Data Structures and Algorithms Design

DSECLZG519

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Parthasarathy

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Contact Session #4

DSECLZG519 – Introduction to Tree

Agenda for CS #4

1) Recap of CS#3
2) Introduction to Trees
   - Tree terminologies
   - Binary tree and types
   - Representation of tree
     - Using array
     - Using linked list
   - Binary tree traversals
     - Inorder
     - Preorder
     - Postorder
3) Applications of binary trees
4) Q&A

Linear data structures

Here are some of the data structures we have studied so far:

- Arrays
- Singly-linked lists and doubly-linked lists
- Stacks and queues
- These all have the property that their elements can be adequately displayed in a straight line
- How to obtain data structures for data that have non-linear relationships?

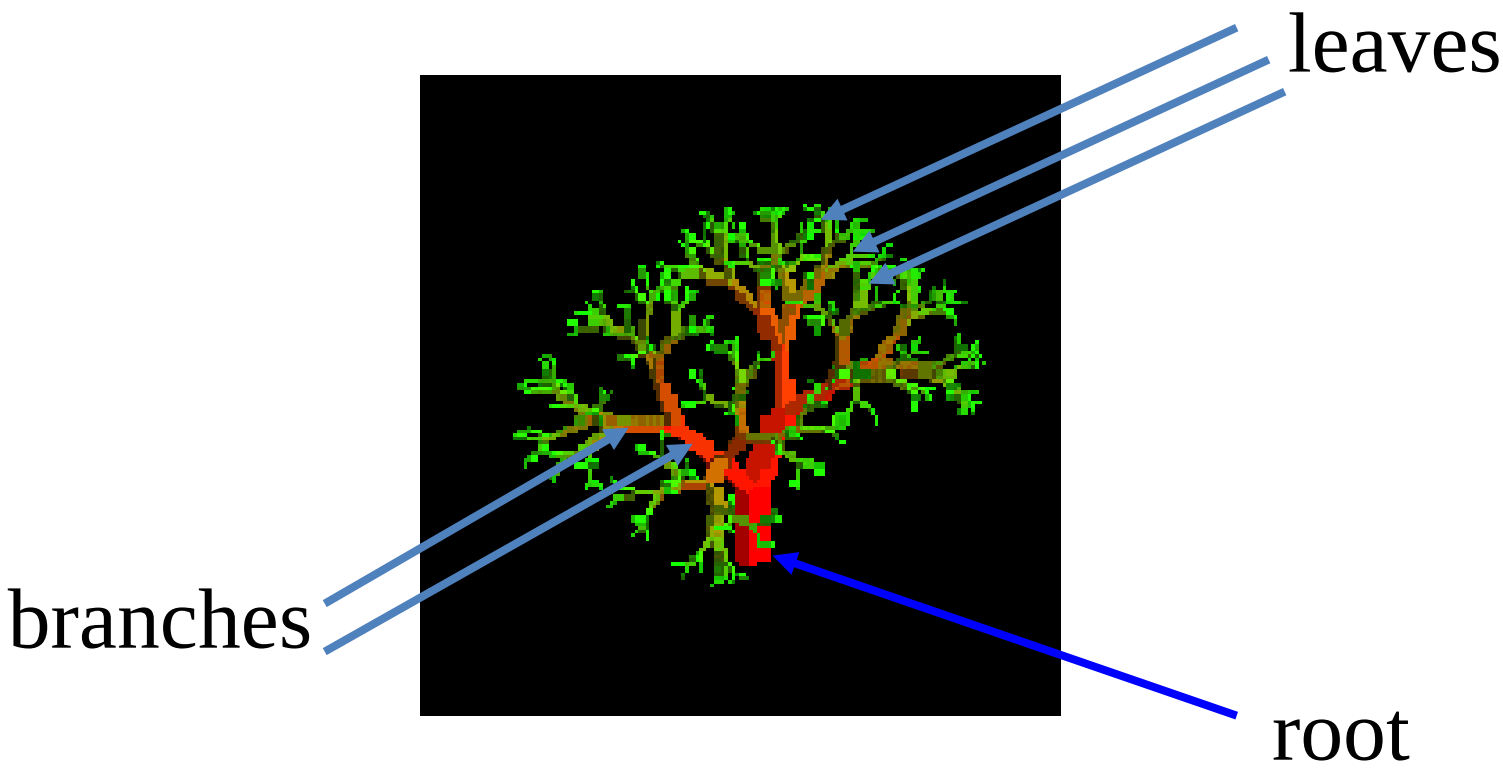
How should one decide which data structure to use ?

- What need to be stored
- Cost of operations
- Memory use
- Ease of implementation ☺

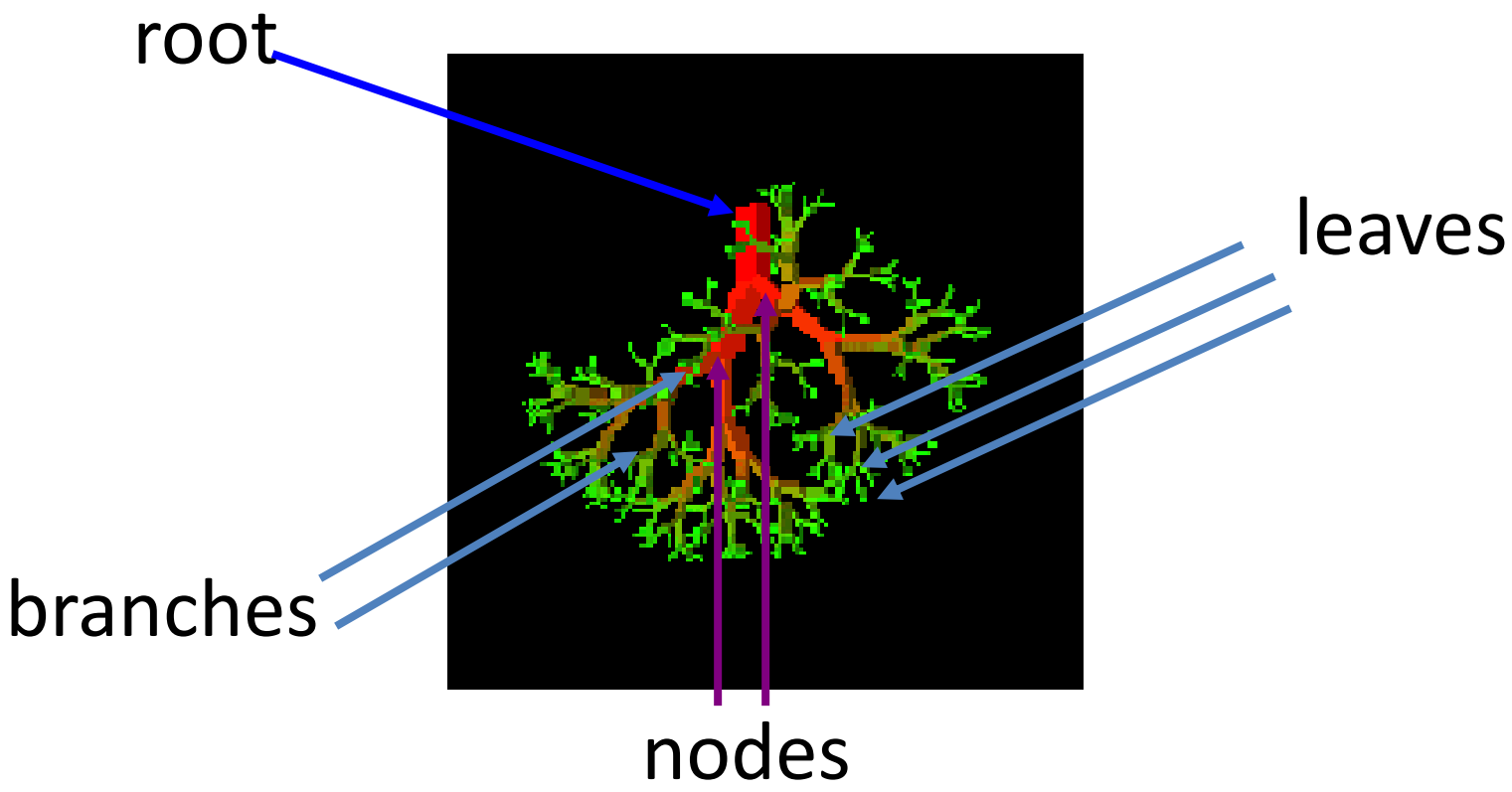
Non-linear Data Structures

- Linear access time of linked lists is expensive
  - Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(log N)?
- Trees
  - Basic concepts
  - Tree traversal
  - Binary tree
  - Binary search tree and its operations
- Stores elements hierarchically
- Top element is called as **root**
- Example
  - Directory structure
  - Family tree
  - Organizational structure , Content of a book etc.

