Data Structure and Algorithms

Affefah Qureshi

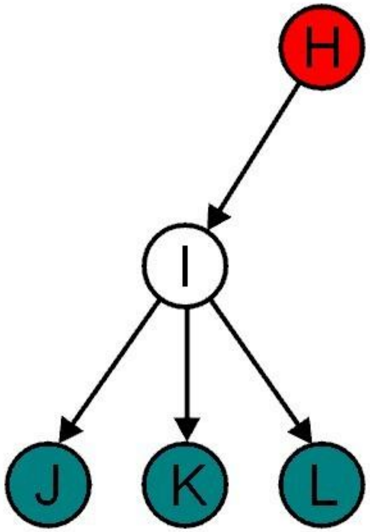
Department of Computer Science

Iqra University, Islamabad Campus.

Terminology: Parent Child Relations

- All nodes have zero or more child nodes or children
  - I has three children: J, K and L

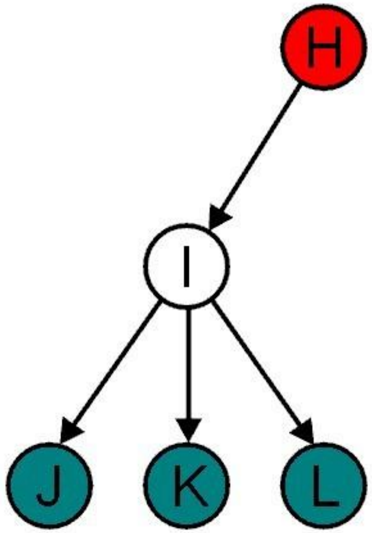
- For all nodes other than the root node, there is
  - H is the parent I



Terminology: Degree

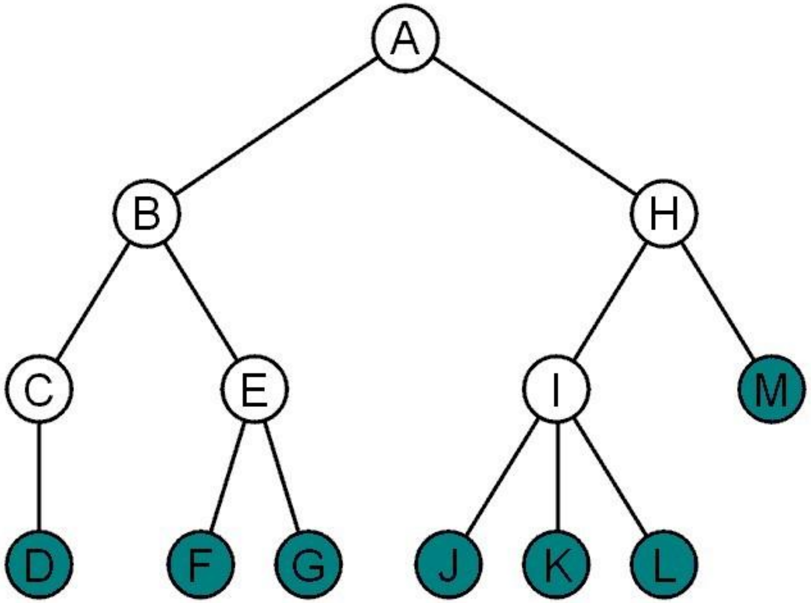
- The degree of a node is defined as the number of its children
  - `deg(I) = 3`

- Nodes with the same parent are siblings
  - J, K, and L are siblings



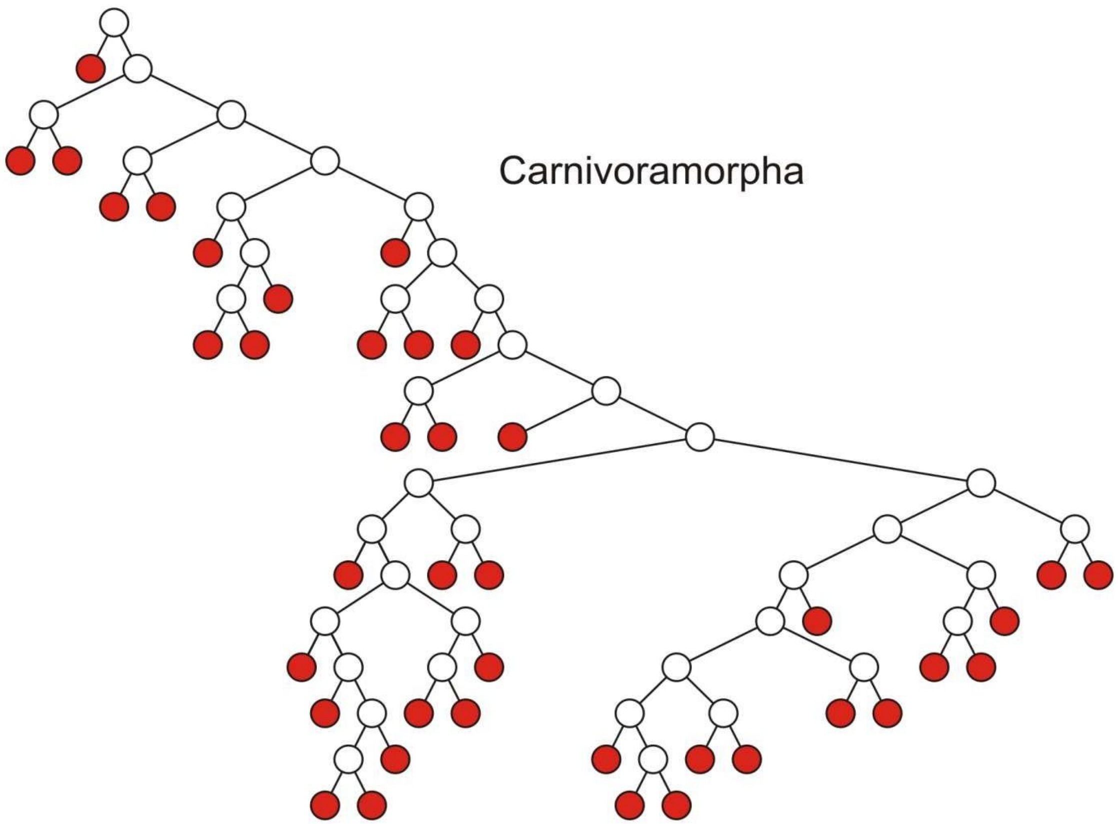
Terminology: Leaf And Internal Nodes

- Nodes with degree zero are also called leaf nodes
- All other nodes are said to be internal nodes, that is, they are internal to the tree



Terminology: Leaf Nodes Examples

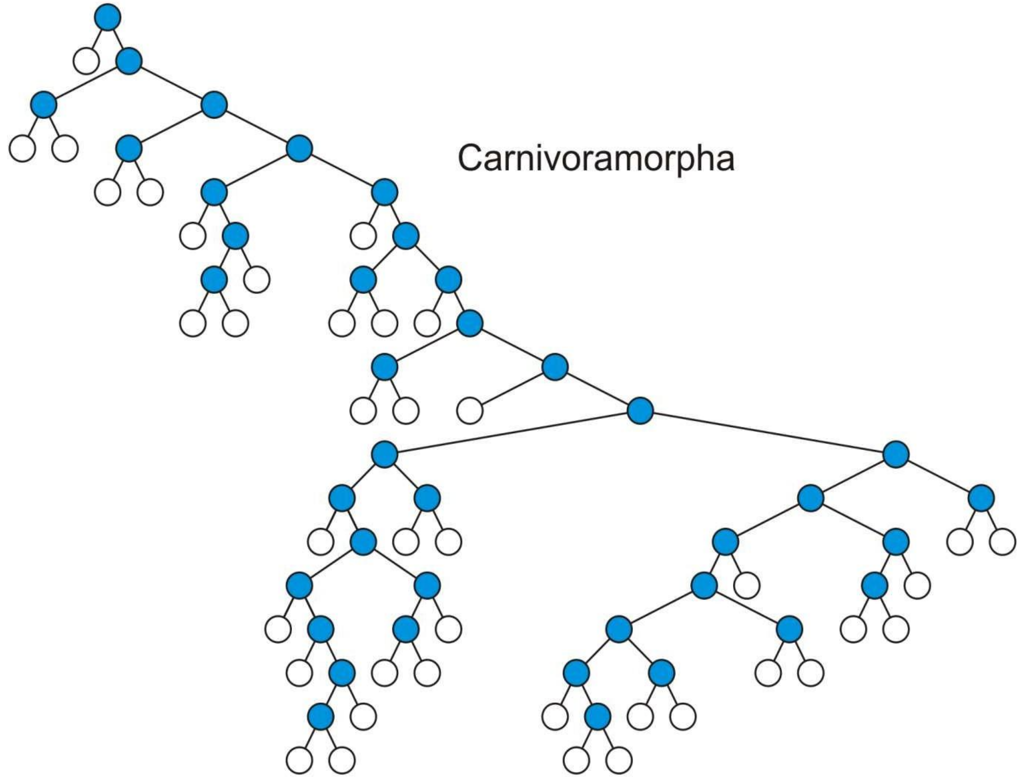
- Leaf nodes



Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Terminology: Internal Nodes Example

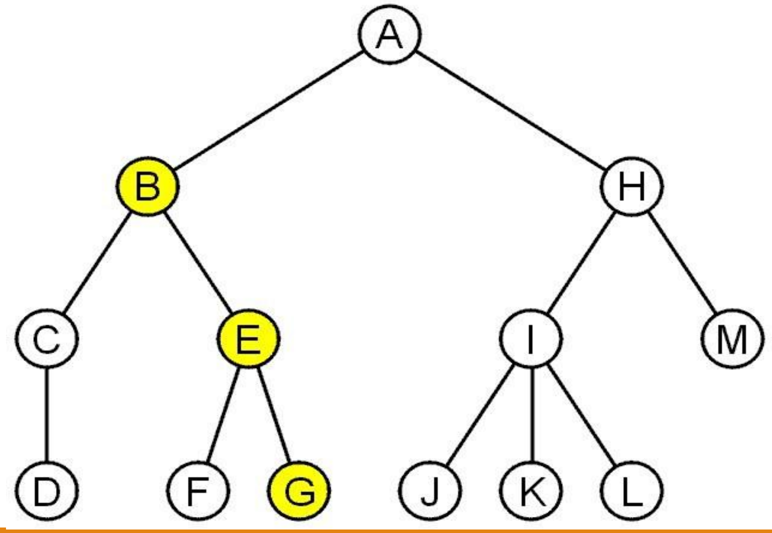
- Internal nodes



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

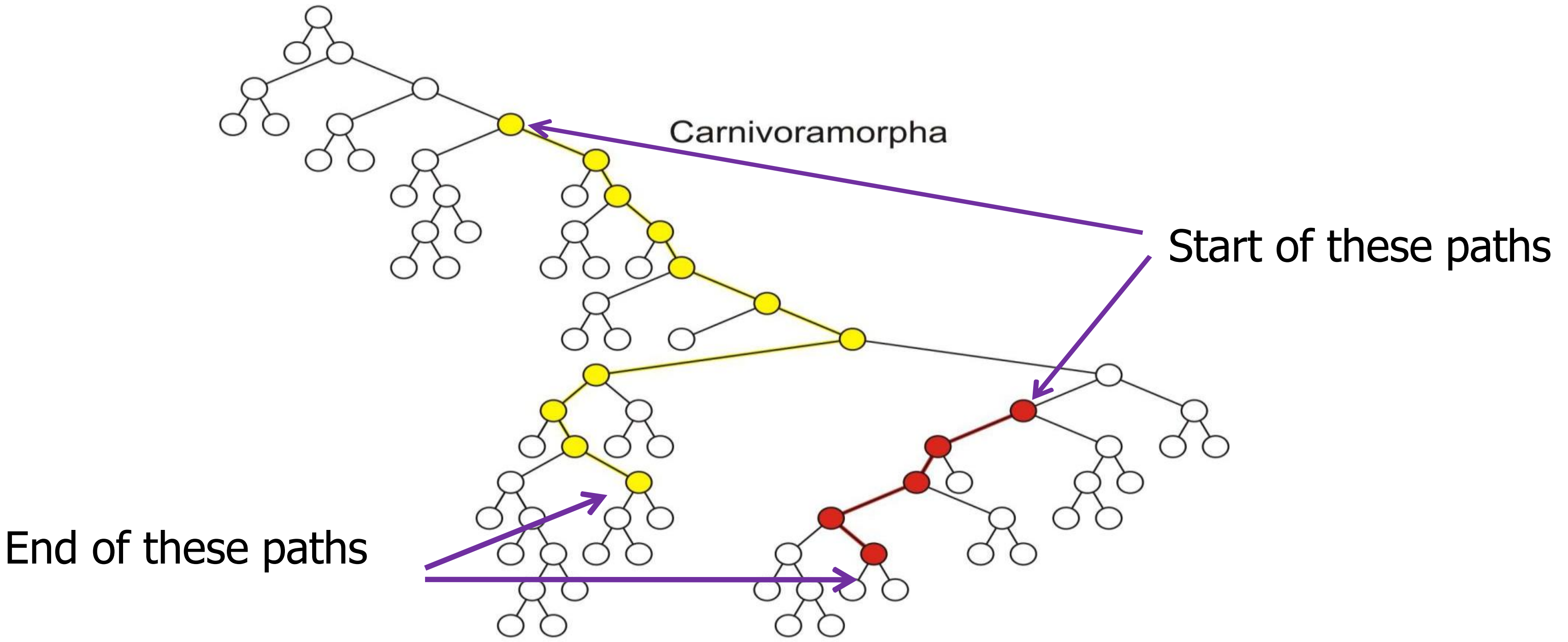
Terminology: Path

- A path is a sequence of nodes $(a_0, a_1, ..., a_n)$
  - Where $a_k + 1$ is a child of a_k is
- The length of this path is: `n = |nodes in the path| - 1`
  - For example, the path (B, E, G) has length 2



Terminology: Path Example

- Paths of length 10 (11 nodes) and 4 (5 nodes)

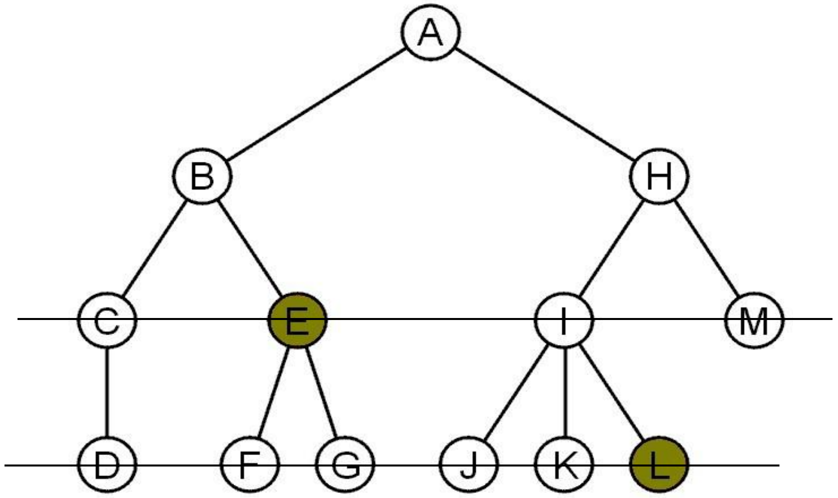


Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Terminology: Depth (or Level)

