



CS-2001

Data Structures

Fall 2023

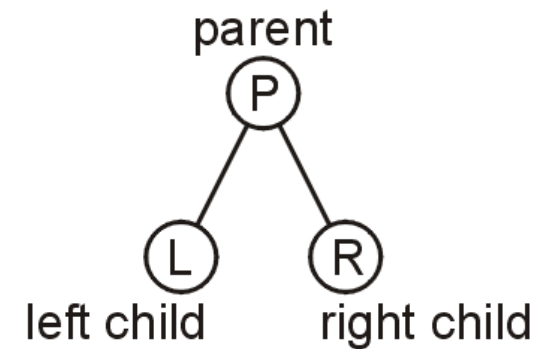
Binary Tree and Tree ADT

Mr. Muhammad Usman Joyia

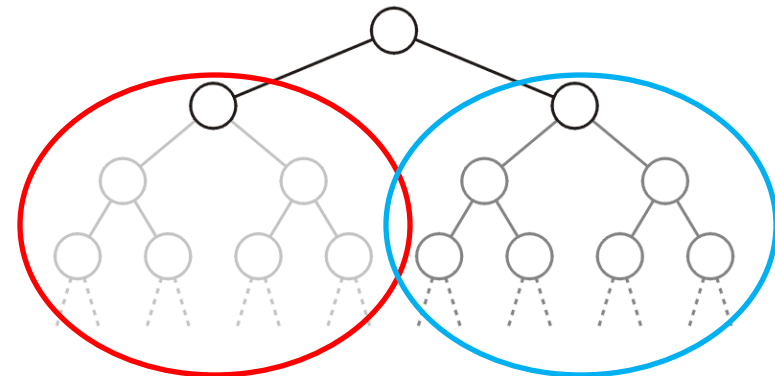
National University of Computer and
Emerging Sciences,
Faisalabad, Pakistan.

Binary Tree

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

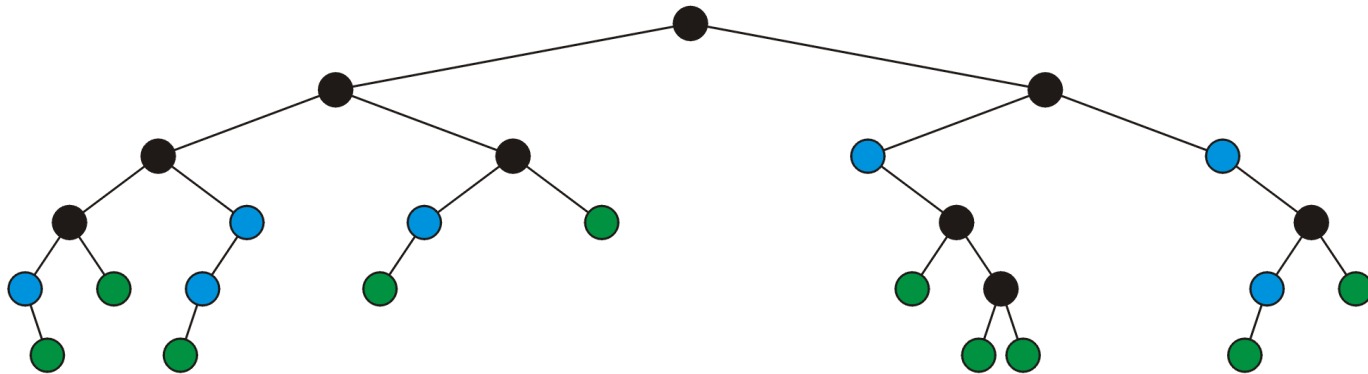


- Likewise, the two sub-trees are referred to as
 - Left sub-tree
 - Right sub-tree



Binary Tree: Full Node

- A **full node** is a node where both the left and right sub-trees are non-empty trees
- (OR) if it has exactly two child nodes