Natural View of a Tree

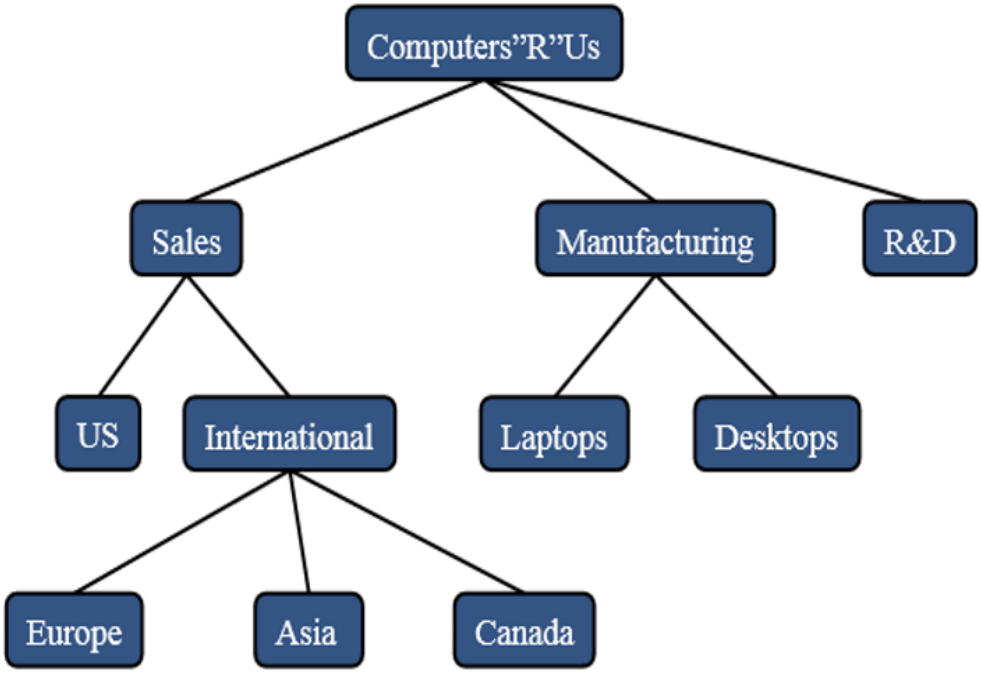


Computer Scientist's View



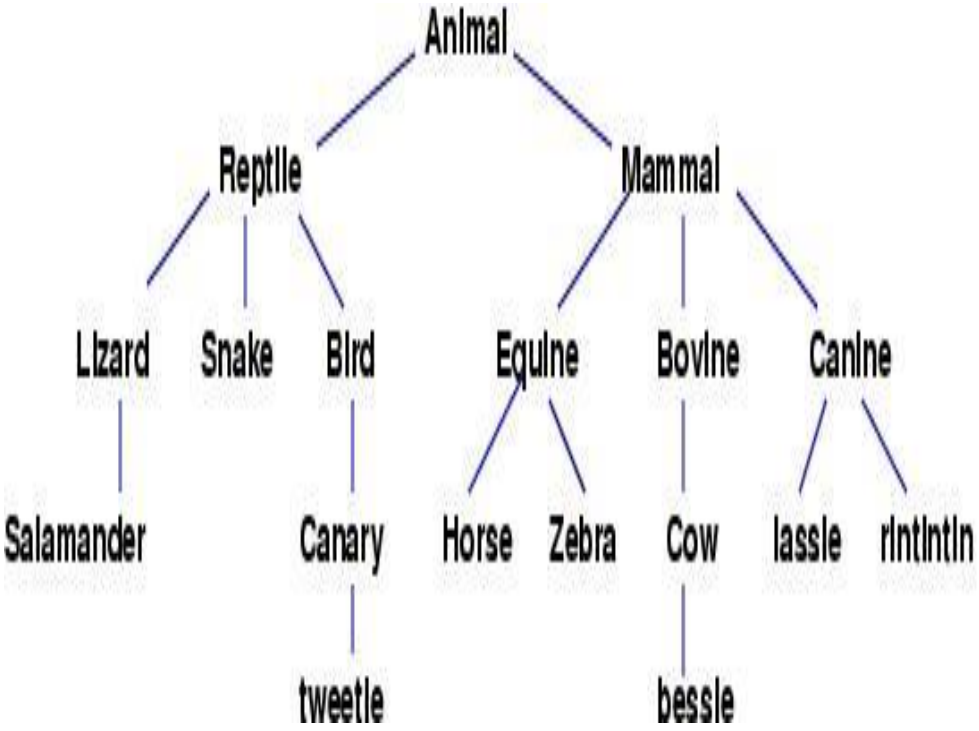
Tree: A Hierarchical ADT

- A tree (upside down) is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relation.
- Each element (except the top element) has a parent and zero or more children elements.



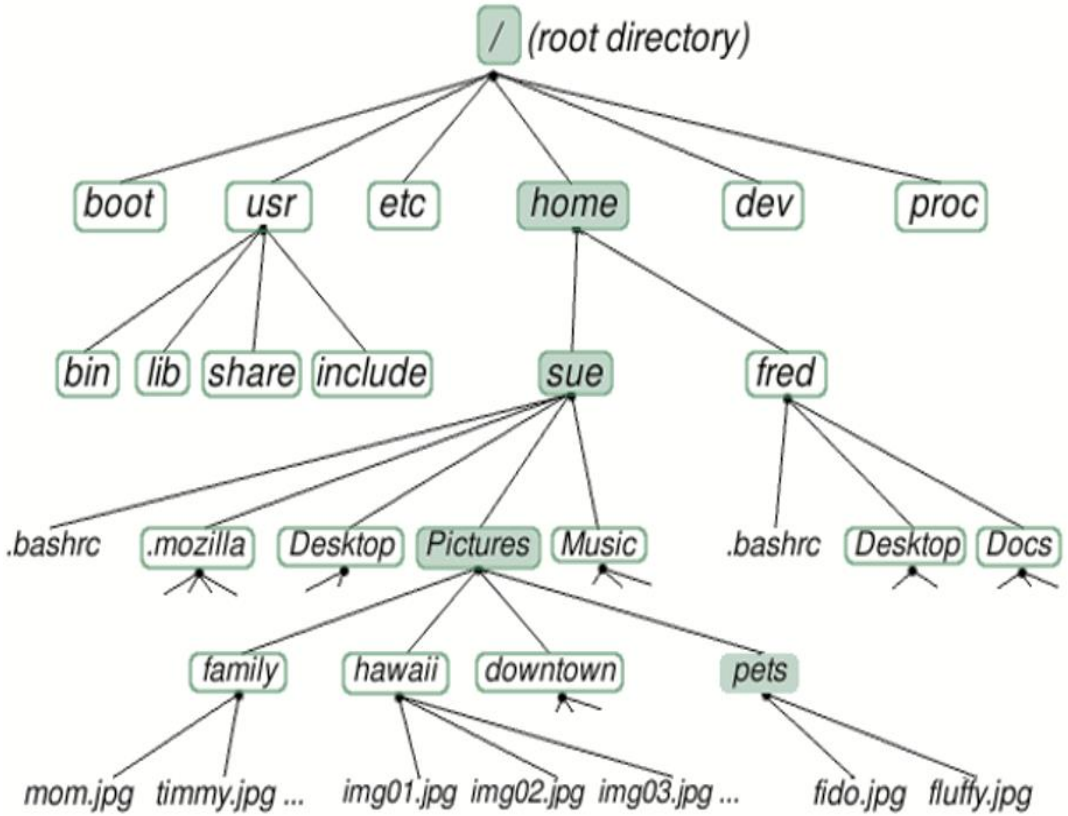
What is a Tree

- A tree is a finite non-empty set of elements.
- It is an abstract model of a hierarchical structure.
- Consists of nodes with a parent-child relation.
- Applications:
  - Organizational data
  - File systems /storing naturally
  - Network routing algorithm.
  - Programming environments
  - Compiler Design/Text processing (syntax analysis & to display the structure of a sentence in a language)
  - Searching Algorithms and sorting
  - Evaluating a mathematical expression.



Why do we need trees ?

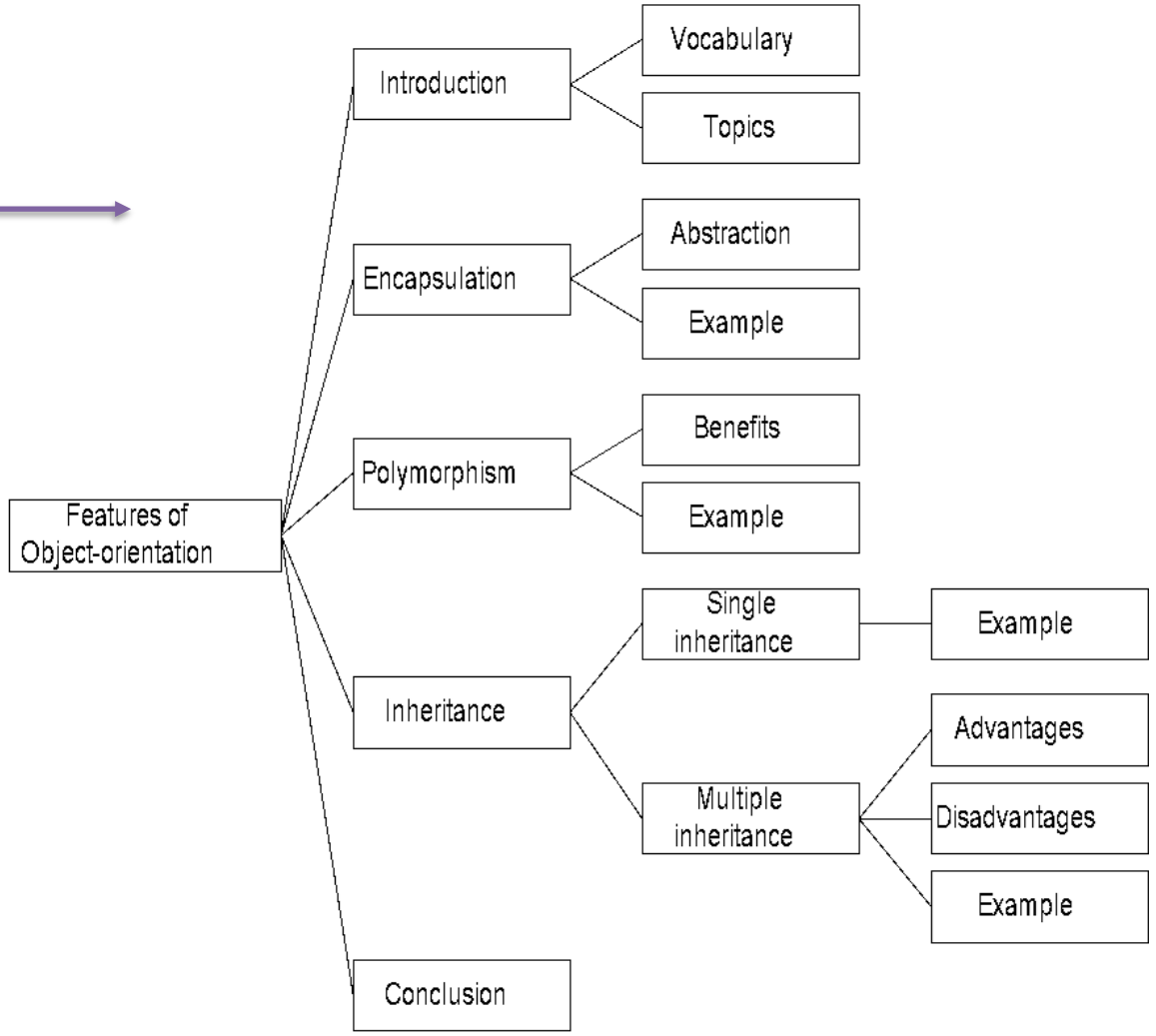
- Lists, Stacks, and Queues are *linear* relationships
- Information often contains hierarchical relationships:
  - File directories or folders
  - Moves in a game
  - Hierarchies in organizations
- Can build a tree to support these relationships and also support fast searching.