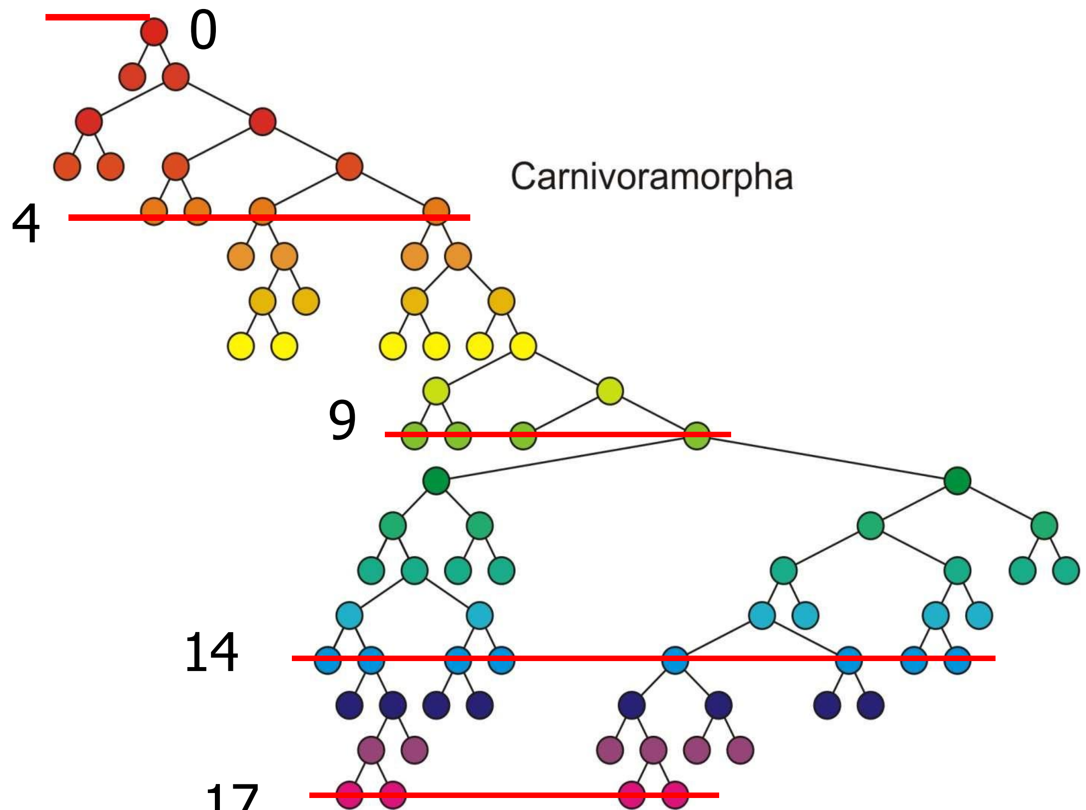
- For each node in a tree, there exists a unique path from the root node to that node

- The length of this path is the depth of the node, e.g.,
  - E has depth 2
  - L has depth 3



Terminology: Depth Example

- Nodes of depth up to 17



Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology: Height

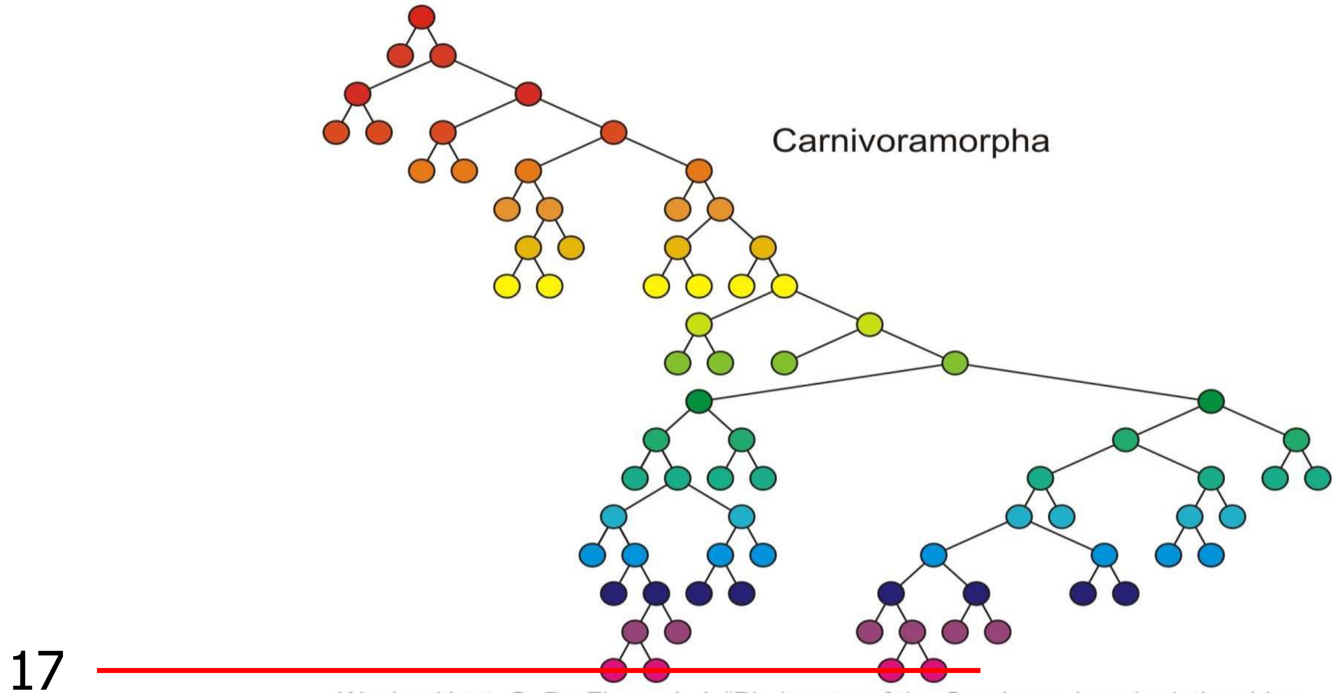
- The height of a tree is defined as the maximum depth of any node within the tree

- The height of a tree with one node is 0
  - Just the root node

- For convenience, we define the height of the empty tree to be –1

Terminology: Height Example

- Height of this tree is 17



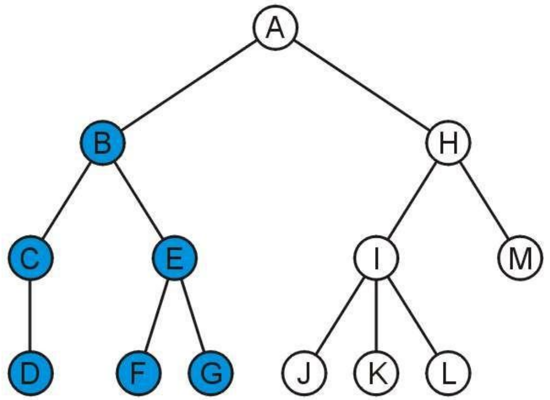
Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Terminology: Ancestors And Descendants

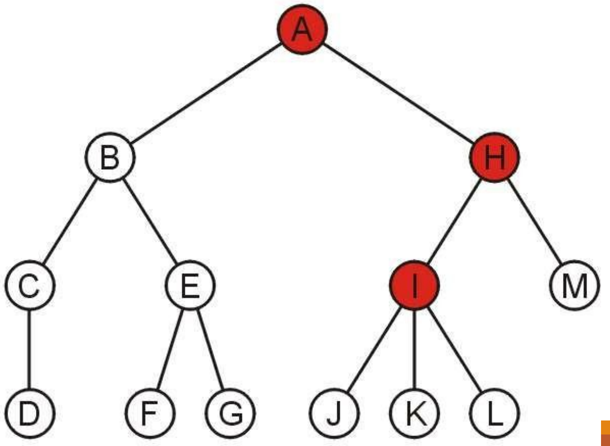
- If a path exists from node a to node b
  - a is an ancestor of b
  - b is a descendent of a
- Thus, a node is both an ancestor and a descendant of itself
  - We can add the adjective strict to exclude equality
  - a is a strict descendent of b if a is a descendant of b but a ≠ b
- The root node is an ancestor of all nodes

Terminology: Ancestors And Descendants Example

- The descendants of node B are C, D, E, F, and G

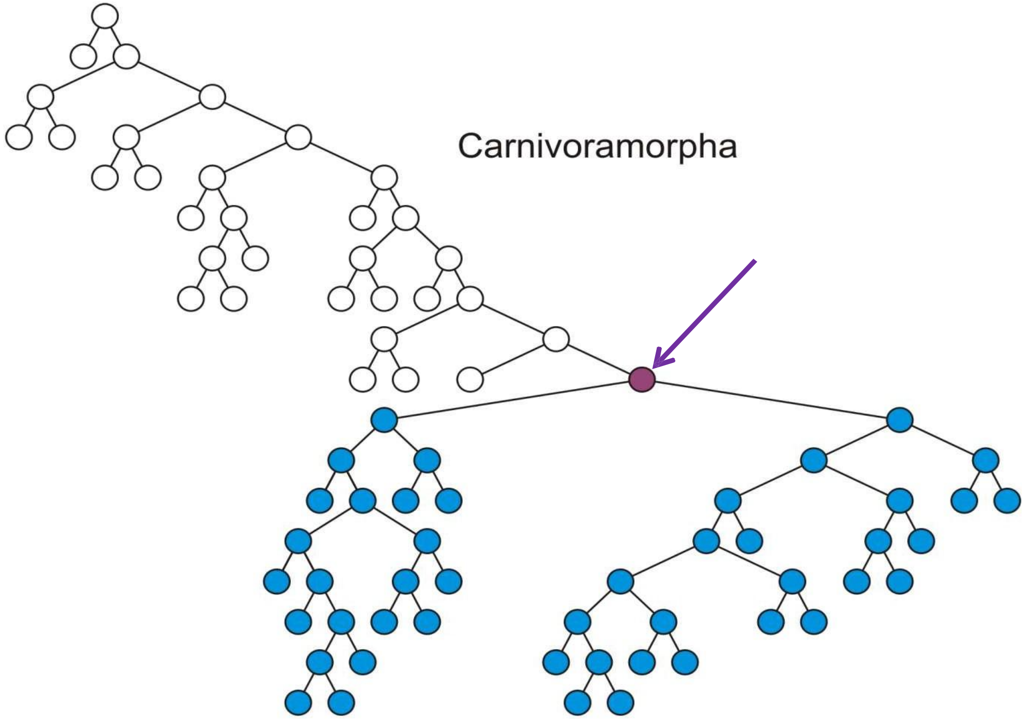


- The ancestors of node I are H and A



Terminology: Descendants Example

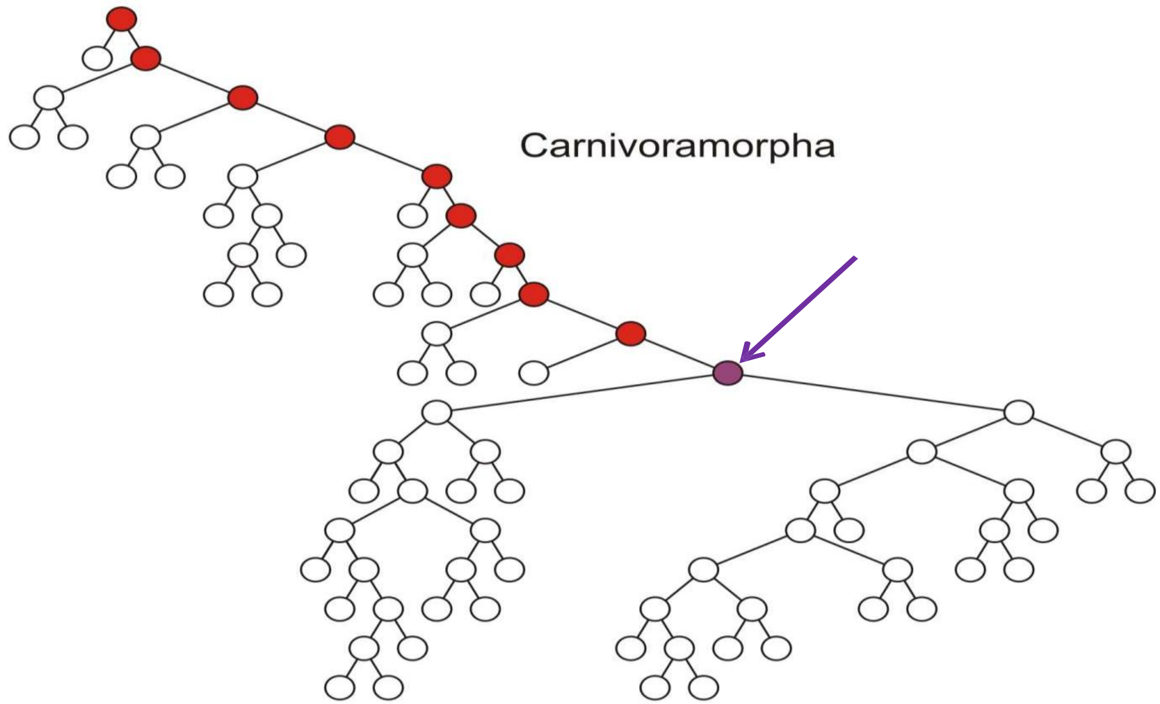
- All descendants (including itself) of the indicated node



Wesley-Hunt, G. D.; Flynn, J. J. "Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of 'Miacoidea'

Terminology: Ancestors Example

- All ancestors (including itself) of the indicated node



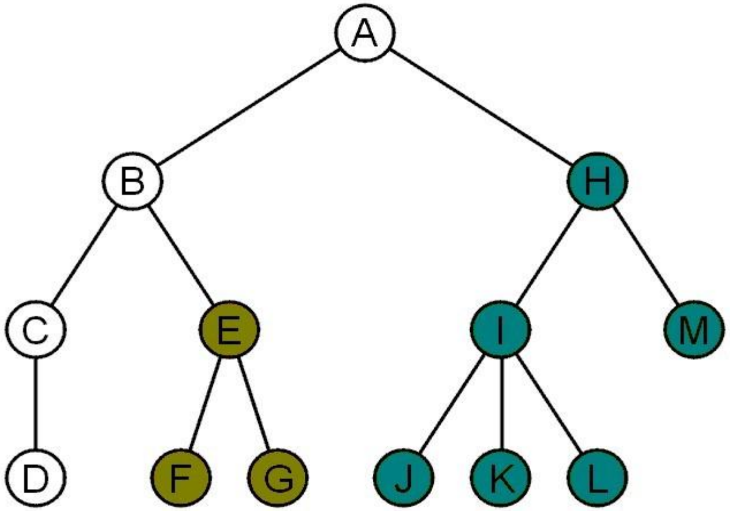
Wesley-Hunt, G. D.; Flynn, J. J. “Phylogeny of the Carnivora: basal relationships among the Carnivoramorphans, and assessment of the position of ‘Miacoidea’

