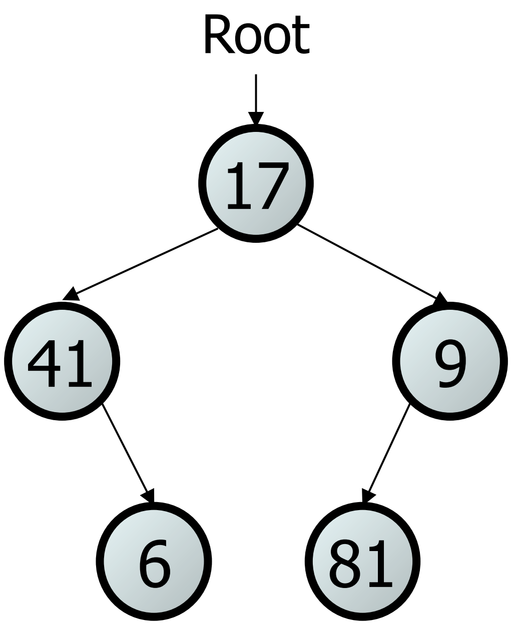
  6

  41

  81

  9

  17



Exercise

- **Pre-order Traversal**:

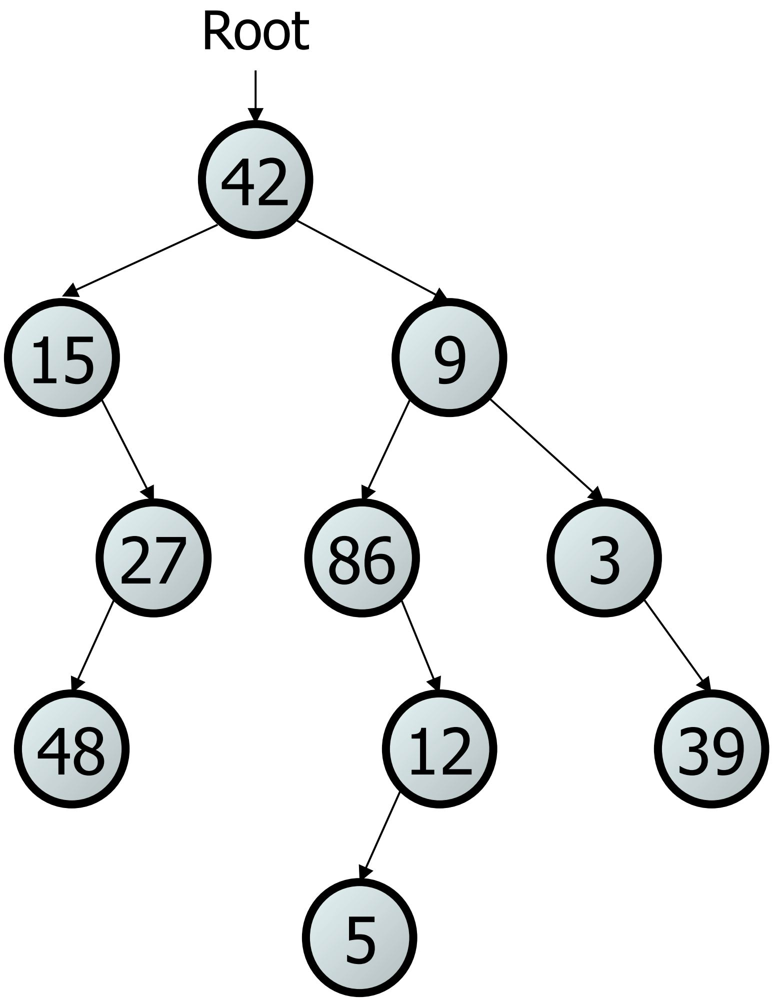
  42 15 27 48 9 86 12 5 3 39

- **In-order Traversal:**

  15 48 27 42 86 5 12 9 3 39

- **Post-order Traversal**:

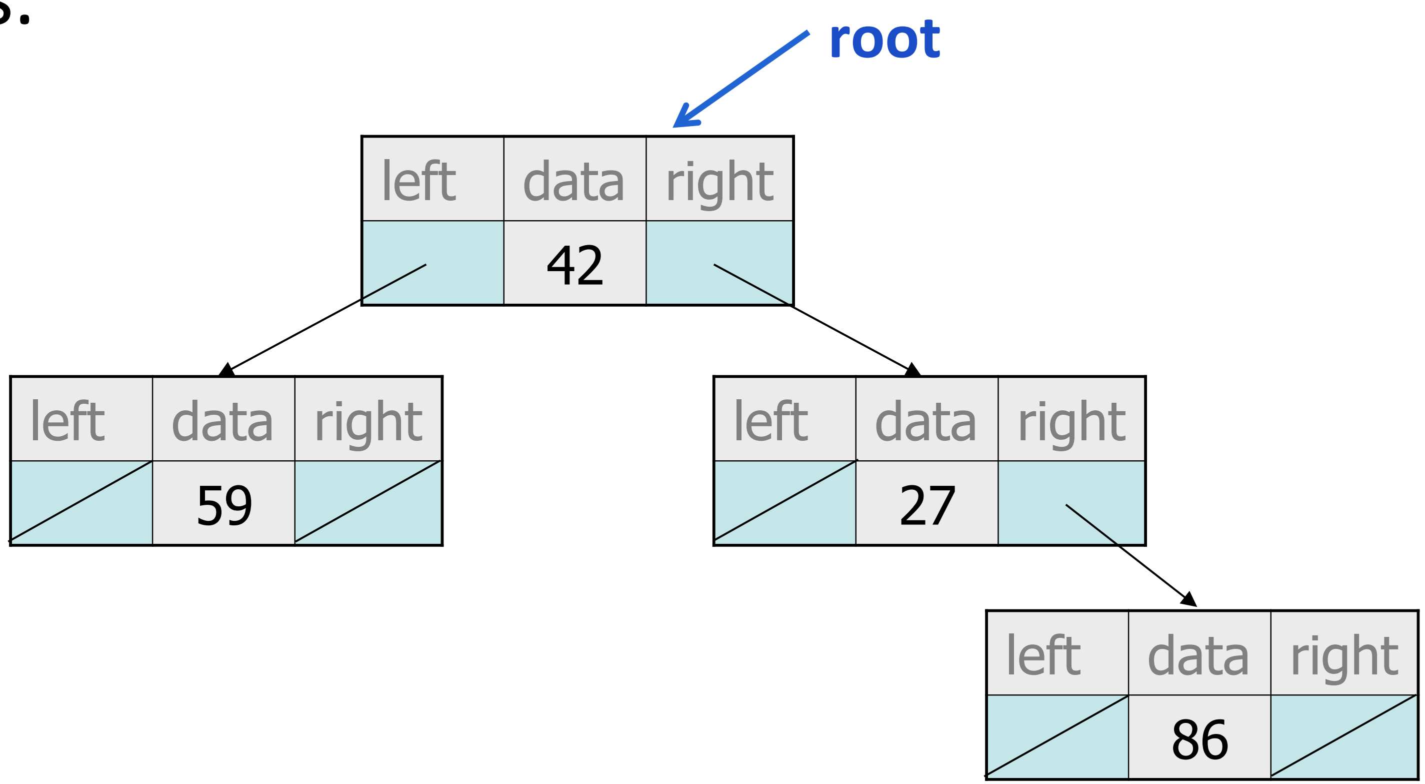
  48 27 15 5 12 86 39 3 9 42



BINARY TREE IMPLEMENTATION

A tree node for integers

- A basic **tree node** stores data and left/right links.



Basic Binary Tree Operations

- Create the tree, leaving it empty.
- Determine whether the tree is empty or not
- Determine whether the tree is full or not
- Find the size of the tree.
- Find the depth of the tree.
- Traverse the tree, visiting each entry
- Clear the tree to make it empty

Tree Implementation

```
typedef struct node_type{
  entry_type info;
  node_type *right,*left;
} tree_node;

typedef tree_node *tree;
```

Tree Initialization

Pre: None.