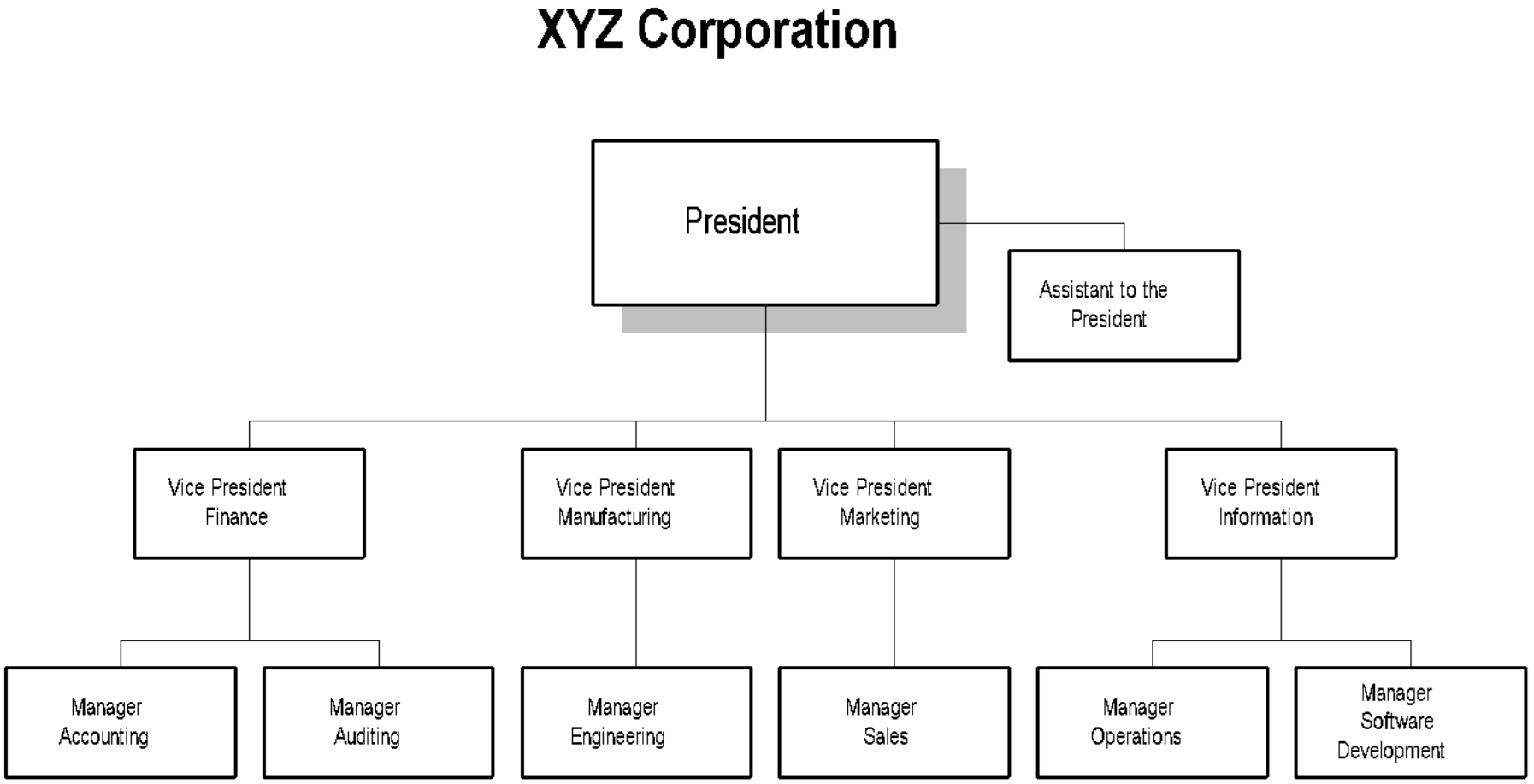
Tree Applications : Outline as Tree

Features of Object-orientation

I. Introduction
    A. Vocabulary
    B. Topics
II. Encapsulation
    A. Abstraction
    B. Example
III. Polymorphism
    A. Benefits
    B. Example
IV. Inheritance
    A. Single inheritance
        1. Example
    B. Multiple inheritance
        1. Advantages
        2. Disadvantages
        3. Example
IV. Conclusion



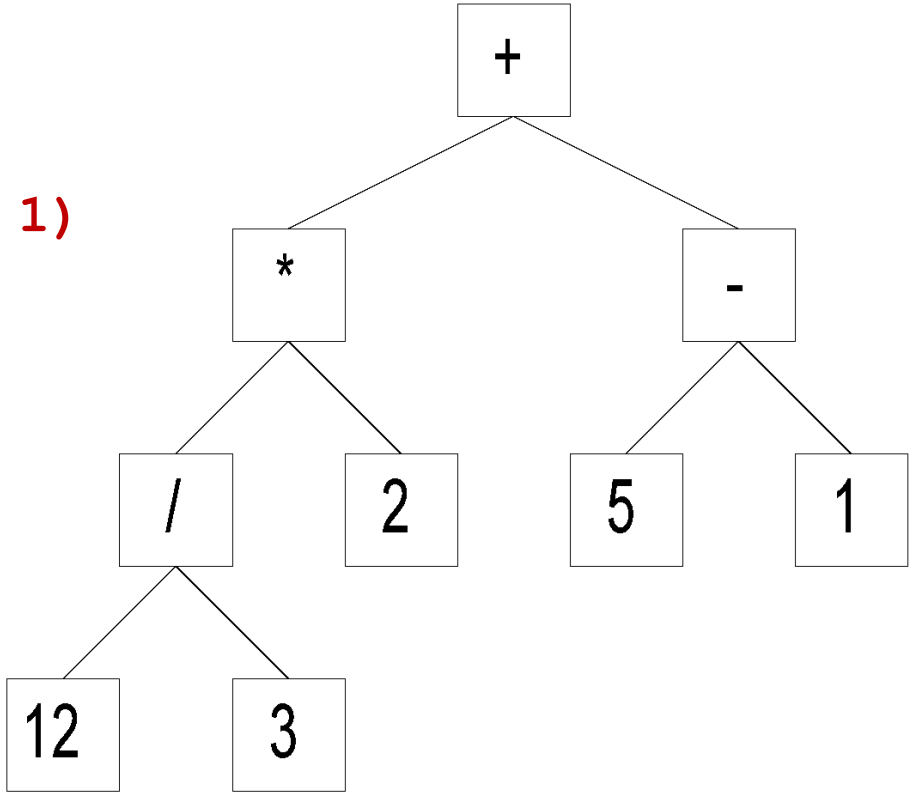
Tree Applications : Organizational Chart



Tree Applications : Expression Tree

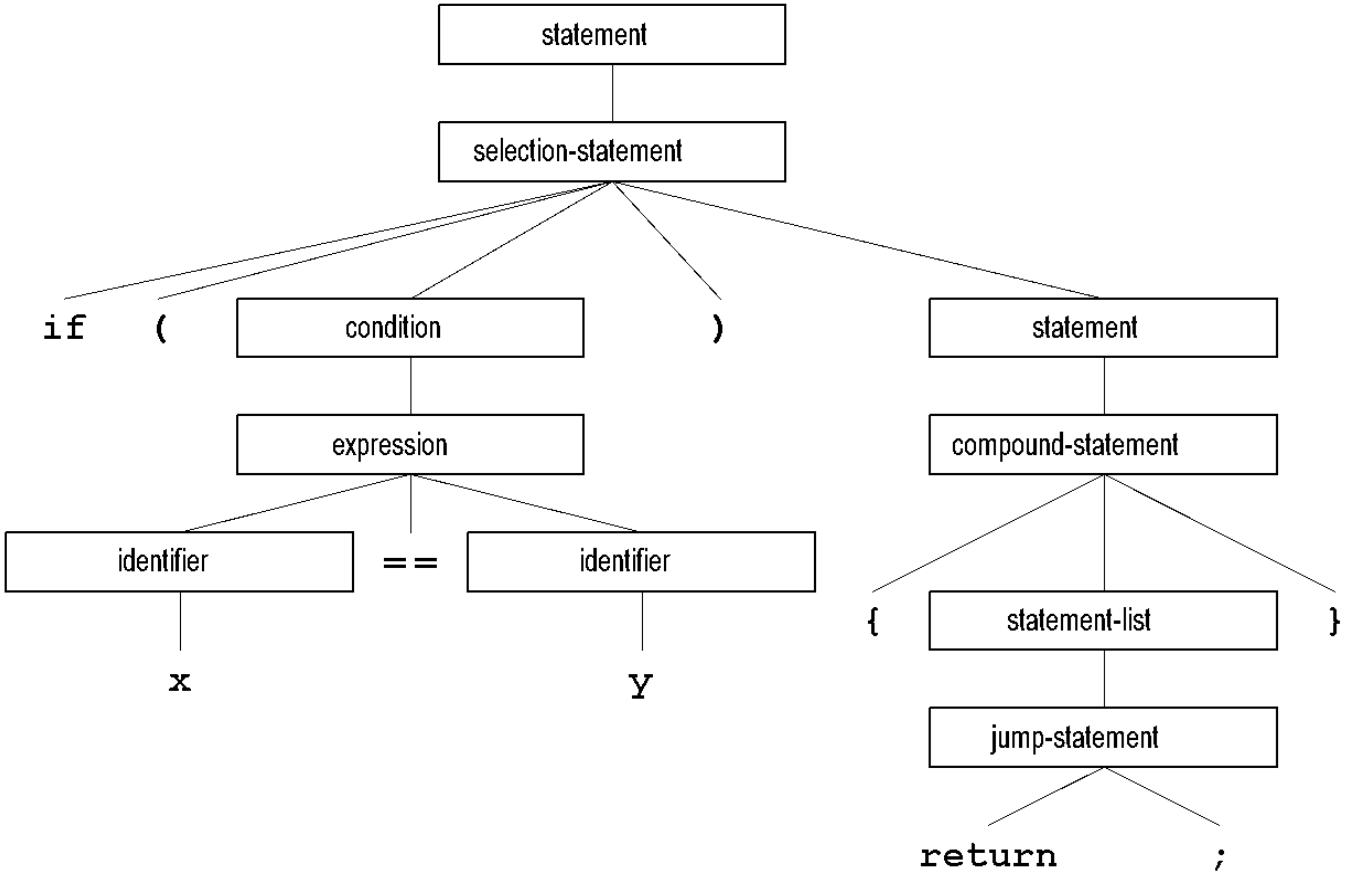
Can represent an arithmetic expression as a tree

```
((12 / 3) * 2) + (5 - 1)
```