Terminology: Subtree

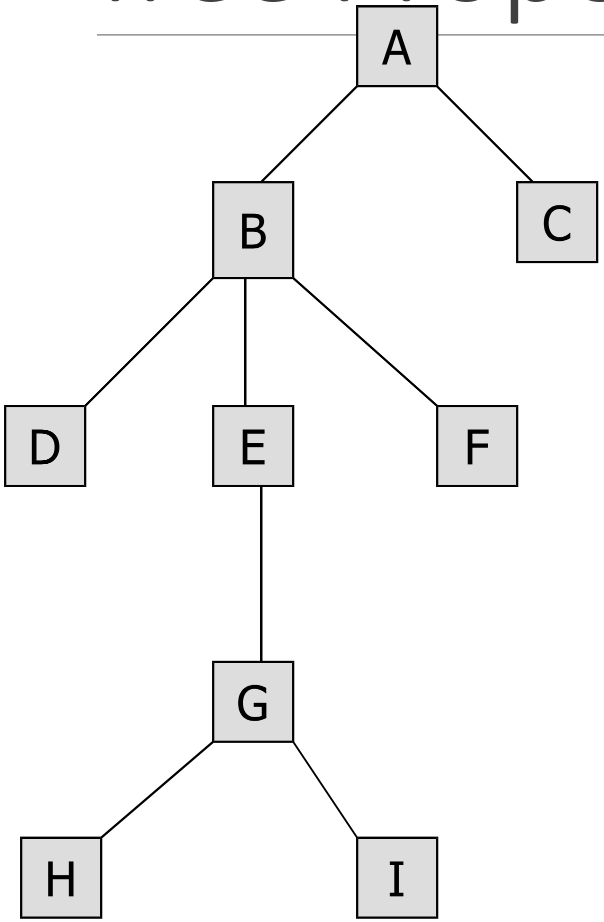
- Another approach to a tree is to define the tree recursively
  - A degree-0 node is a tree

- A node with degree n is a tree if it has n children
  - All of its children are disjoint trees (i.e., with no intersecting nodes)

- Given any node a within a tree with root r, the collection of a and all of its descendants is said to be a subtree of the tree with root a



Tree Properties



Property	Value
Number of nodes	
Height	
Root Node	
Leaves	
Ancestors of H	
Descendants of B	
Siblings of E	
Left subtree	

Example: HTML

- HTML document has a tree structure

```
<html>
   <head>
      <title>Hello World!</title>
   </head>
   <body>
      <h1>This is a <u>Heading</u></h1>

      <p>This is a paragraph with some
      <u>underlined</u> text.</p>
   </body>
</html>
```

Example: HTML

- HTML document has a tree structure

```
<html>
   <head>
      <title>Hello World!</title>
   </head>
   <body>
      <h1>This is a <u>Heading</u></h1>

      <p>This is a paragraph with some
      <u>underlined</u> text.</p>
   </body>
</html>
```

title

heading

body of page

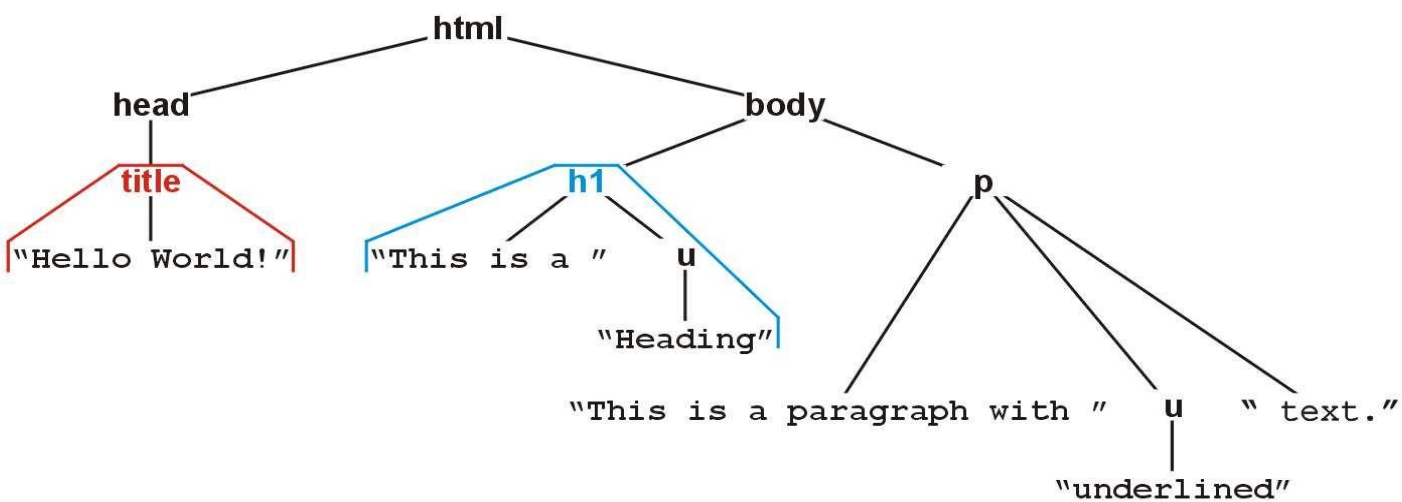
underlining

paragraph

Example: HTML

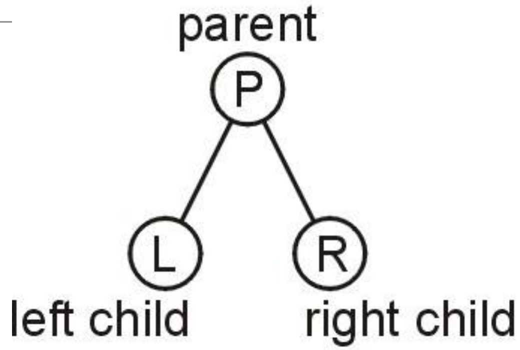
- The nested tags define a tree rooted at the HTML tag

```
<html>
   <head>
      <title>Hello World!</title>
   </head>
   <body>
      <h1>This is a <u>Heading</u></h1>
      <p>This is a paragraph with some
      <u>underlined</u> text.</p>
   </body>
</html>
```

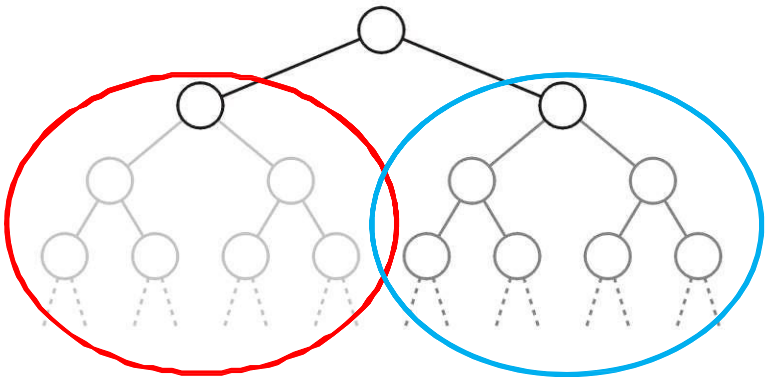


Binary Tree

- In a binary tree each node has at most two
  - Allows to label the children as left and right

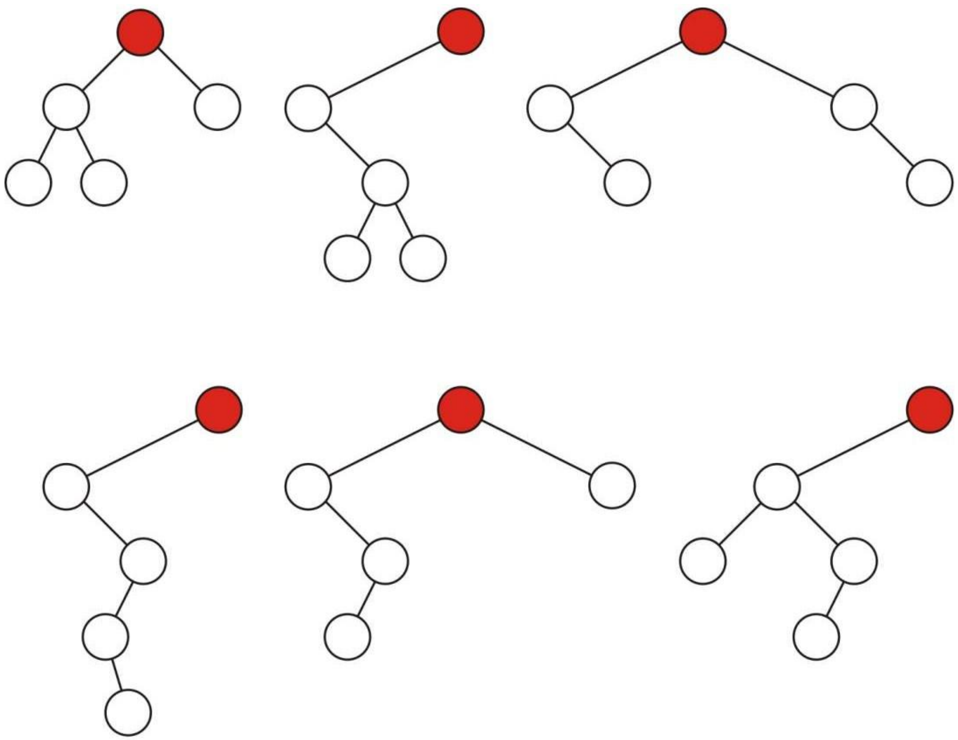


- Likewise, the two sub-trees are referred as
  - Left-hand subtree
  - Right-hand subtree



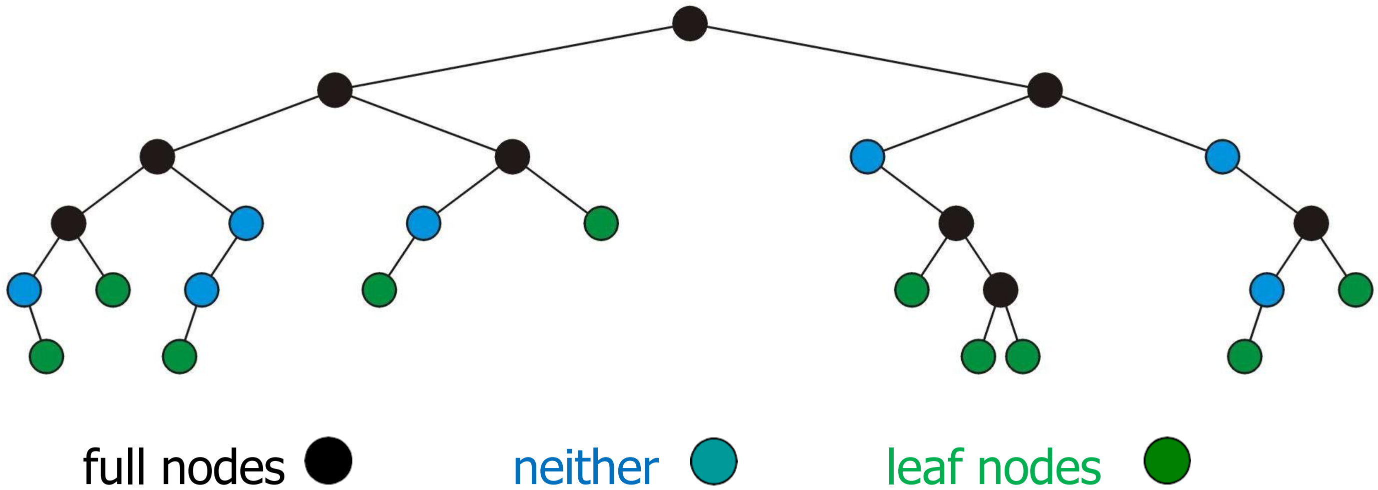
Binary Tree: Example

- Some variations on binary trees with five nodes



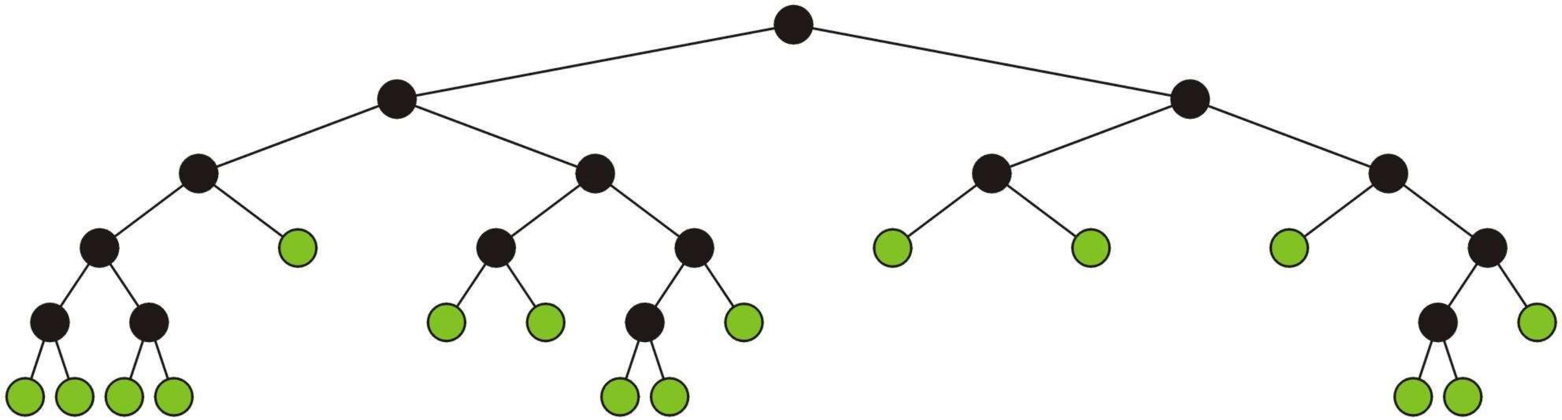
Binary Tree: Full Node

- A full node is a node where both the left and right sub-trees are non-empty trees



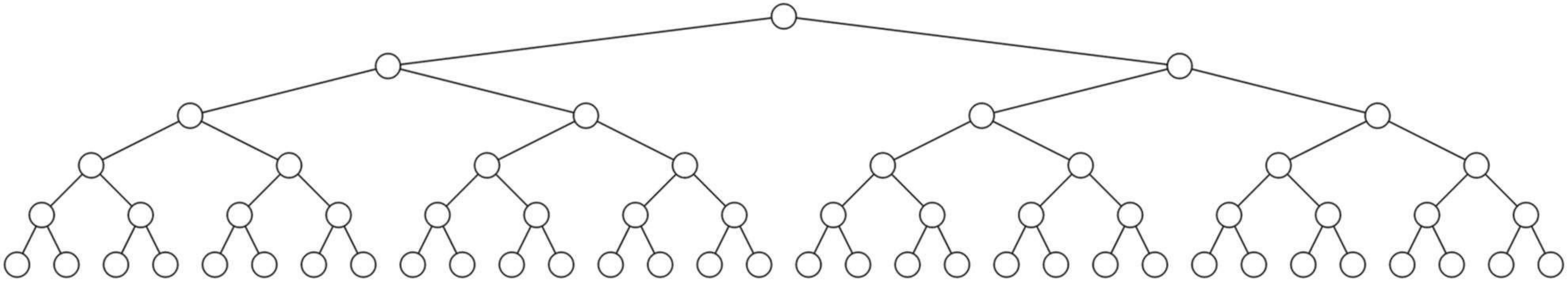
Full Binary Tree

- A full binary tree is where each node is:
  - A full node, or
  - A leaf node
- Full binary tree is also called proper binary tree, strictly binary tree or 2-tree