Post: The tree is initialized to be empty.

```
void create_tree(tree *t){
    *t = NULL;
}
```

In-order Tree Traversal

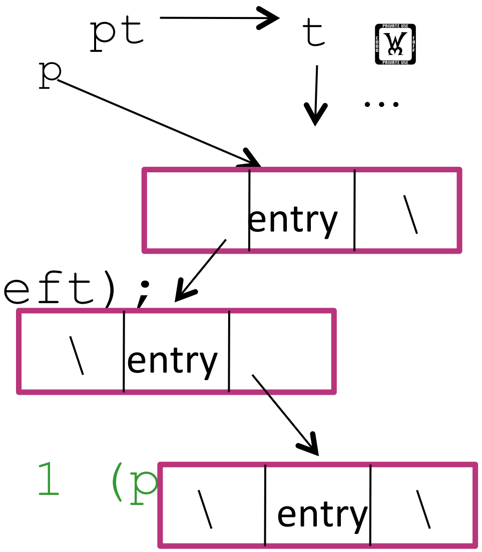
Pre: The tree is initialized.

Post: The tree has been traversed in infix order sequence.

```
void inorder_traversal(tree t,
                       void(*pvisit)(entry_type)){
   if(t){
      inorder_traversal(t->left, pvisit);
      (*pvisit)(t->info);
      inorder_traversal(t->right, pvisit);
   }
}
```

How can we write inorder_traversal Iteratively?

```
void inorder_traversal(tree *pt,
                         void (* pvisit)(entry_type)){
   stack s;
   void *p=(void *)(*pt);
/* p will be Pushed in the stack; we do not define it
TreeNode * to avoid modifying Stack.h to include Tree.h*/
   if(p){
      create_stack(&s);
      do{
         while(p){
            push(p, &s);
            p=(void *)(((tree_node *)p)->left);
         }
         if(!is_stack_empty(&s)){
         //This is redundant check; always 1 (p
            pop(&p, &s);
            (*pvisit) (((tree_node *)p)->entry);
            p=(void *)(((tree_node *)p)->right);
         }
      }while(!is_stack_empty(&s) || p);
   }}
```



Pre-order Tree Traversal

Pre: The tree is initialized.

Post: The tree has been traversed in prefix order sequence.

```
void preorder_traversal(tree_type t,
        void(*pvisit)(entry_type)){
    if(t){
        (*pvisit)(t->info);
        Preorder(t->left, pvisit);
        Preorder(t->right, pvisit);
    }
}
```

Post-order Tree Traversal

Pre: The tree is initialized.

Post: The tree has been traversed in Postfix order.

```
void post_order(tree t,
        void(*pvisit)(EntryType)){
  if(t){
     post_order(t->left, pvisit);
     post_order(t->right, pvisit);
     (*pvisit)(t->info);
  }
}
```

Tree Size

```
int tree_size(tree t){
    if (!t)
        return 0;
    return(1 + tree_size (t->left)
             +tree_size (t->right));
}
```

Tree Depth

```
int tree_depth(tree t){
    if (!t)
        return 0;
    int a= tree_depth(t->left);
    int b= tree_depth(t->right);
    return (a>b)? 1+a : 1+b;
}
```

Clear Tree

```
void clear_tree(tree *t){
  if (*t){
    clear_tree(&(*t)->left);
    clear_tree(&(*t)->right);
    free(*t);
    *t=NULL;
  }
}
```

Tree Empty

Pre: The tree is initialized.

Post: If the tree is empty (1) is returned, otherwise (0) is returned.

```
int is_tree_empty(tree t){
    return (!t);
}
```

Tree Full

Pre: The tree is initialized.

Post: If the tree is full (1) is returned. Otherwise (0) is returned.

```
int is_tree_full(tree t){
    return 0;
}
```

CS 214 – DATA STRUCTURES FALL 2015

Lecture 7 – Linked Lists

Group **B & C**

November 13th 2017

Dr. Mai Hamdalla

mai@fci.helwan.edu.eg