Tree Applications : Programs as Trees

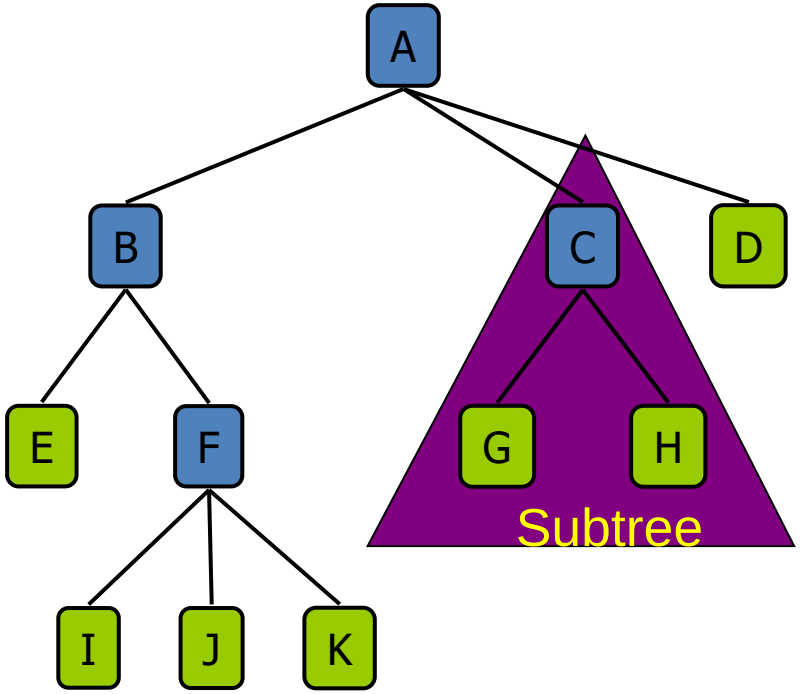
- Many compilers represent programs as trees



Tree Terminology

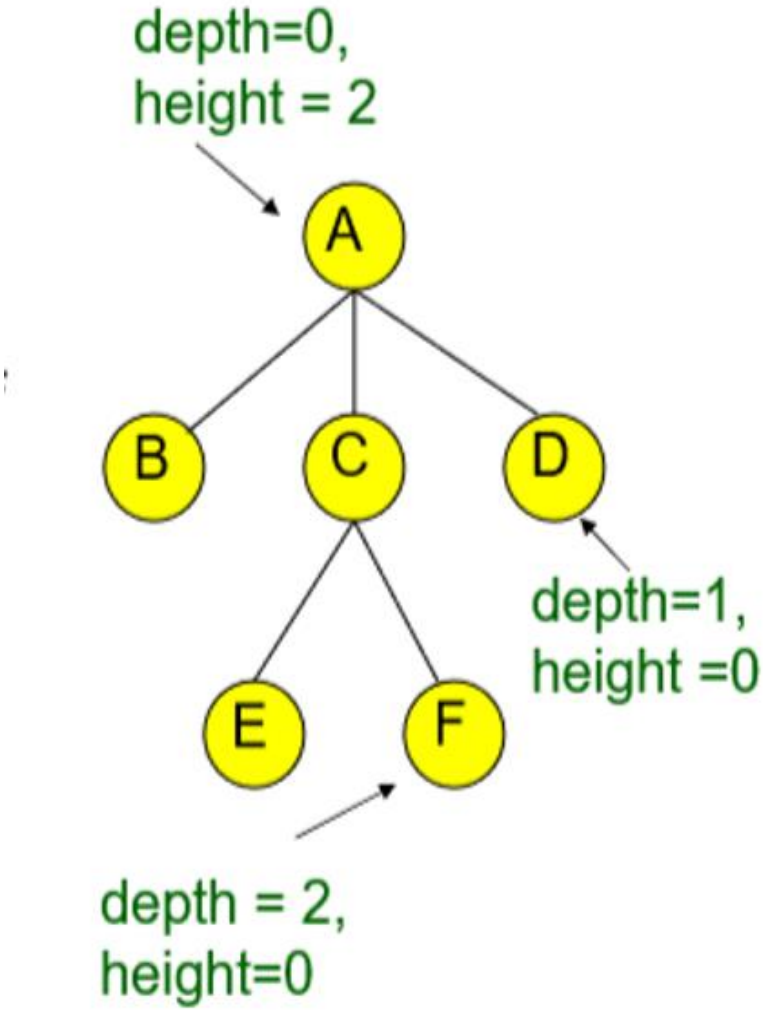
- ✓ **Root**: node without parent (A)
- ✓ **Siblings**: nodes share the same parent (B,C,D)
- ✓ **Internal node**: node with at least one child (A, B, C, F)
- ✓ **External node** (leaf): node without children (E, I, J, K, G, H, D)
- ✓ **Ancestors** of a node: parent, grandparent, great-grandparent, etc. (For E, B and A are ancestors).
- ✓ **Descendant** of a node: child, grandchild, great-grandchild, etc. (For B, E and F,I,J,K are descendants)
- ✓ **Degree** of a node: the number of its children (For A its 3, F its 3, C its 2).
- ✓ **Depth of x** = no of edges in path from root to x. (depth of F is 2)
- ✓ **Height of x**= no of edges in longest path from x to a leaf. (height of B is 2, height of A is 3).

- ✓ **Sub-tree**: tree consisting of a node and its descendants
- ✓ **Level:** set of all nodes at the same depth. Root at 0.
- ✓ **Forest:** set of disjoint trees.

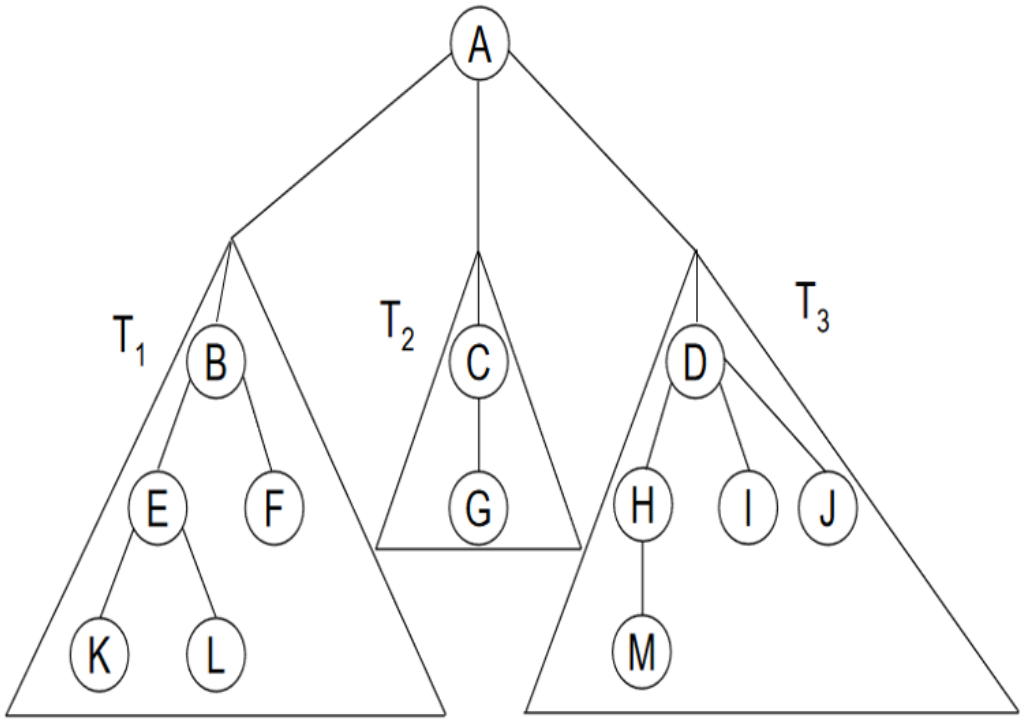


More Tree Jargon

- *Length of a path*= number of edges
- *Depth of a node N* = length of path from root to N
- *Height of node N* = length of longest path from N to a leaf
- *Depth of a tree* = depth of deepest node
- *Height of a tree* = height of root



Tree Terminology Activity



- ✓ *Root ?*
- ✓ *Internal nodes ?*
- ✓ *Leaf's ?*
- ✓ *Ancestors of M ?*
- ✓ *Descendants of E ?*
- ✓ *Degree of A ?*
- ✓ *Height of tree ?*
- ✓ *I is in which level ?*
- ✓ *Depth of H ?*
- ✓ *Height of D ?*

Tree ADT

- We use positions to abstract nodes
- Generic methods:

  integer size(): Return the number of nodes in the tree.

  boolean isEmpty() : Return if tree is empty

  ObjectIterator elements(): Return an iterator of all the elements stored at nodes of the tree.
- Accessor methods:

  position root(): Return the root of the tree

  position parent(p): Return the parent of node p; error if p is root.

  positionIterator children(p): Return an iterator of the children of node p.

Tree ADT

- Query methods:
  - boolean isInternal(v):Test whether node v is internal.
  - boolean isExternal(v):Test whether node v is external
  - boolean isRoot(v):Test whether node v is the root.
- Update methods:
  - swapElements(v, w):Swap the elements stored at the nodes v and w.
  - object replaceElement(v, e):Replace with e and return the element stored at node v

TREE ADT-Depth and Height

- Let v be a node of a tree T.
- The depth of v is the number of ancestors of v, excluding v itself
- If v is the root, then the depth of v is 0.
- Otherwise, the depth of v is one plus the depth of the parent of v.