Complete (Or Perfect) Binary Tree

- A complete binary tree of height h is a binary tree where
  - All leaf nodes have the same depth h
  - All other nodes are full



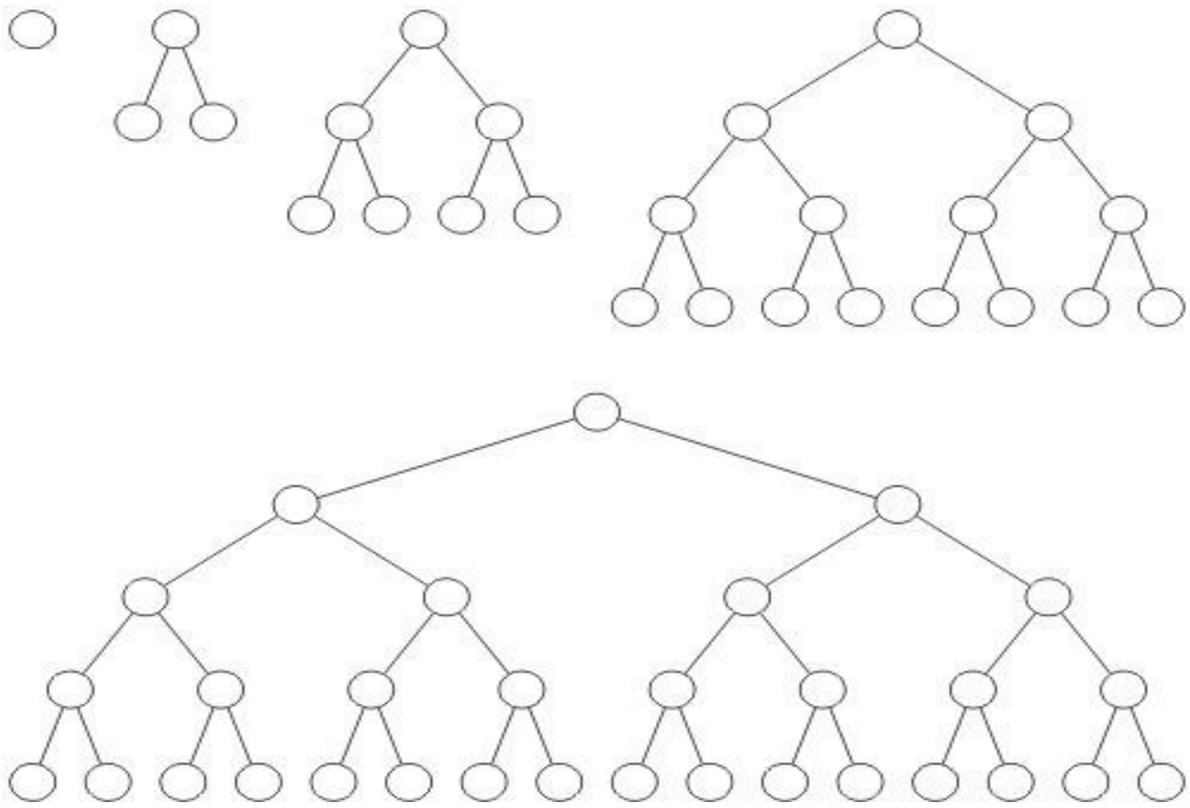
Complete Binary Tree: Recursive Definition

- A binary tree of height h = 0 is perfect

- A binary tree with height h > 0 is perfect
  - If both sub-trees are prefect binary trees of height h – 1

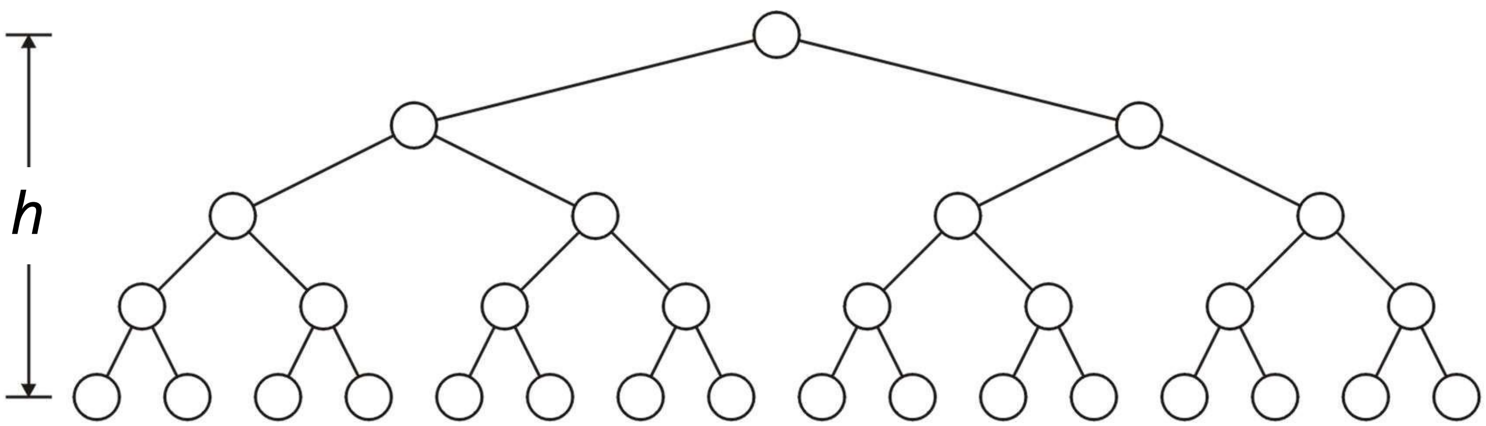
Complete Binary Tree: Example

- Complete binary trees of height h = 0, 1, 2, 3 and 4



Binary Tree: Properties

- A complete binary tree with height h has 2^h leaf nodes

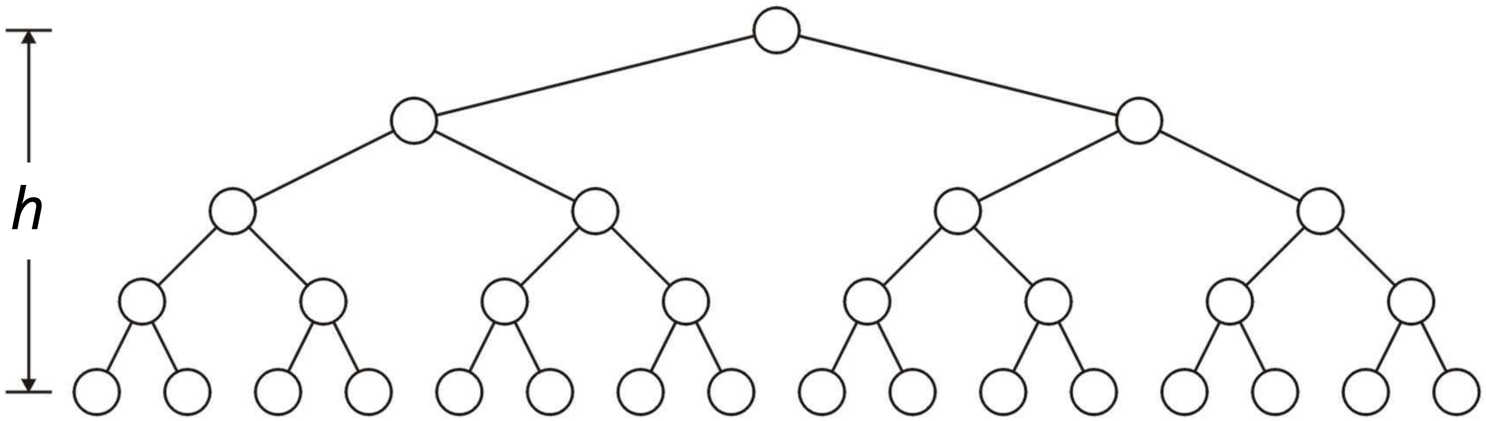


Binary Tree: Properties

- A complete binary tree with height h has 2^h leaf nodes

- A complete binary tree of height h has $2^{h+1} - 1$ nodes

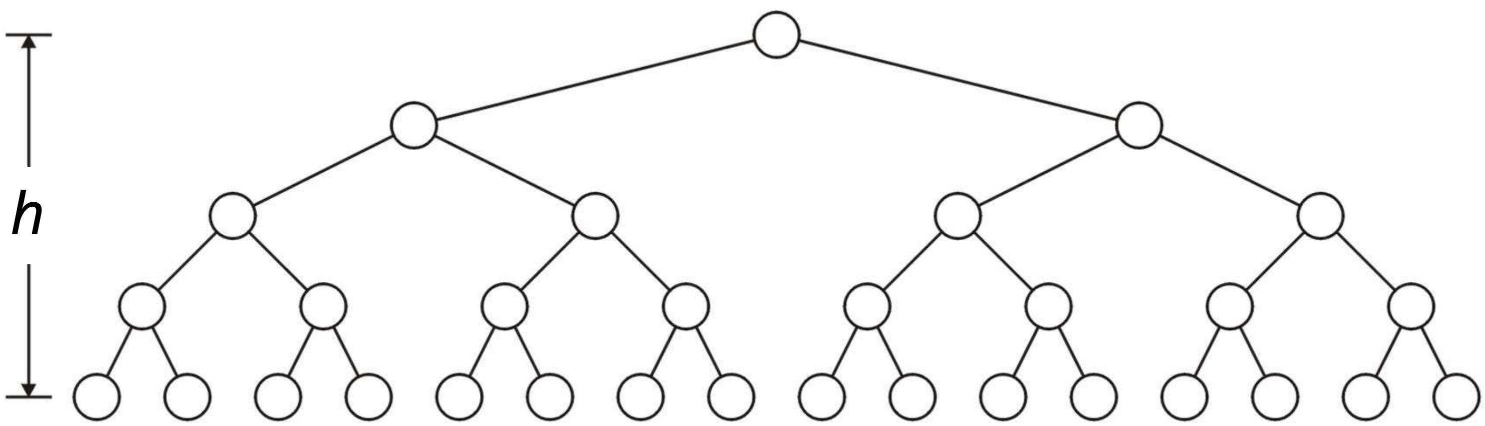
$$n = 2^0 + 2^1 + 2^2 + \ldots + 2^h = \sum_{j=0}^{h} 2^j = 2^{h+1} - 1$$



Binary Tree: Properties

- A complete binary tree with height h has 2^h leaf nodes

- A complete binary tree of height h has $2^{h+1} - 1$ nodes
  - Number of leaf nodes: $L = 2^h$
  - Number of internal nodes: $2^h - 1$
  - Total number of nodes: $2L-1 = 2^{h+1} - 1$



Binary Tree: Properties

- A complete binary tree with height h has 2^h leaf nodes
- A complete binary tree of height h has $2^{h+1} - 1$ nodes
  - Number of leaf nodes: $L = 2^h$
  - Number of internal nodes: $2^h - 1$
  - Total number of nodes: $2L-1 = 2^{h+1} - 1$
- A complete binary tree with n nodes has height $\log_2(n + 1) - 1$

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

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

$$h + 1 = \log_2(n + 1)$$

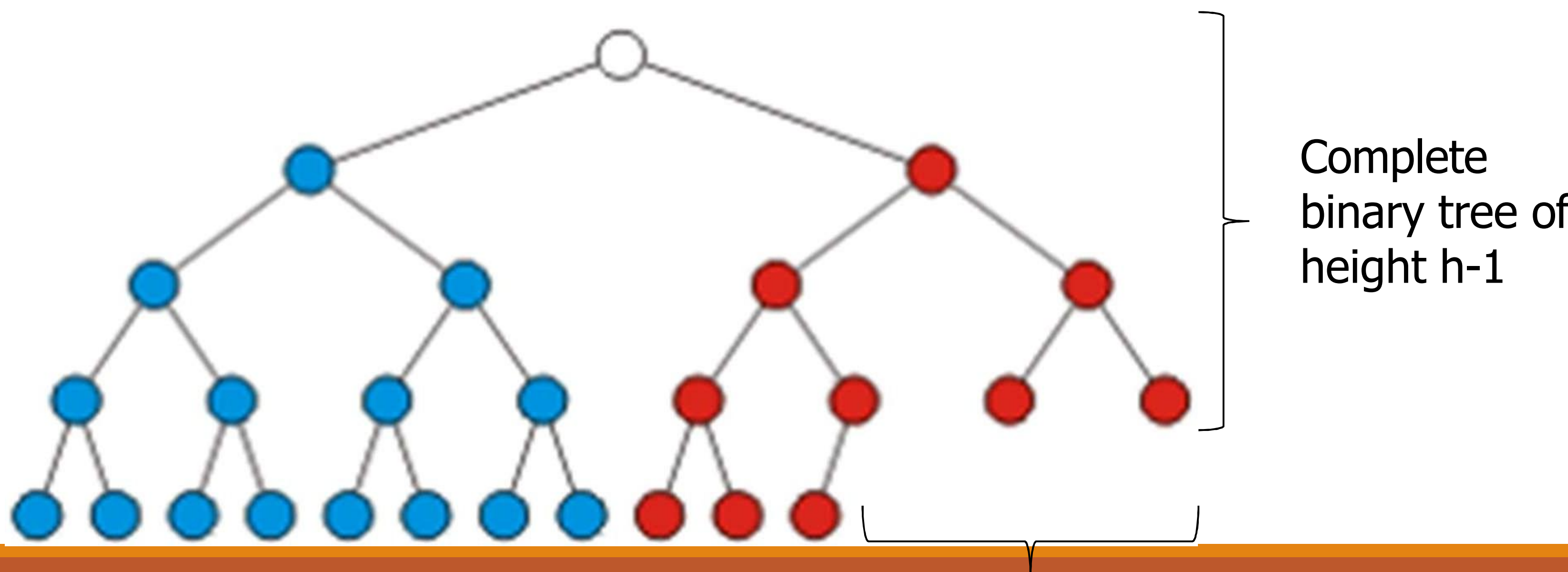
$$\Rightarrow h = \log_2(n + 1) - 1$$

Binary Tree: Properties

- A complete binary tree with height h has 2^h leaf nodes
- A complete binary tree of height h has $2^{h+1} - 1$ nodes
  - Number of leaf nodes: $L = 2^h$
  - Number of internal nodes: $2^h - 1$
  - Total number of nodes: $2L-1 = 2^{h+1} - 1$
- A complete binary tree with n nodes has height $\log_2(n + 1) - 1$
- Number n of nodes in a binary tree of height h is at least $h+1$ and at most $2^{h+1} - 1$