Algorithm depth(T, v):

```
if T. isRoot(v) then
    return 0
else
    return 1 + depth (T, T. parent(v))
```

- The running time of algorithm depth (T, v) is $O(1 + d_v)$, where d_v denotes the depth of the node v in the tree T.
- In the worst case, the depth algorithm runs in O (n) time, where n is the total number of nodes in the tree T

TREE ADT- Depth and Height

- The ***height of a tree*** T is equal to the maximum depth of an external node of T.
- If v is an external node, then the height o f v is 0.
- Otherwise, the height of v is one plus the maximum height of a child of v.

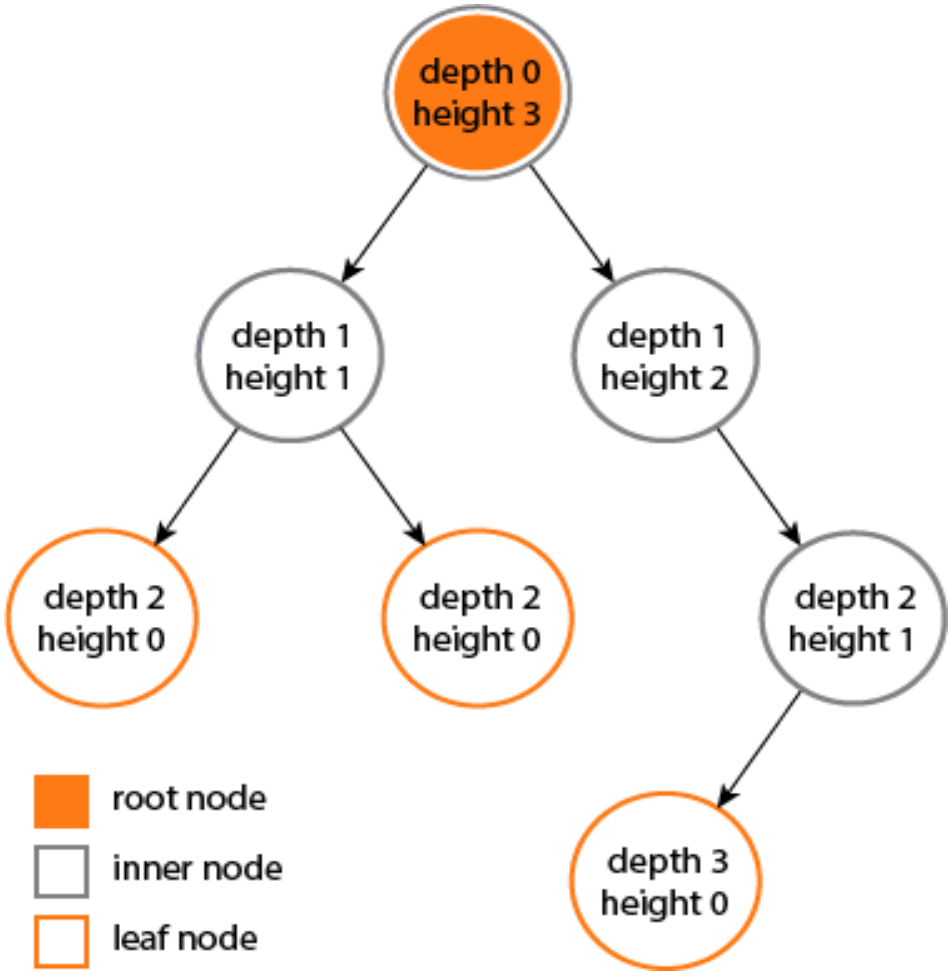
Algorithm height(T, v) :

```
if T. isExternal (v) then
return 0
else
h = 0
for each w ∈ T.children (v) do
h = max(h, height(T, w))
return 1 + h
```

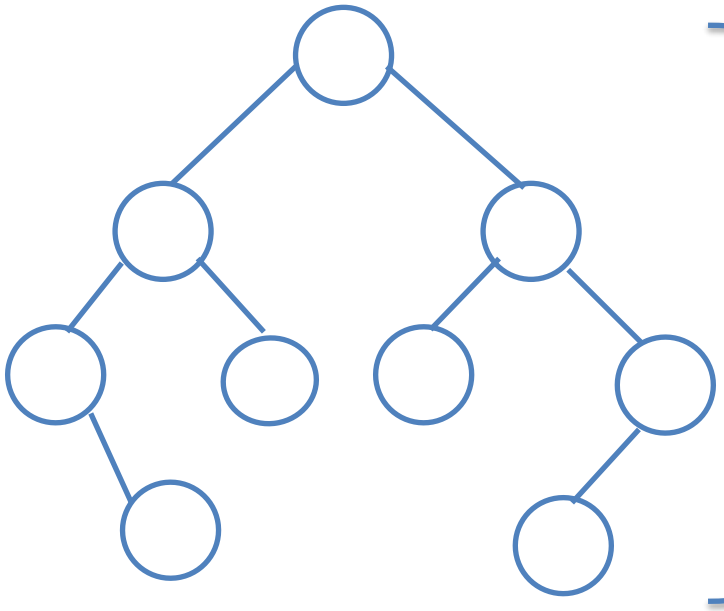
The height of a tree T is obtained by calling height(T,T.root())

Complexity of this algorithm is also O(n), Refer Chapter 2, T1. (This involves Amortized analysis).

TREE ADT-Dept h and Height



Binary Tree

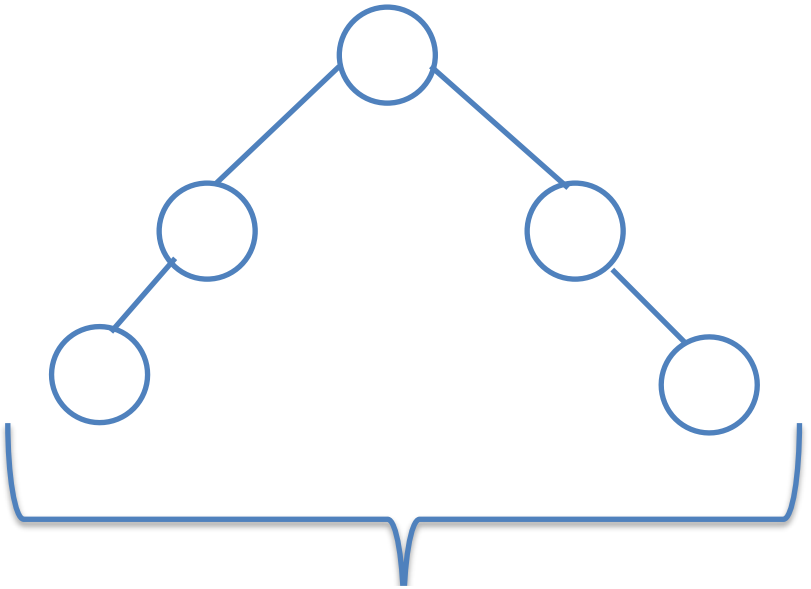


Each node can have at most 2 – children

A binary tree is a tree with the following properties:

- Each internal node has **at most two** children.
- The children of a node are called as left child and right child.

This is also binary tree

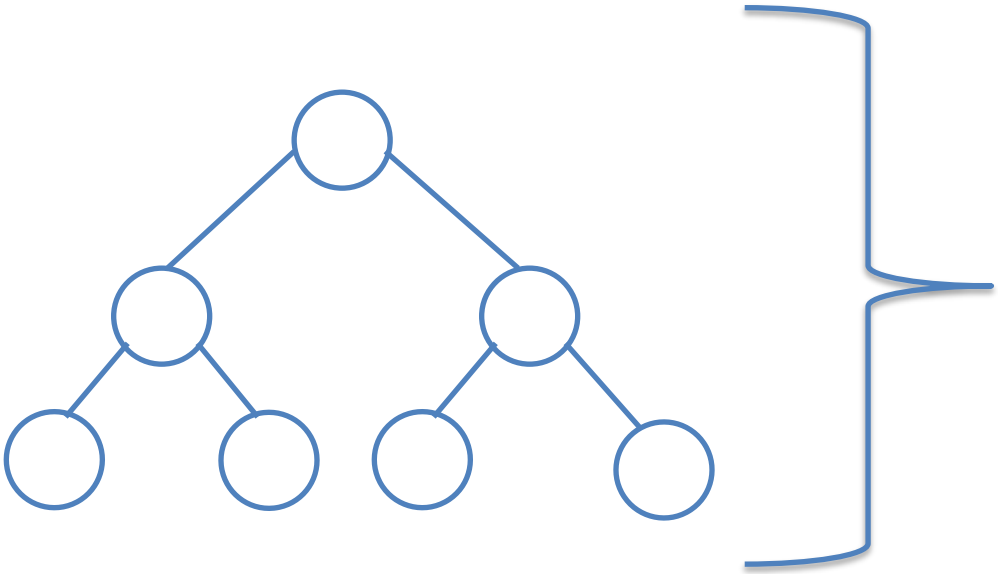


What about this ?

Strict / Proper/ Full Binary tree

Proper Binary Tree

- Each internal node has **exactly** two children

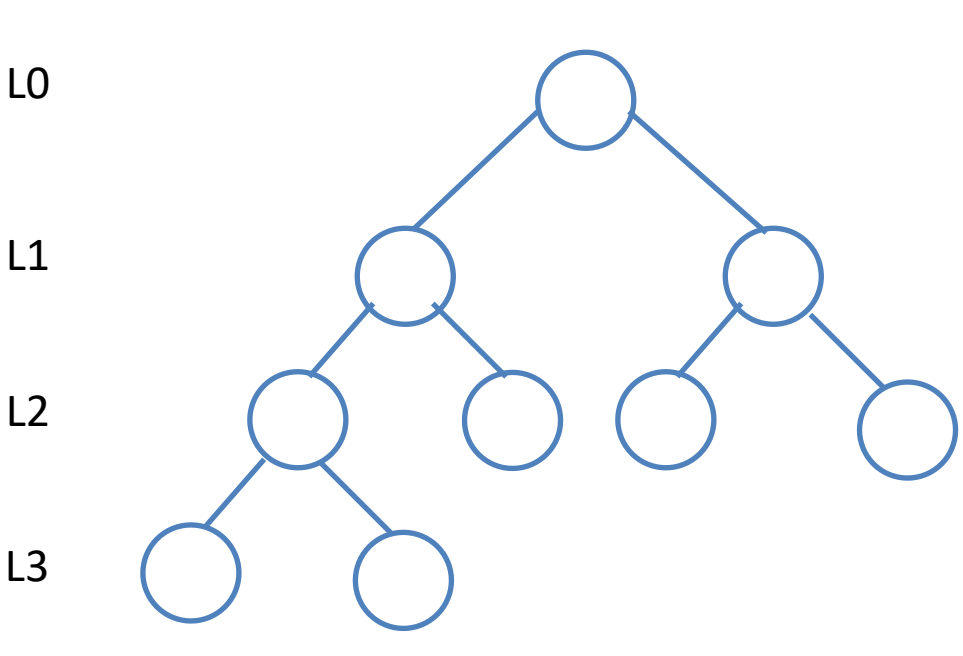


Each node can have either 2 or 0 children.