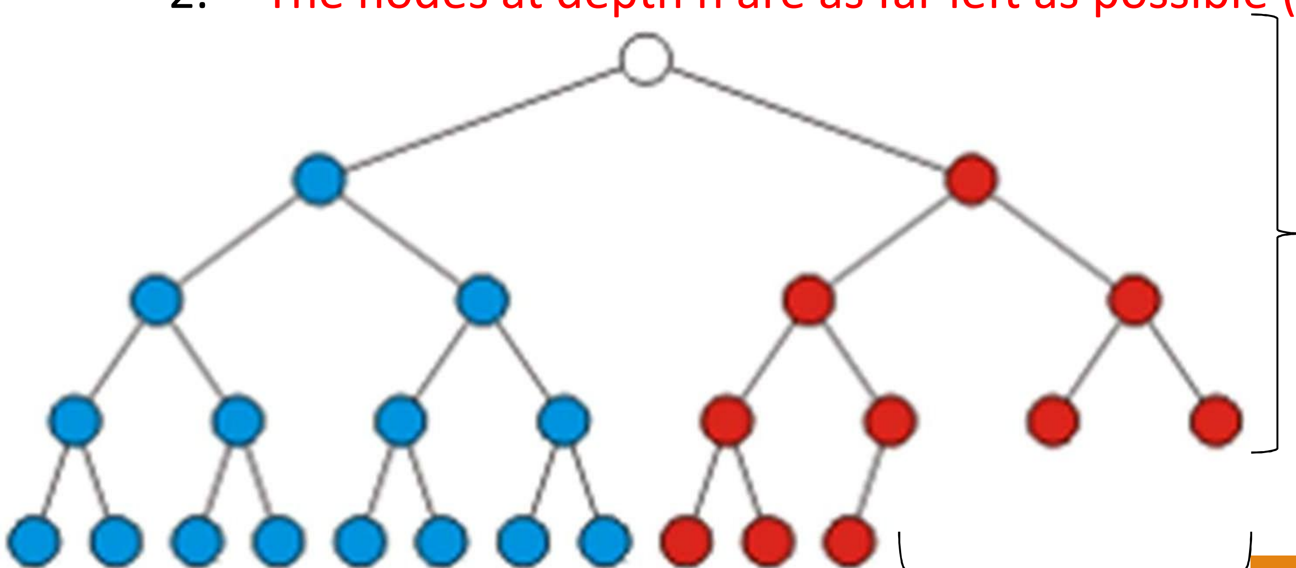
Almost (or Nearly) Complete Binary Tree

- Almost complete binary tree of height h is a binary tree in which
  1. There are 2^d nodes at depth d for d = 1,2,...,h−1
     - Each leaf in the tree is either at level h or at level h− 1
  2. The nodes at depth h are as far left as possible



Almost (or Nearly) Complete Binary Tree

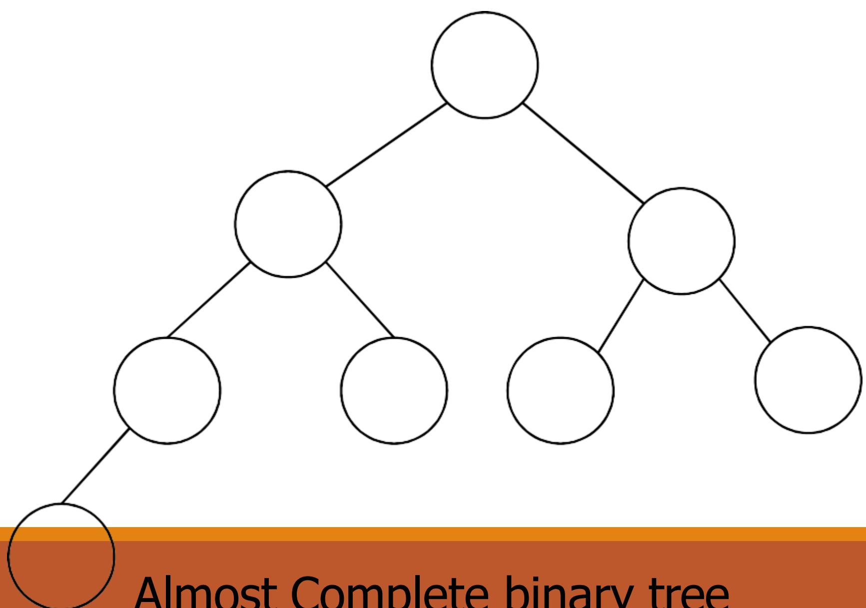
- Almost complete binary tree of height h is a binary tree in which
  1. There are 2d   nodes at depth d for d = 1,2,...,h−1
     - Each leaf in the tree is either at level h or at level h– 1
  2. The nodes at depth h are as far left as possible (Formal ?)



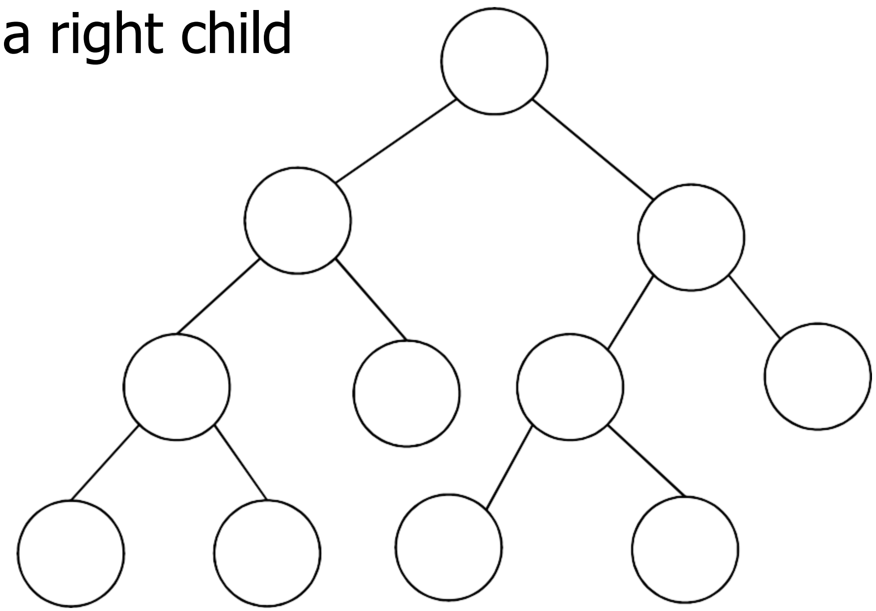
Almost (or Nearly) Complete Binary Tree

Condition 2: The nodes at depth h are as far left as possible

- If a node p at depth h−1 has a left child
  - Every node at depth h−1 to the left of p has 2 children
- If a node at depth h−1 has a right child
  - It also has a left child



Almost Complete binary tree



Not Almost Complete binary tree
(condition 2 violated)

Tree ADT

- Data Type: Any type of objects can be stored in a tree

- Accessor methods
  - `root()` – return the root of the tree
  - `parent(p)` – return the parent of a node
  - `children(p)` – return the children of a node

- Query methods
  - `size()` – return the number of nodes in the tree
  - `isEmpty()` – return true if the tree is empty
  - `elements()` – return all elements
  - `isRoot(p)` – return true if node p is the root

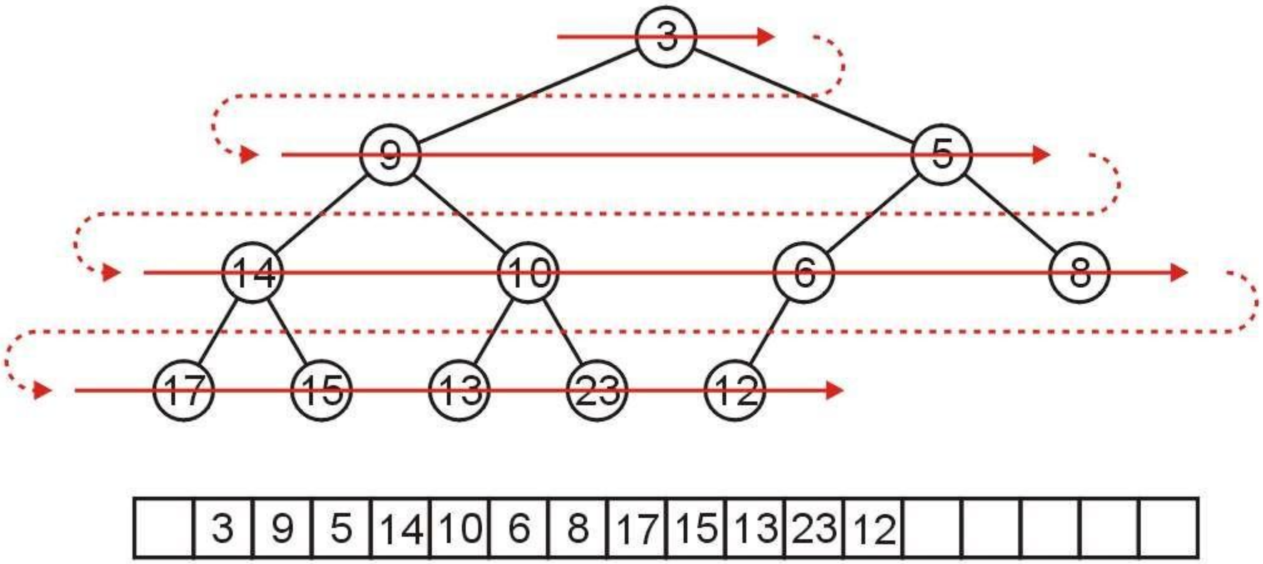
- Other methods
  - Tree traversal, Node addition/deletion, create/destroy

Binary Tree Storage

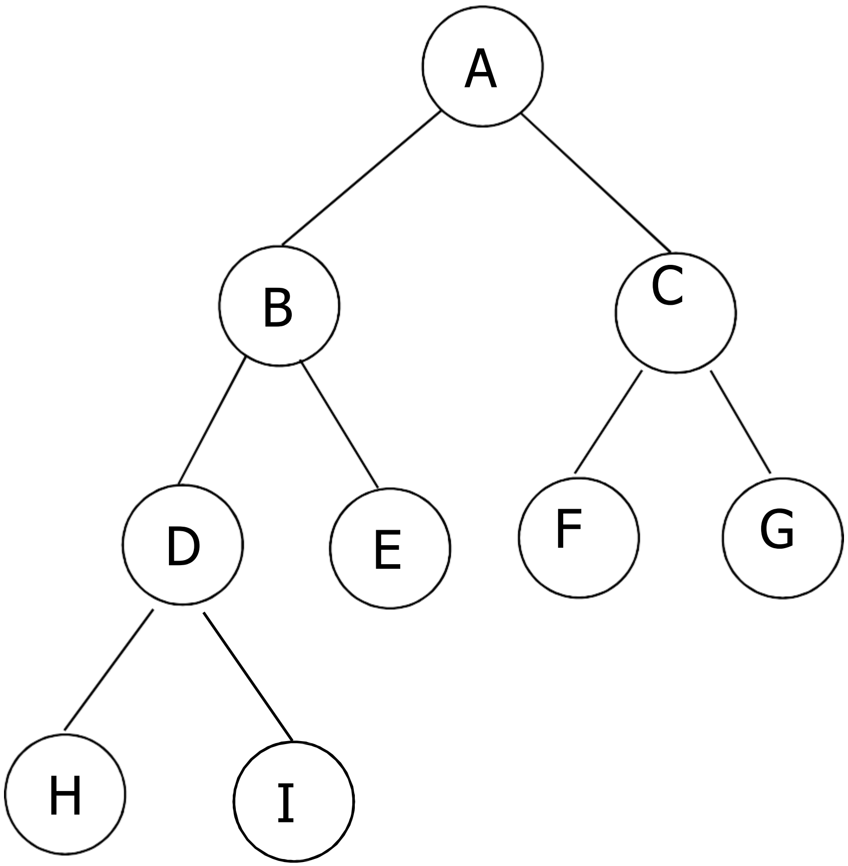
- Contiguous Storage (Array Storage)
- Linked List based Storage

Array Storage

- We are able to store a binary tree as an array
- Traverse tree in breadth-first order, placing the entries into array
  - Storage of elements (i.e., objects/data) starts from root node
  - Nodes at each level of the tree are stored left to right



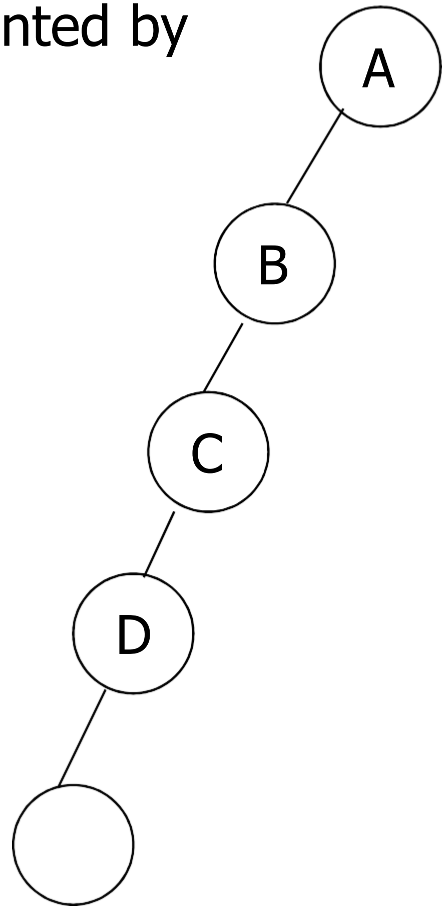
Array Storage Example



[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Array Storage Example

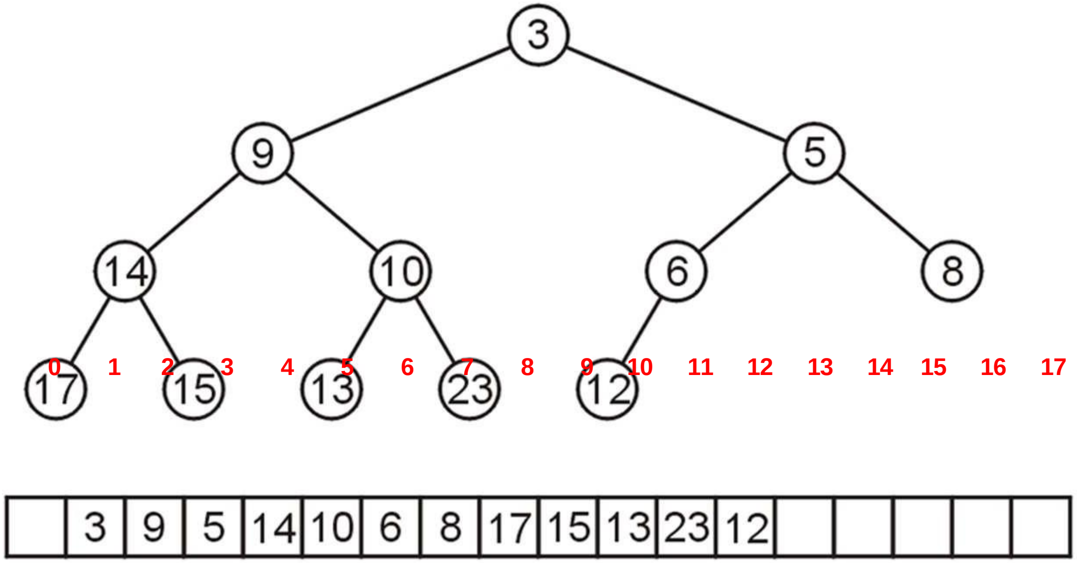
- Unused nodes in tree represented by a predefined bit pattern