Complete Binary tree

Complete binary tree

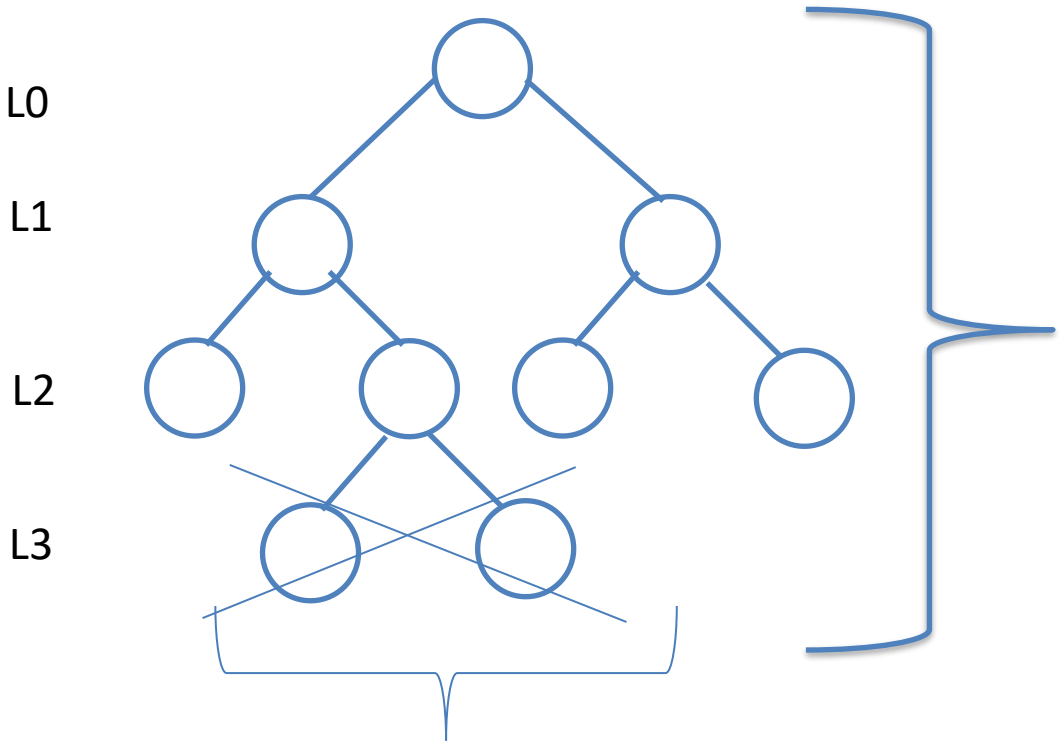
- A binary tree in which every level, except the lowermost, is completely (max) filled
- At the lowermost level nodes must be filled from left to right



All levels except possibly the last are completely filled and all nodes are as left as possible.

Max no of nodes at level $i = 2^i$

Complete Binary tree

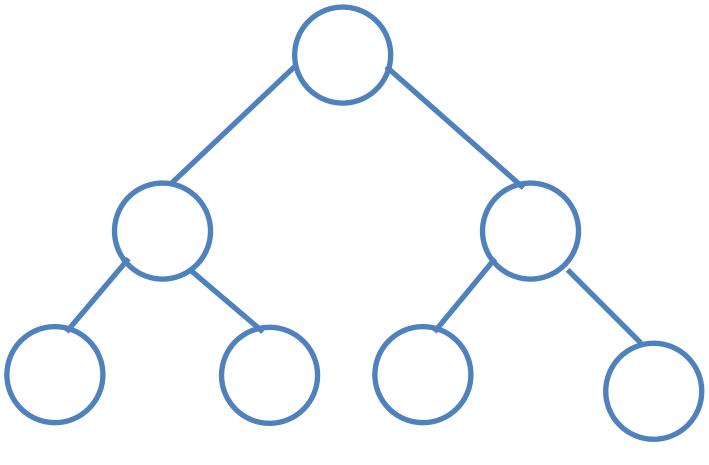


All levels except possibly the last are completely filled and all nodes are not as left as possible. So this is not a complete Binary tree

This is not the left as possible

Perfect binary tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



In perfect binary all levels will be completely filled.

Proof of complete binary can be found in the introduction to algorithms textbook ….

Maximum no of nodes in a binary tree with height h

$= 2^0 + 2^1 + \ldots\ldots + 2^h$

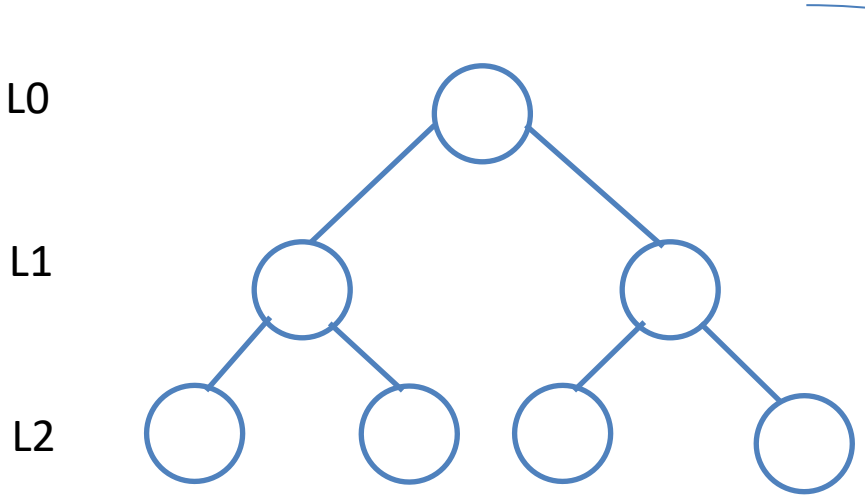
$= 2^{h+1} - 1$

$= 2^{(\text{no of levels})} - 1$

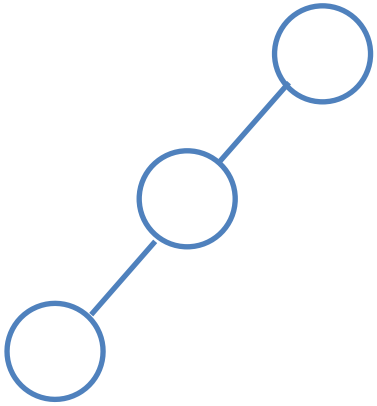
$n = 2^{h+1} - 1$

- Now you can find the high of a tree, with n
  - $n = 2^{h+1} - 1$
  - $2^{h+1} = n+1$
  - $h = \log_2(n+1) - 1$
- Height of complete binary tree h = floor ($\log_2 n$)

Min and max height of a binary tree

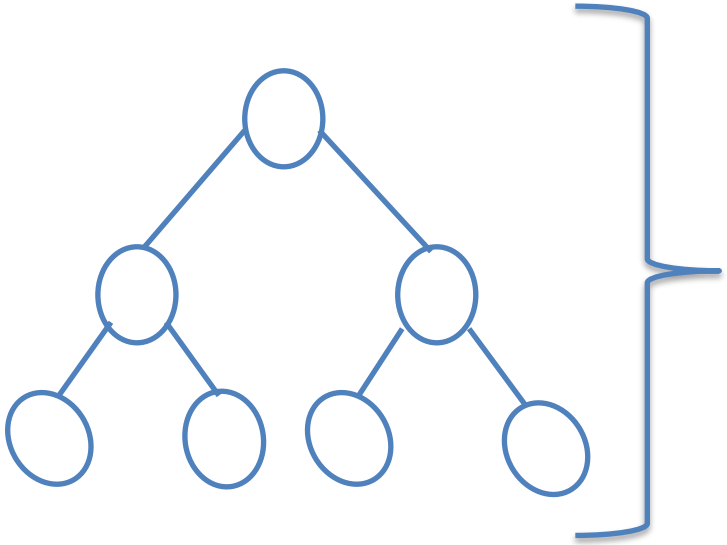


Min-height of n-nodes = floor ($\log_2 n$)



Max-height of n-nodes = n-1

Balanced binary tree



Difference between height of left & right subtree for every node is not more than k (mostly 1).