[1]	A
[2]	B
[3]	-
[4]	C
[5]	-
[6]	-
[7]	-
[8]	D
[9]	-
…	
[16]	E

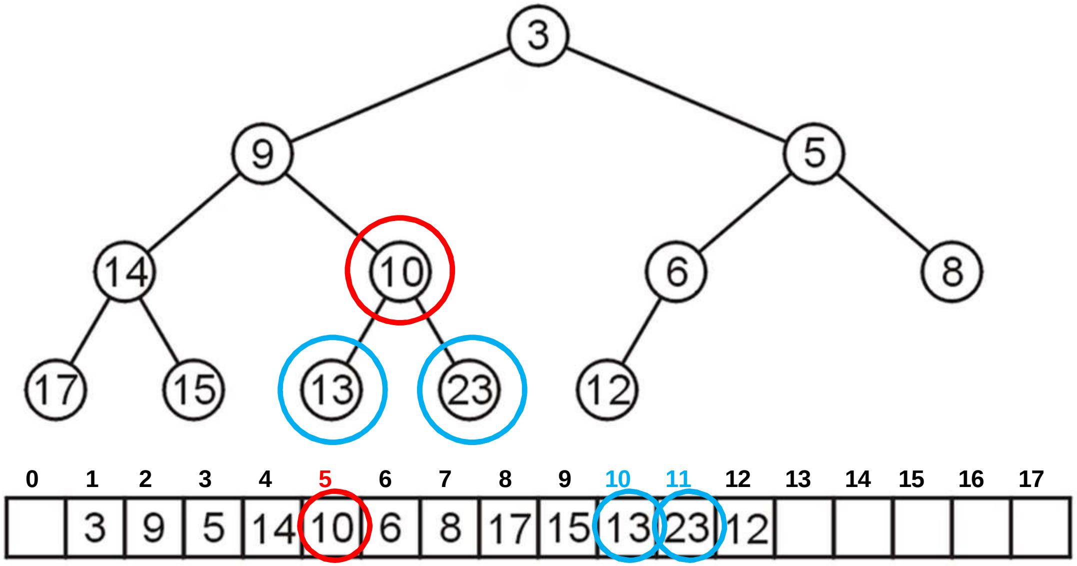
Array Storage

- The children of the node with index k are in 2k and 2k + 1
- The parent of node with index k is in k ÷ 2



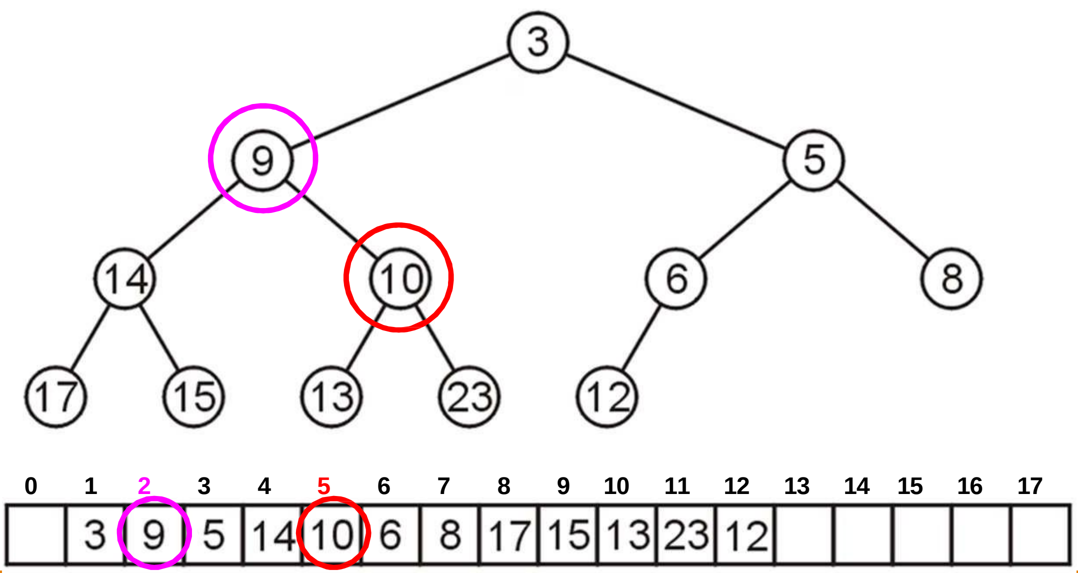
Array Storage Example

- Node 10 has index 5
  - Its children 13 and 23 have indices 10 and 11, respectively



Array Storage Example

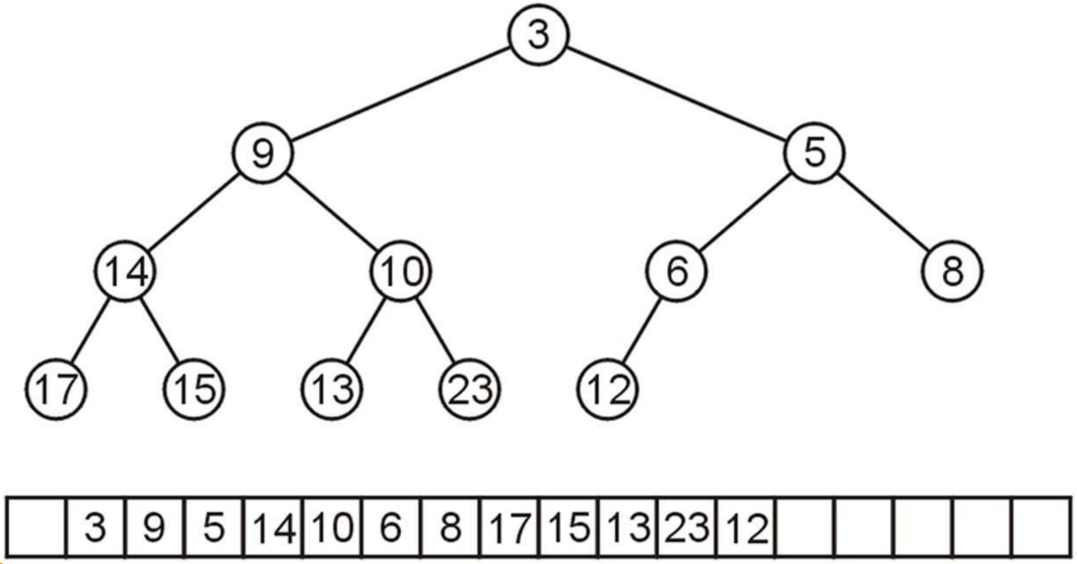
- Node 10 has index 5
  - Its children 13 and 23 have indices 10 and 11, respectively
  - Its parent is node 9 with index 5/2 = 2



Array Storage

- Why array index is not started from 0
  - In C++, this simplifies the calculations

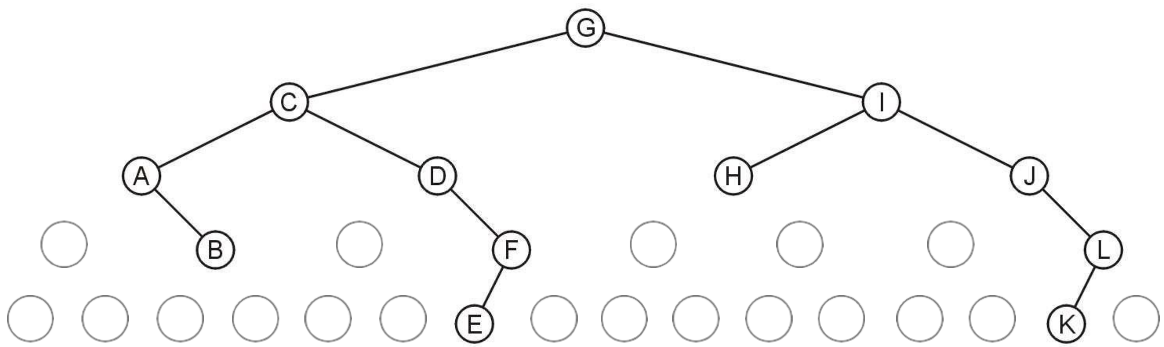
```
parent = k >> 1;
left_child = k << 1;
right_child = left_child | 1;
```



Array Storage: Disadvantage

- Why not store any tree as an array using breadth-first traversals?
  - There is a significant potential for a lot of wasted memory

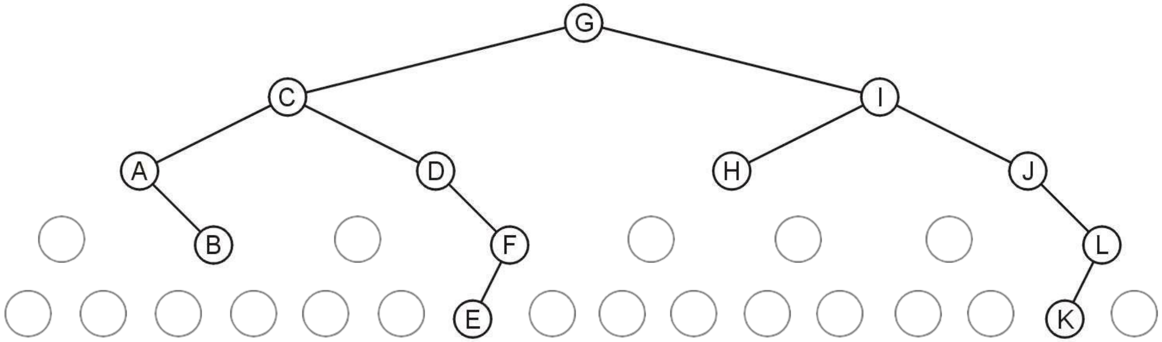
- Consider the following tree with 12 nodes
  - What is the required size of array?



☒	G	C	I	A	D	H	J		B		F				L								E								K	

Array Storage: Disadvantage

- Why not store any tree as an array using breadth-first traversals?
  - There is a significant potential for a lot of wasted memory
- Consider the following tree with 12 nodes
  - What is the required size of array? **32**
  - What will be the array size if a child is added to node K?

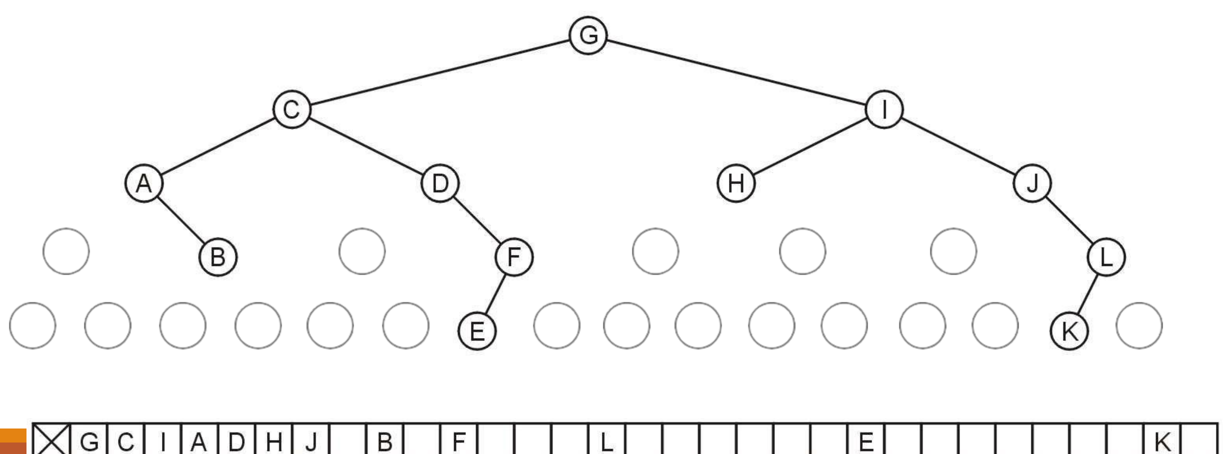


☒	G	C	I	A	D	H	J		B		F				L								E								K	

Array Storage: Disadvantage

- Why not store any tree as an array using breadth-first traversals?
  - There is a significant potential for a lot of wasted memory

- Consider the following tree with 12 nodes
  - What is the required size of array? **32**
  - What will be the array size if a child is added to node K? **double**



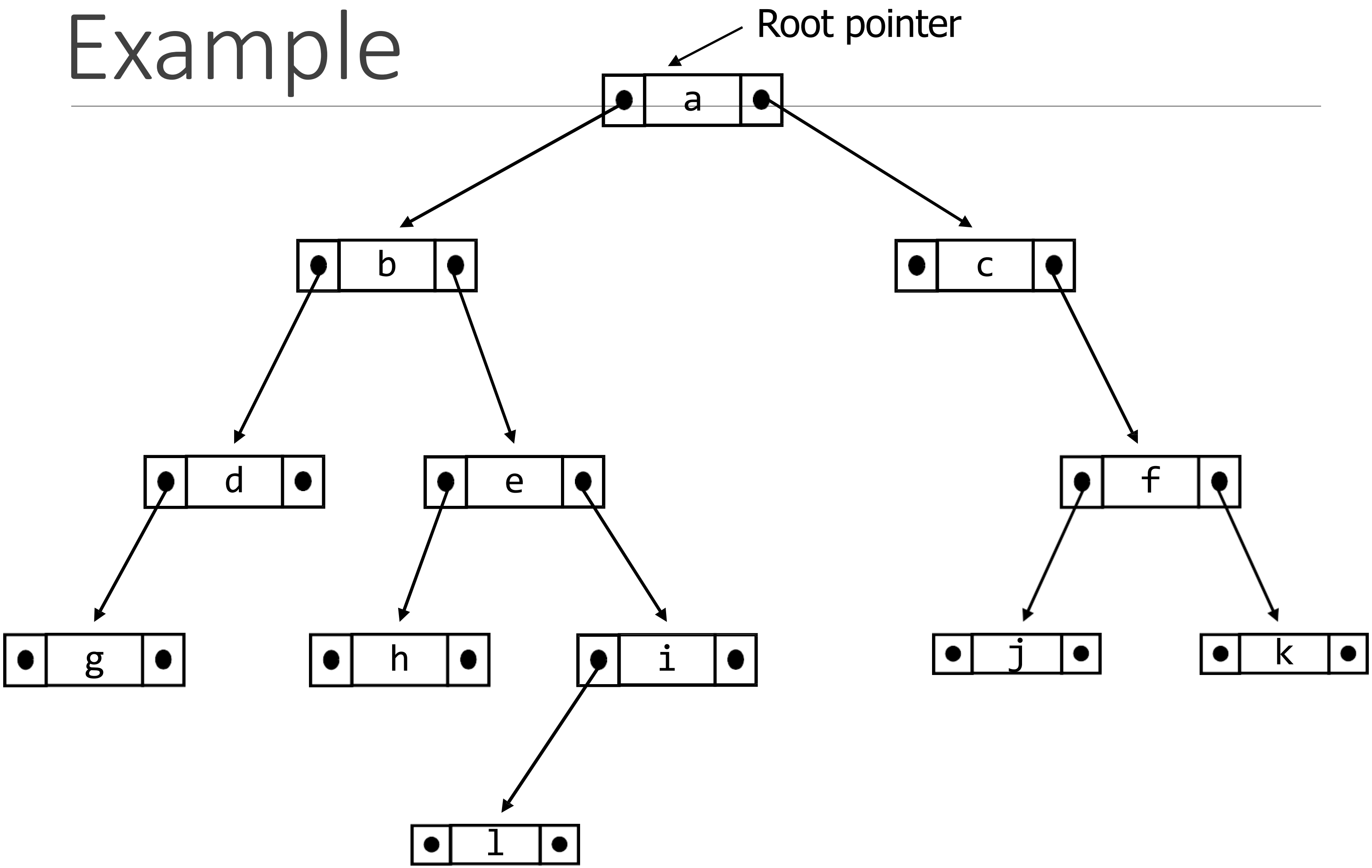
As Linked List Structure

- We can implement a binary tree by using a class which:
  - Stores an element
  - A left child pointer (pointer to first child)
  - A right child pointer (pointer to second child)

```
class Node{
   Type value;
   Node *LeftChild,*RightChild;
}root;
```

- The root pointer points to the root node
  - Follow pointers to find every other element in the tree
- Leaf nodes have `LeftChild` and `RightChild` pointers set to `NULL`

As Linked List Structure: Example



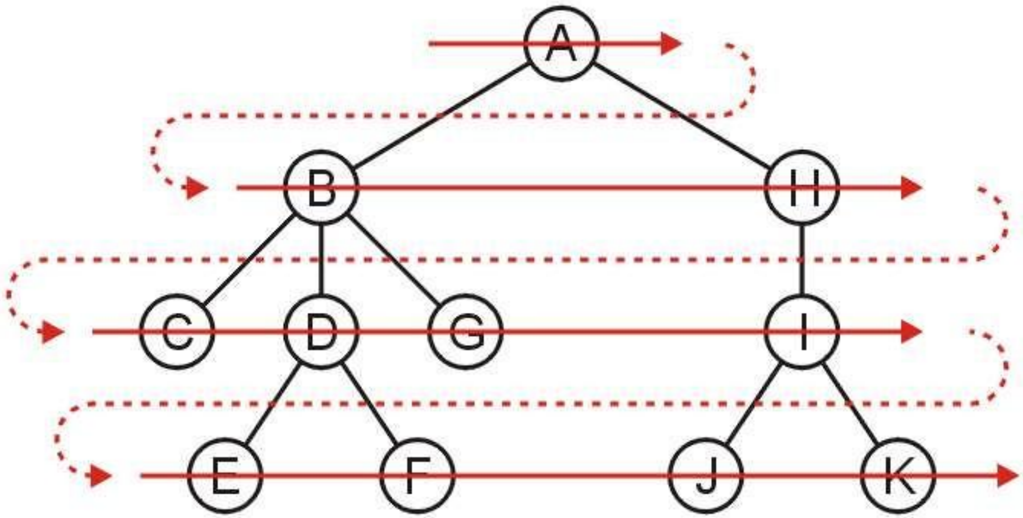
Tree Traversal

- To traverse (or walk) the tree is to visit each node in the tree exactly once
  - Traversal must start at the root node
    - There is a pointer to the root node of the binary tree

- Two types of traversals
  - Breadth-First Traversal
  - Depth-First Traversal

Breadth-First Traversal (For Arbitrary Trees)

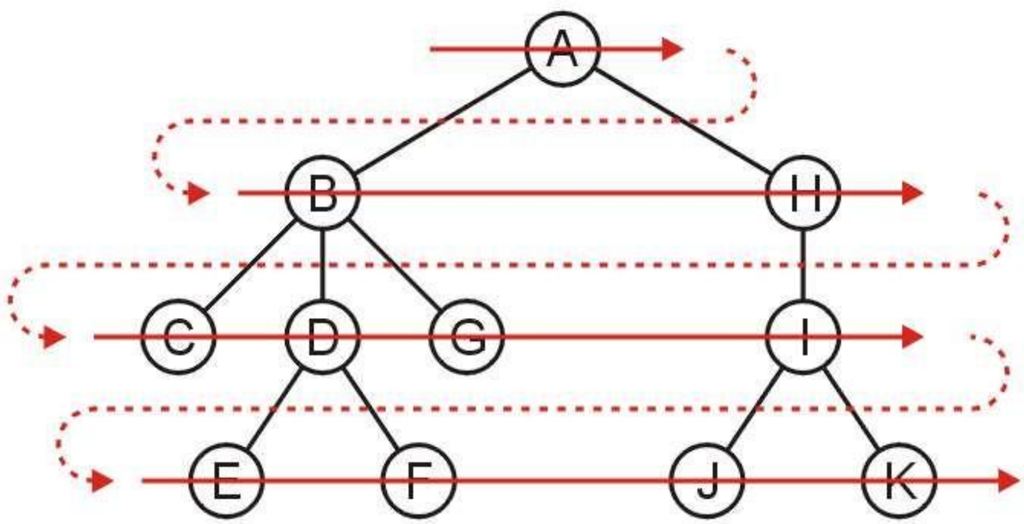
- All nodes at a given depth `d` are traversed before nodes at `d+1`
- Can be implemented using a queue



- Order: A B H C D G I E F J K

Breadth-First Traversal – Implementation

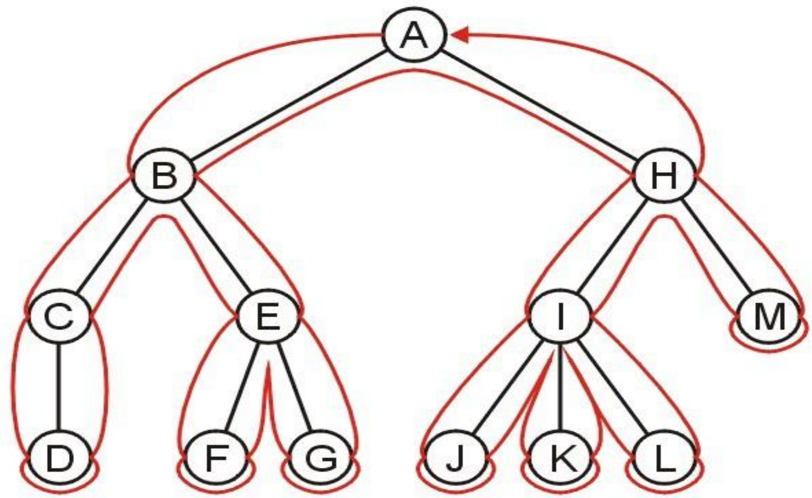
- Create a queue and push the root node onto the queue
- While the queue is not empty:
  - Enqueue all children of the front node onto the queue
  - Dequeue the front node



Depth-First Traversal (For Arbitrary Trees)

- Traverse as much as possible along the branch of each child before going to the next sibling
  - Nodes along one branch of the tree are traversed before backtracking

- Each node could be visited multiple times in such a scheme
  - The first time the node is approached (before any children)
  - The last time it is approached (after all children)



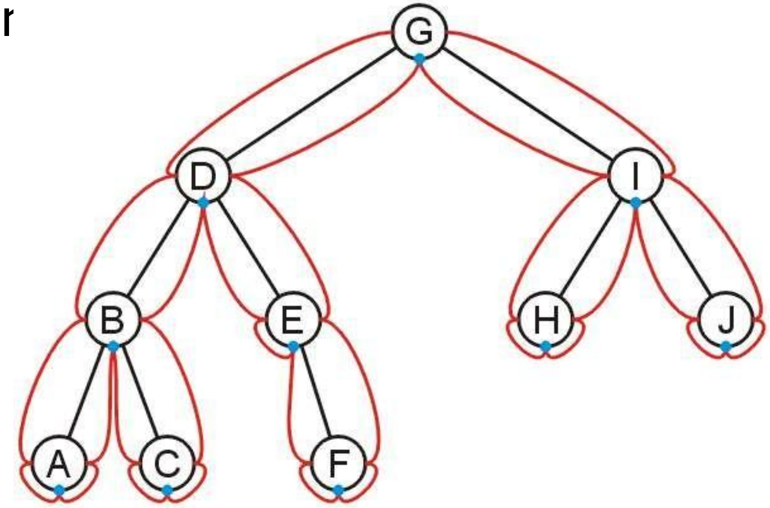
Depth-First Tree Traversal (Binary Trees)

- For each node in a binary tree, there are three choices
  - Visit the node first
  - Visit the one of the subtrees first
  - Visit the both the subtrees first

- These choices lead to three commonly used traversals
  - Inorder traversal: (Left subtree) visit Root (Right subtree)
  - Preorder traversal: visit Root (Left subtree) (Right subtree)
  - Postorder traversal: (Left subtree) (Right subtree) visit Root

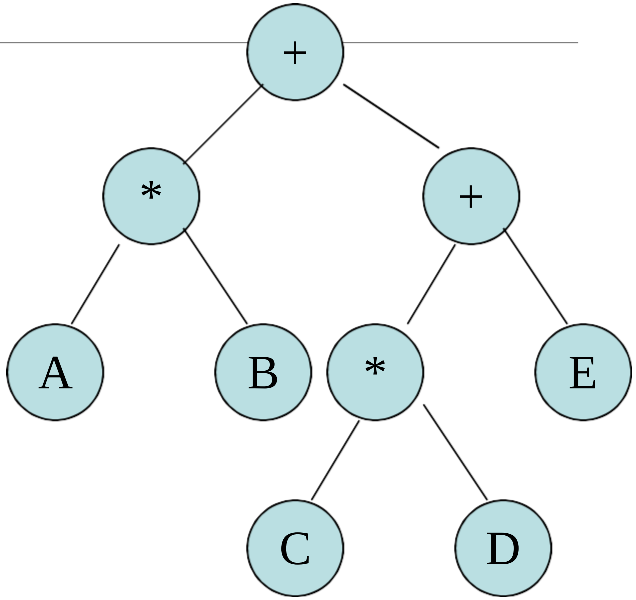
Inorder Traversal

- Algorithm
  1. Traverse the left subtree in inorder
  2. Visit the root
  3. Traverse the right subtree in inor



A, B, C, D, E, F, G, H, I, J

Inorder Traversal



- Algorithm

1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse the right subtree in inorder

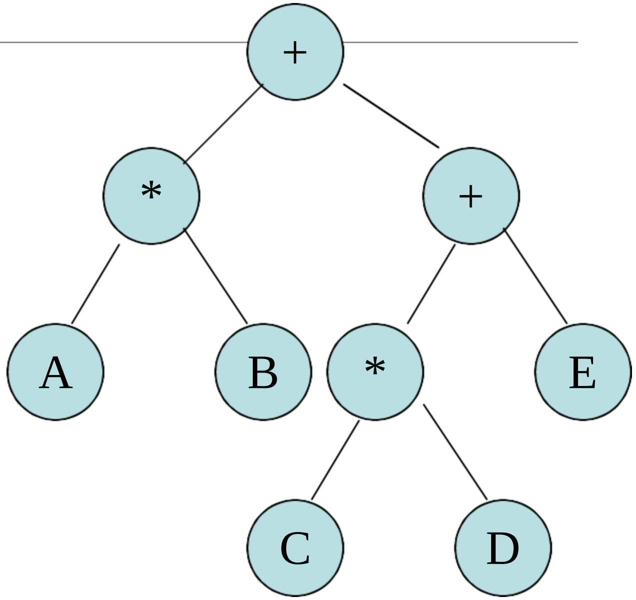
- Example
  - Left + Right
  - [Left * Right] + [Left + Right]
  - (A * B) + [(Left * Right) + E)
  - (A * B) + [(C * D) + E]

Inorder Traversal – Implementation

```
void inorder(Node *p) const
{
   if (p != NULL)
   {
      inorder(p->leftChild);
      cout << p->info << " ";
      inorder(p->rightChild);
   }
}

void main () {
   . . .
   inorder (root);
}
```

Preorder Traversal



- Algorithm

1. Visit the node
2. Traverse the left subtree
3. Traverse the right subtree

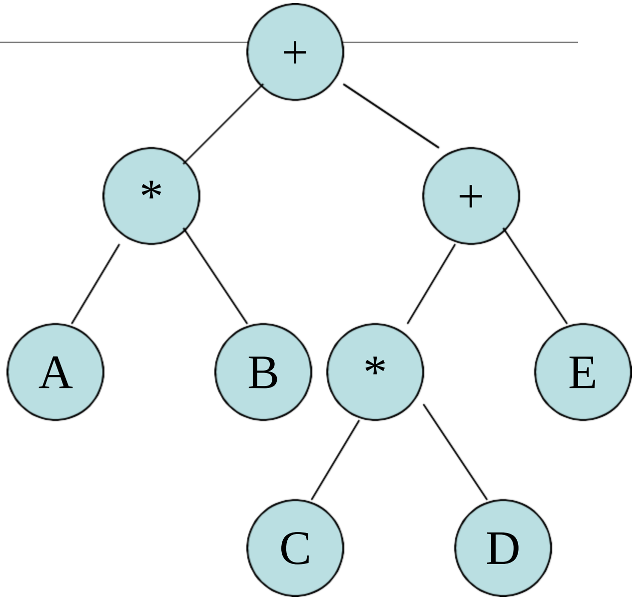
- Example
  - + Left Right
  - + [* Left Right] [+ Left Right]
  - + (* AB) [+ * Left Right E]
  - +*AB + *C D E

Preorder Traversal – Implementation

```
void preorder(Node *p) const
{
    if (p != NULL)
    {
        cout << p->info << " ";
        preorder(p->leftChild);
        preorder(p->rightChild);
    }
}

void main () {
    . . .
    preorder (root);
}
```

Postorder Traversal



- Algorithm

  1. Traverse the left subtree
  2. Traverse the right subtree
  3. Visit the node

- Example
  - Left Right +
  - [Left Right *] [Left Right+] +
  - (AB*) [Left Right * E +]+
  - (AB*) [C D * E +]+
  - AB* C D * E + +

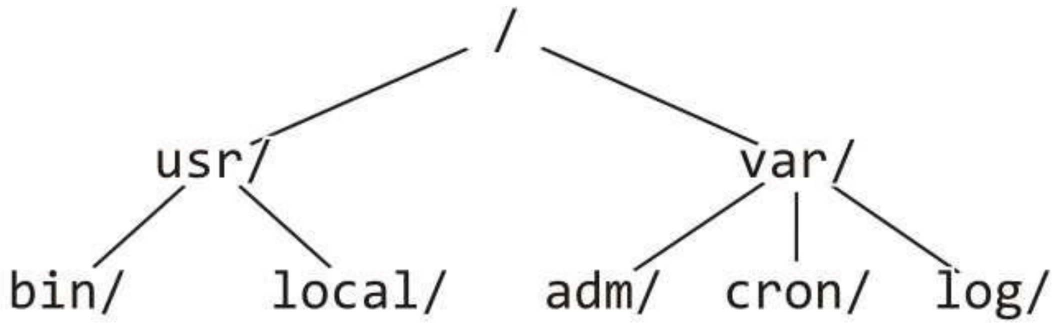
Postorder Traversal – Implementation

```
void postorder(Node *p) const
{
   if (p != NULL)
   {
      postorder(p->leftChild);
      postorder(p->rightChild);
      cout << p->info << " ";
   }
}

void main () {
   . . .
   postorder (root);
}
```