- Height → no of edges in longest path from root to a leaf.
- Height of an empty tree =-1
- Height of tree with one node = 0

Binary tree implementation

We can implement binary tree using

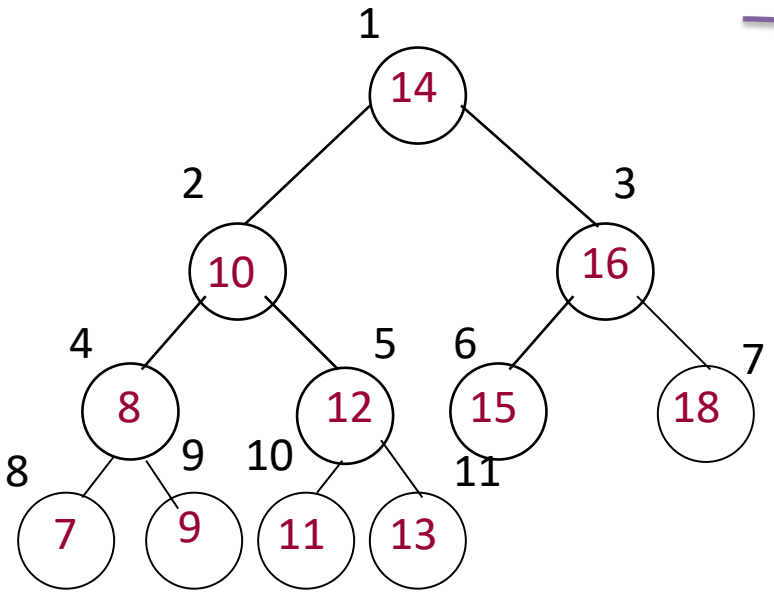
- Linked List [Dynamically created nodes]
- Arrays [this will be efficient only for complete binary tree]

Complete Binary Trees: Array Representation

Complete Binary Trees can be represented in memory with the use of an array A so that all nodes can be accessed in O(1) time:

- Root will be the first position in A
- Label nodes sequentially top-to-bottom and left-to-right
- Left child of A[i] is at position A[2i]
- Right child of A[i] is at position A[2i + 1]
- Parent of A[i] is at A[i/2]

Complete Binary Trees: Array Representation

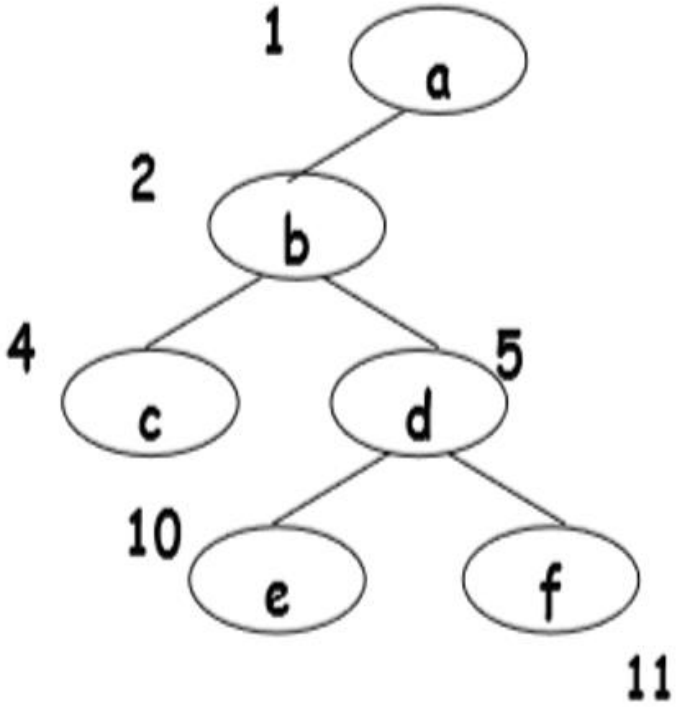


- Left child of A[i] is at position A[2i]
- Right child of A[i] is at position A[2i + 1]
- Parent of i is at position floor(i/2)

14	10	16	8	12	15	18	7	9	11	13
1	2	3	4	5	6	7	8	9	10	11

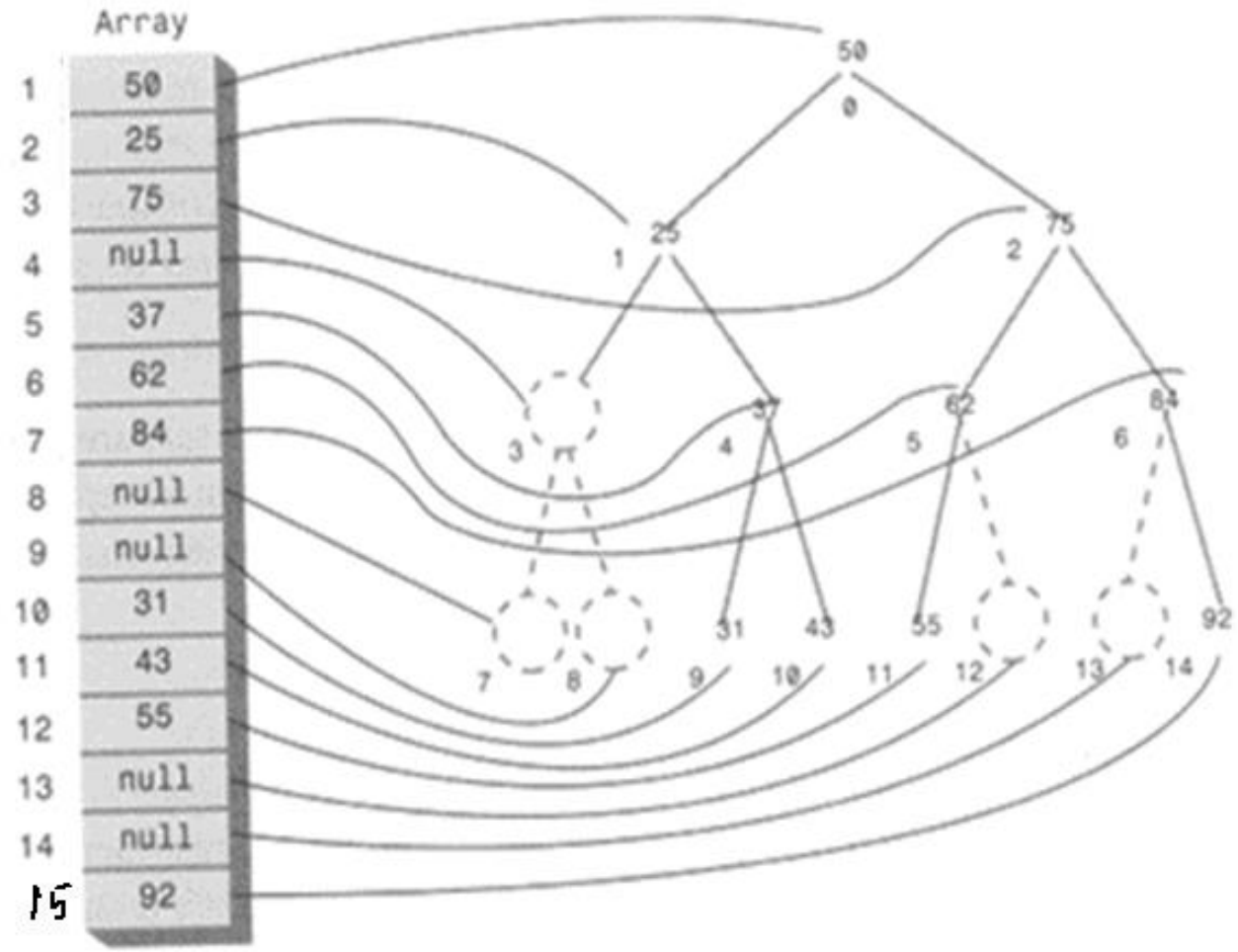
Binary Trees: Array Representation

Sequential representation of a tree with depth d will require an array with approx. $(2^{d+1} - 1)$ elements



1	2	3	4	5	6	7	8	9	10	11
a	b		c	d					e	f

Binary Trees: Array Representation



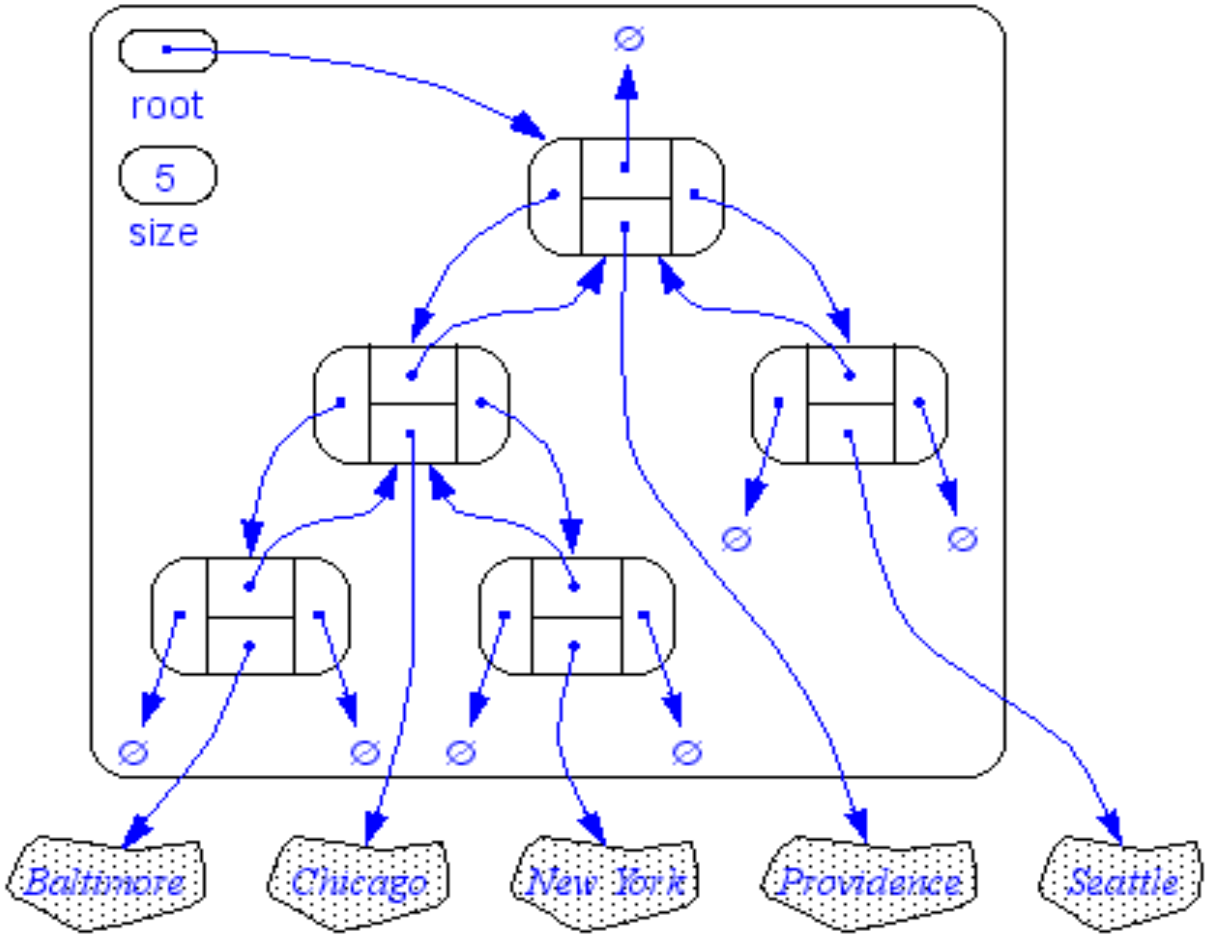
Binary Trees: Linked List Representation

A **Binary tree** is a linked data structure. Each node contains data ,and pointers left, right and p.

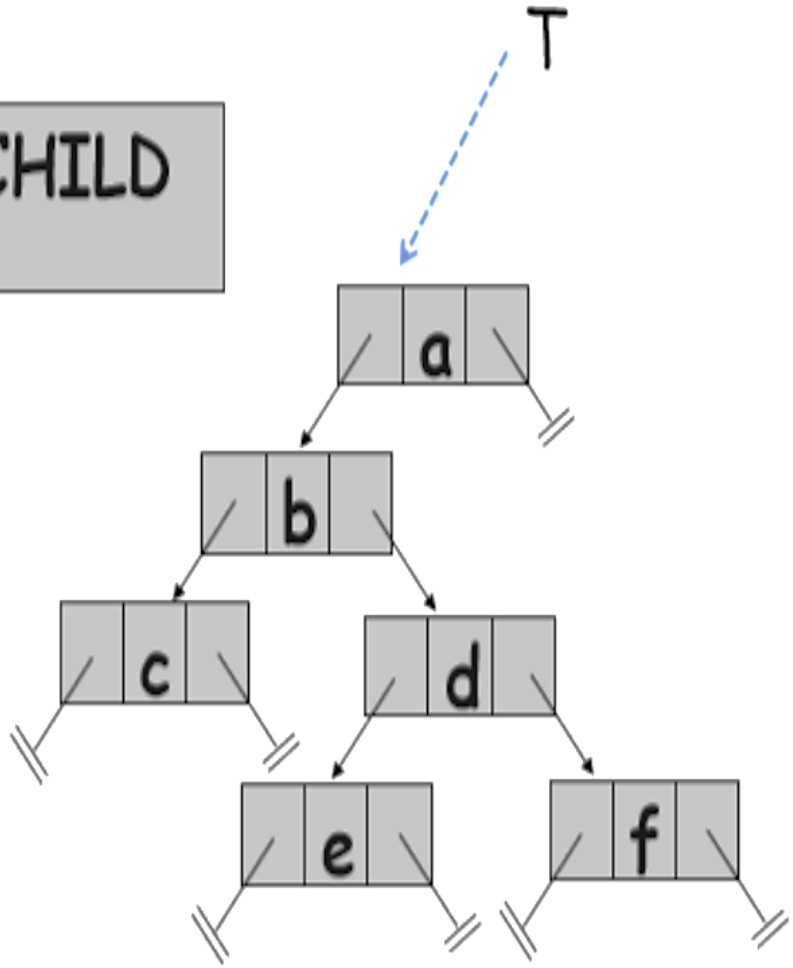
- ***Left* points to the left child of the node.**
- ***Right* points to the right child of the node.**
- ***p* points to the parent of the node.**

- If a child is missing, the pointer is NIL.
- If a parent is missing, p is NIL.
- The **root** of the tree is the only node for which p is NIL.
- Nodes for which both left and right are NIL are **leaves.**

Binary Trees: Linked List Representation

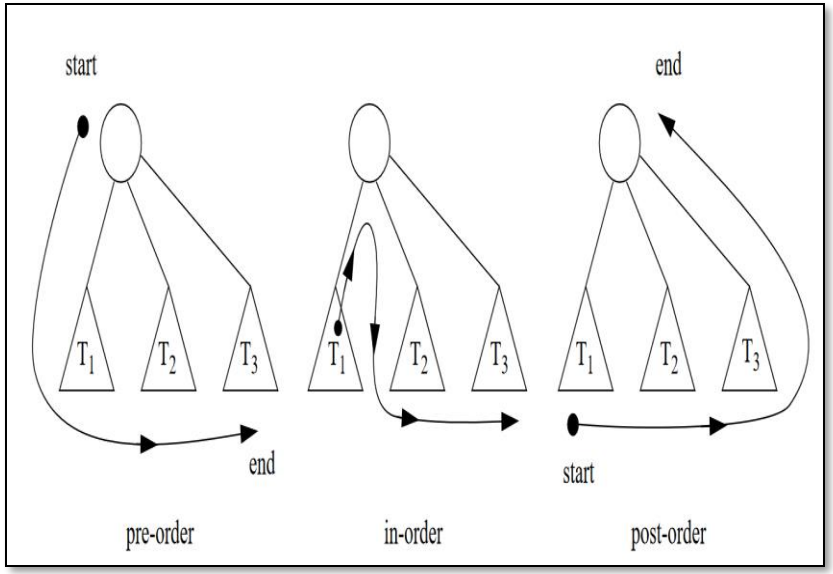


Binary Trees: Linked Representation



Binary Tree Traversals

- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once.
- Tree traversals are naturally recursive.
- Since a binary tree has three parts, the ways to traverse the binary tree are:
  - ✓root, left, right : Preorder
  - ✓left, root, right: Inorder
  - ✓left, right, root: Postorder



Binary Tree Traversals

Three ways of traversing the binary tree T with root R:

Preorder

- Process the root R
- Traverse the left sub-tree of R in preorder
- Traverse the right sub-tree of R in preorder

a.k.a node-left-right (**NLR**)

In-order

- Traverse the left sub-tree of R in in-order
- Process the root R
- Traverse the right sub-tree of R in in-order

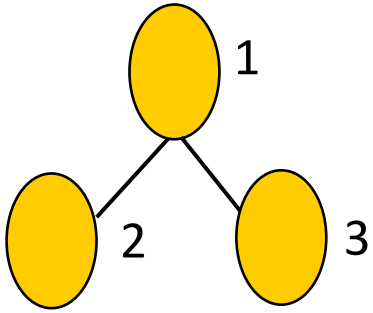
a.k.a left-node-right (**LNR**)

Post-order

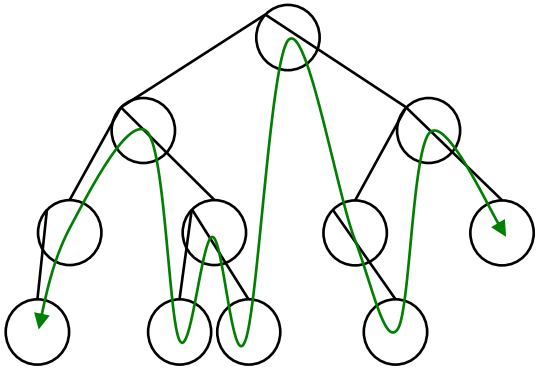
- Traverse the left sub-tree of R in post-order
- Traverse the right sub-tree of R in post-order
- Process the root R

a.k.a left-right-node (**LRN**)

Binary Tree Traversals

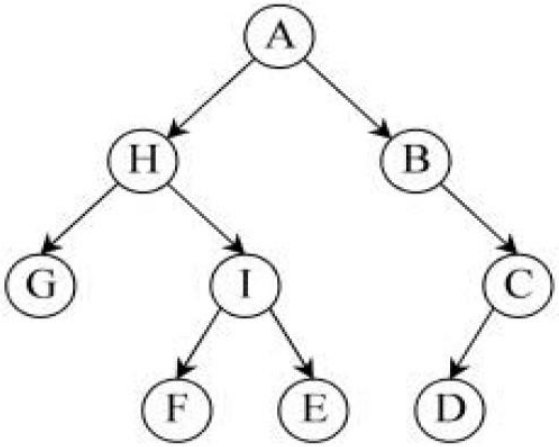


- Preorder(Root(N),left (L),right (R)): 1 2 3
- Inorder(left(L),root(N),right(R)): 2 1 3
- Postorder(left(L),right(R),root(N)): 2 3 1

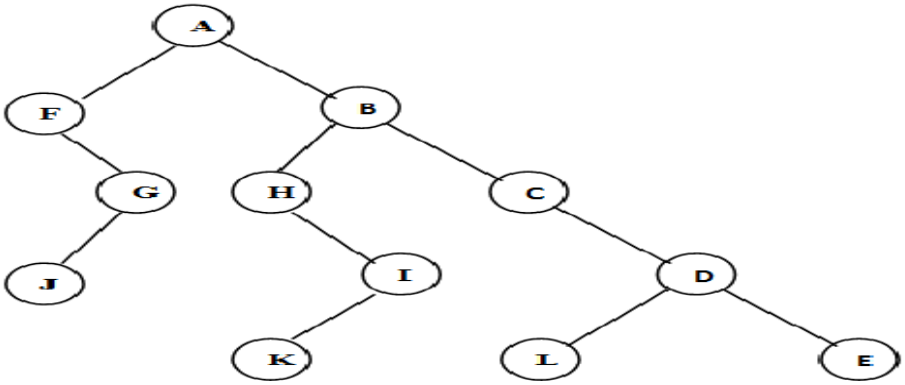


Exercises for CS #4

1) Perform the 3 traversals on the below:



2) Represent below in array and Linked list:

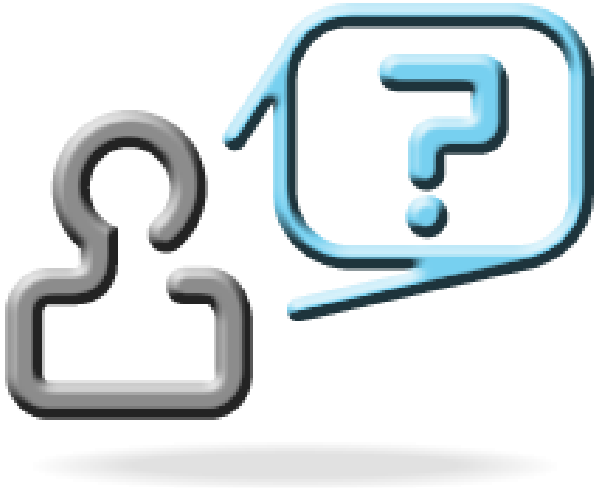


Summary for CS #4

1) Recap of CS#3

2) Introduction to Trees
   - Tree terminologies
   - Binary tree and types
   - Representation of tree
     - Using array
     - Using linked list

3) Q&A



See you in the next class to explore heaps!

Thank You for your time & attention !

Contact : parthasarathypd@wilp.bits-pilani.ac.in