Example: Printing a Directory Hierarchy

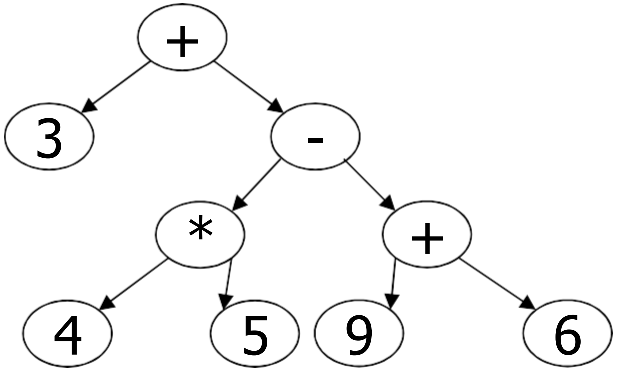
- Consider the directory structure presented on the left
  - Which traversal should be used?




```
/
   usr/
      bin/
      local/
   var/
      adm/
      cron/
      log/
```

Expression Tree

- Each algebraic expression has an inherent tree-like structure
- An expression tree is a binary tree in which
  - The parentheses in the expression do not appear
    - ➢ Tree representation captures the intent of parenthesis
  - The leaves are the variables or constants in the expression
- The non-leaf nodes are the operators in the expression
  - ➢ Binary operator has two non-empty subtrees
  - ➢ Unary operator has one non-empty subtree



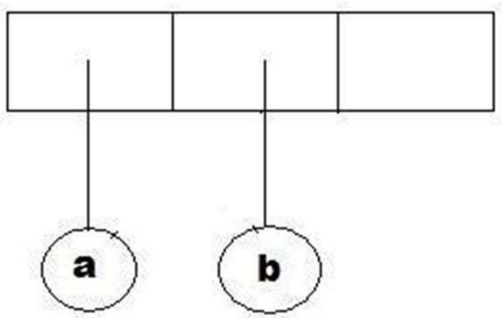
Convert Postfix into Expression Tree – Algorithm

```
while(not the end of the expression) {
if(the next symbol in the expression is an operand) {
  create a node for the operand ;
  push the reference to the created node onto the stack ;
}
if(the next symbol in the expression is a binary operator) {
  create a node for the operator ;
  pop from the stack a reference to an operand ;
  make the operand the right subtree of the operator node ;
  pop from the stack a reference to an operand ;
  make the operand the left subtree of the operator node ;
  push the reference to the operator node onto the stack ; } }
```

Convert Postfix into Expression Tree – Example

```
while(not the end of the expression)
{
  if(the next symbol is an operand) {
      create a node for the operand ;
      push the reference to the created node onto the stack;
  }
  if(the next symbol is a binary operator) {
      create an operator node;
      pop operant from the stack;
      make the operand the right subtree ;
      pop operand from the stack;
      make the operand the left subtree ;
      push the operator node onto the stack;
}}
```

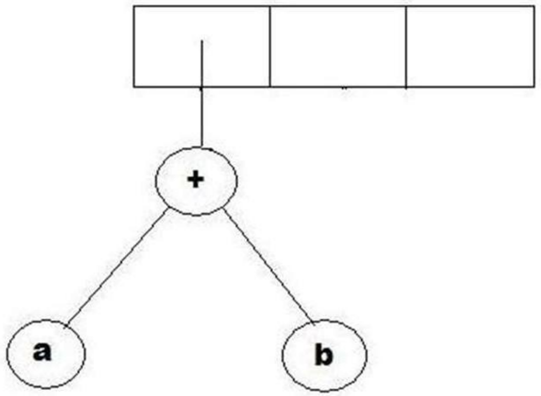
Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example

```
while(not the end of the expression)
{
   if(the next symbol is an operand) {
        create a node for the operand ;
        push  the reference to the created node onto the stack;
   }
   if(the next symbol is a binary operator) {
        create an operator node;
        pop  operant from the stack;
        make the operand the right subtree ;
        pop  operand from the stack;
        make the operand the left subtree ;
        push  the operator node onto the stack;
}}
```

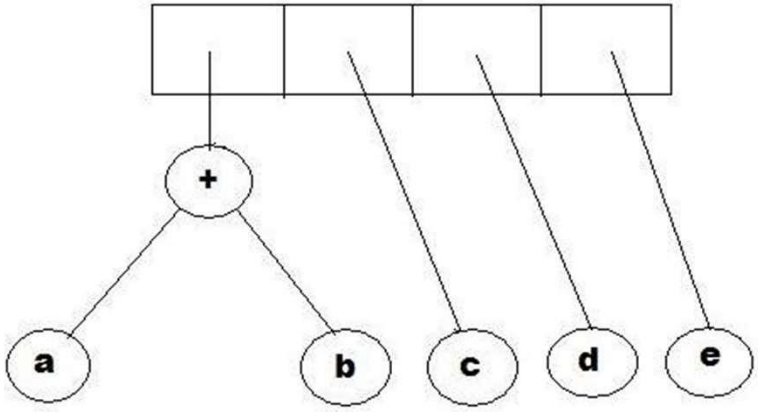
Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example

```
while(not the end of the expression)
{
  if(the next symbol is an operand) {
      create a node for the operand ;
      push  the reference to the created node onto the stack;
  }
  if(the next symbol is a binary operator) {
      create an operator node;
      pop  operant from the stack;
      make the operand the right subtree ;
      pop  operand from the stack;
      make the operand the left subtree ;
      push  the operator node onto the stack;
 }}
```

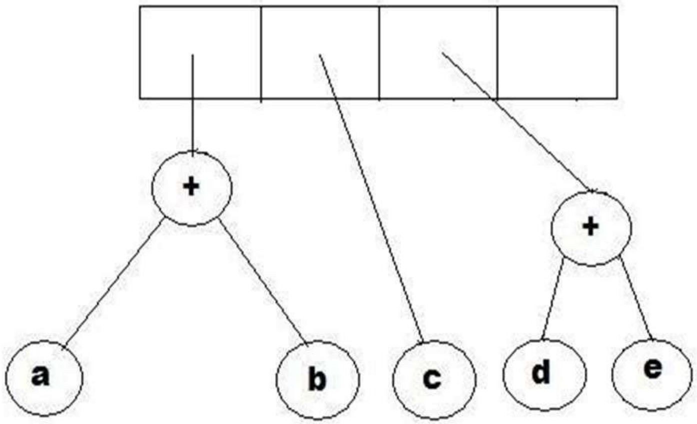
Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example

```
while(not the end of the expression)
{
   if(the next symbol is an operand) {
      create a node for the operand ;
      push  the reference to the created node onto the stack;
   }
   if(the next symbol is a binary operator) {
      create an operator node;
      pop  operant from the stack;
      make the operand the right subtree ;
      pop  operand from the stack;
      make the operand the left subtree ;
      push  the operator node onto the stack;
}}
```

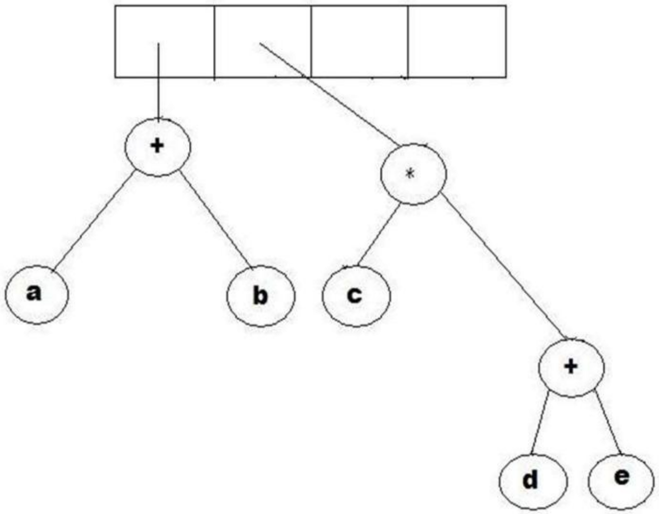
Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example

```
while(not the end of the expression)
{
   if(the next symbol is an operand) {
       create a node for the operand ;
       push  the reference to the created node onto the stack;
   }
   if(the next symbol is a binary operator) {
       create an operator node;
       pop  operant from the stack;
       make the operand the right subtree ;
       pop  operand from the stack;
       make the operand the left subtree ;
       push  the operator node onto the stack;
}}
```

Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example

while(not the end of the expression)

{

if(the next symbol is an operand) {

create a node for the operand ;

push the reference to the created node onto the stack;

}

if(the next symbol is a binary operator) {

create an operator node;

pop operant from the stack;

make the operand the right subtree ;

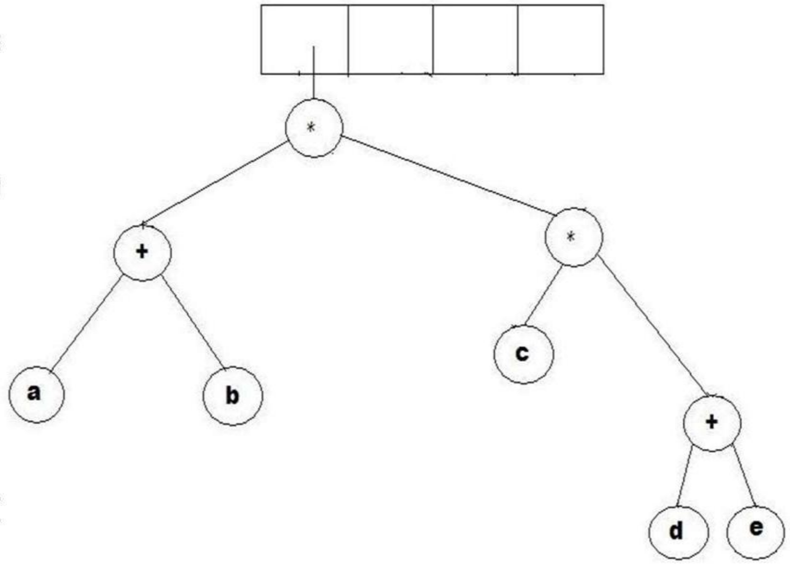
pop operand from the stack;

make the operand the left subtree ;

push the operator node onto the stack;

}}

Example:
a b + c d e + * *



Why Expression Tree?

- Expression trees impose a hierarchy on the operations
  - Terms deeper in the tree get evaluated first
  - Establish correct precedence of operations without using parentheses

- A compiler will read an expression in a language like C++/Java, and transform it into an expression tree

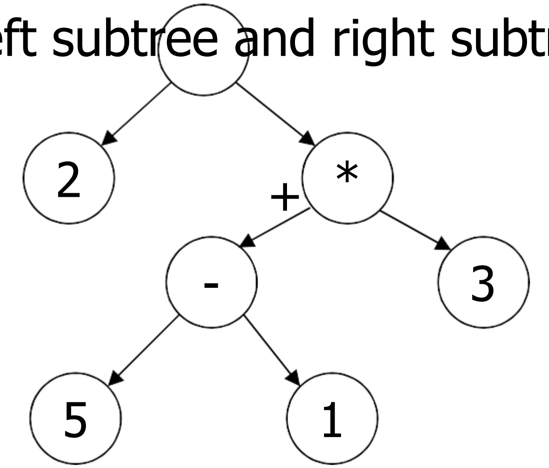
- Expression trees can be very useful for:
  - Evaluation of the expression
  - Generating correct compiler code to actually compute the expression's value at execution time

Evaluating an Expression Tree

- Perform a post-order traversal of the tree
  - Ask each node to evaluate itself

- An operand node evaluates itself by just returning its value

- An operator node has to apply the operator
  - To the results of evaluations from its left subtree and right subtree



Order of evaluation:

3 **1** **2**

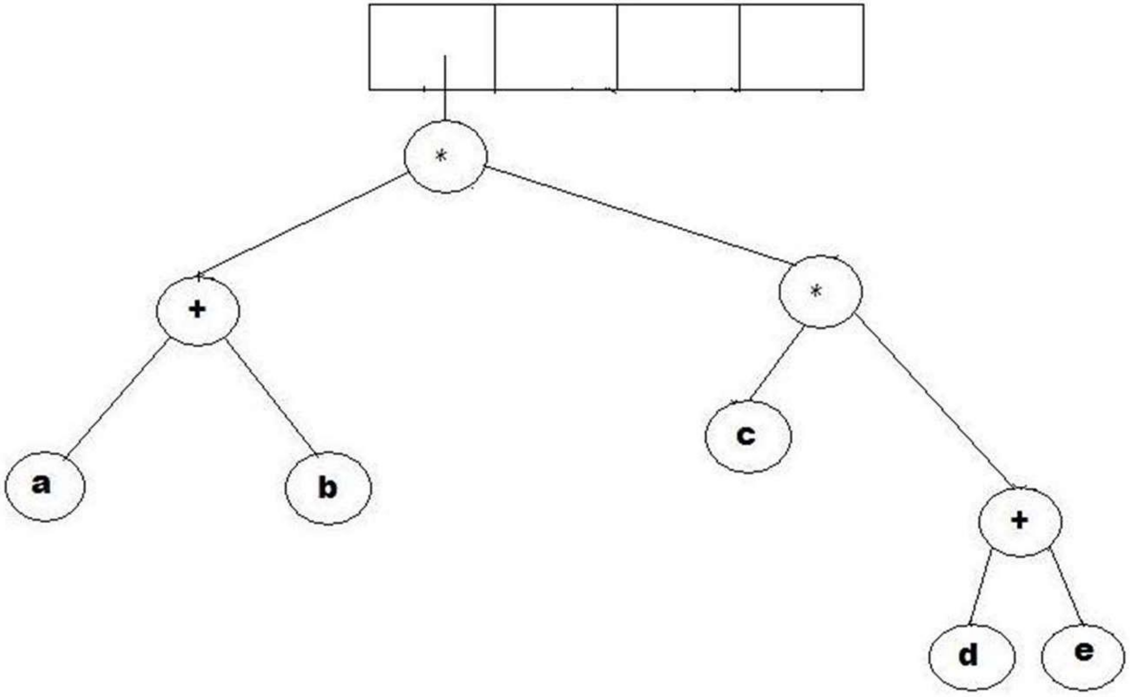
(2 + ((5 – 1) * 3))

Evaluating an Expression Tree – Example

- Expression:

a b + c d e + * *

1 2 + 3 4 5 + * *



Evaluating an Expression Tree - Implementation

```
evaluate(ExpressionTree t){
   if(t is a leaf)
      return value of t's operand ;
   else{
    operator =  t.element ;
    operand1 = evaluate(t.left) ;
    operand2 = evaluate(t.right) ;
    return(applyOperator(operand1, operator, operand2) ;
  }
}
```

Any Question So Far?