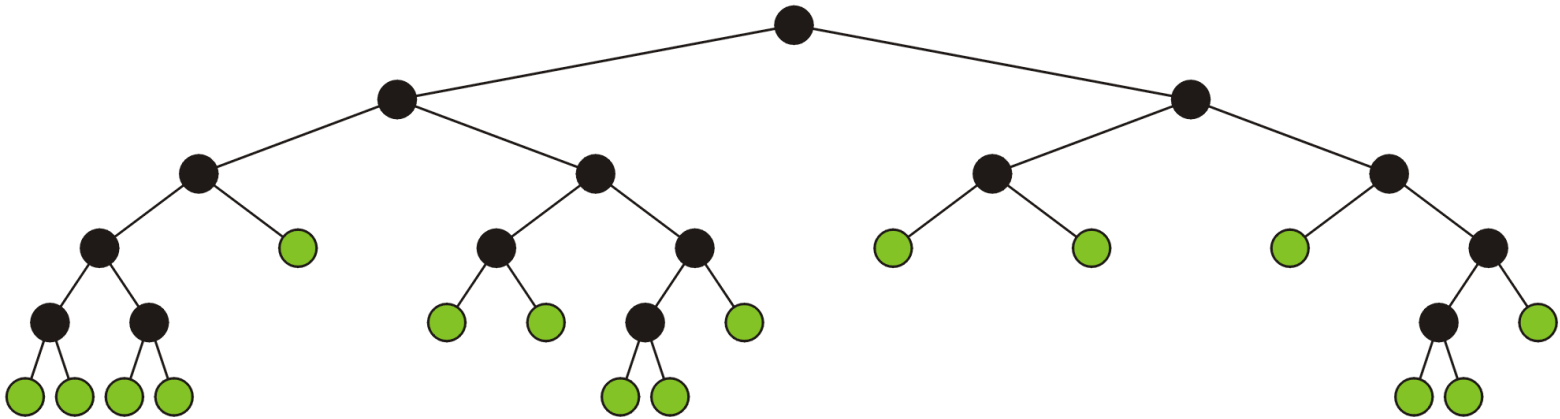
full nodes ●

neither ●

leaf nodes ●

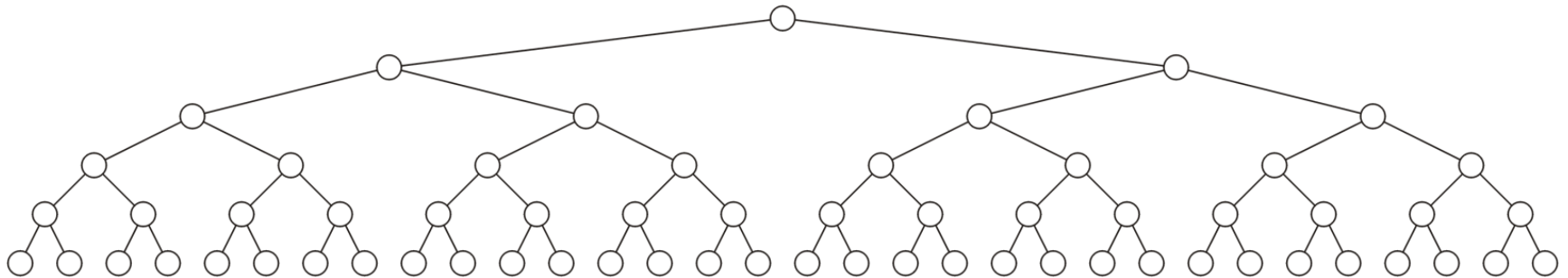
Full Binary Tree

- 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**



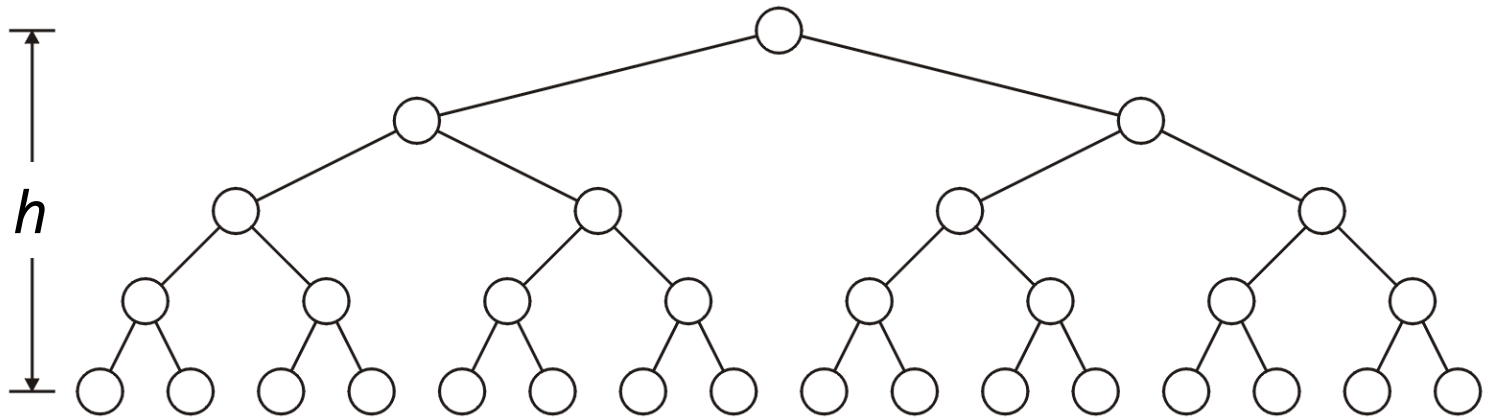
Perfect Binary Tree

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



Binary Tree: Properties (3)

- A perfect binary tree with height h has 2^h leaf nodes
- A perfect 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 (4)

- A perfect binary tree with height h has 2^h leaf nodes
- A perfect 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 perfect binary tree with n nodes has **height** $\log_2(n + 1) - 1$

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

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

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

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

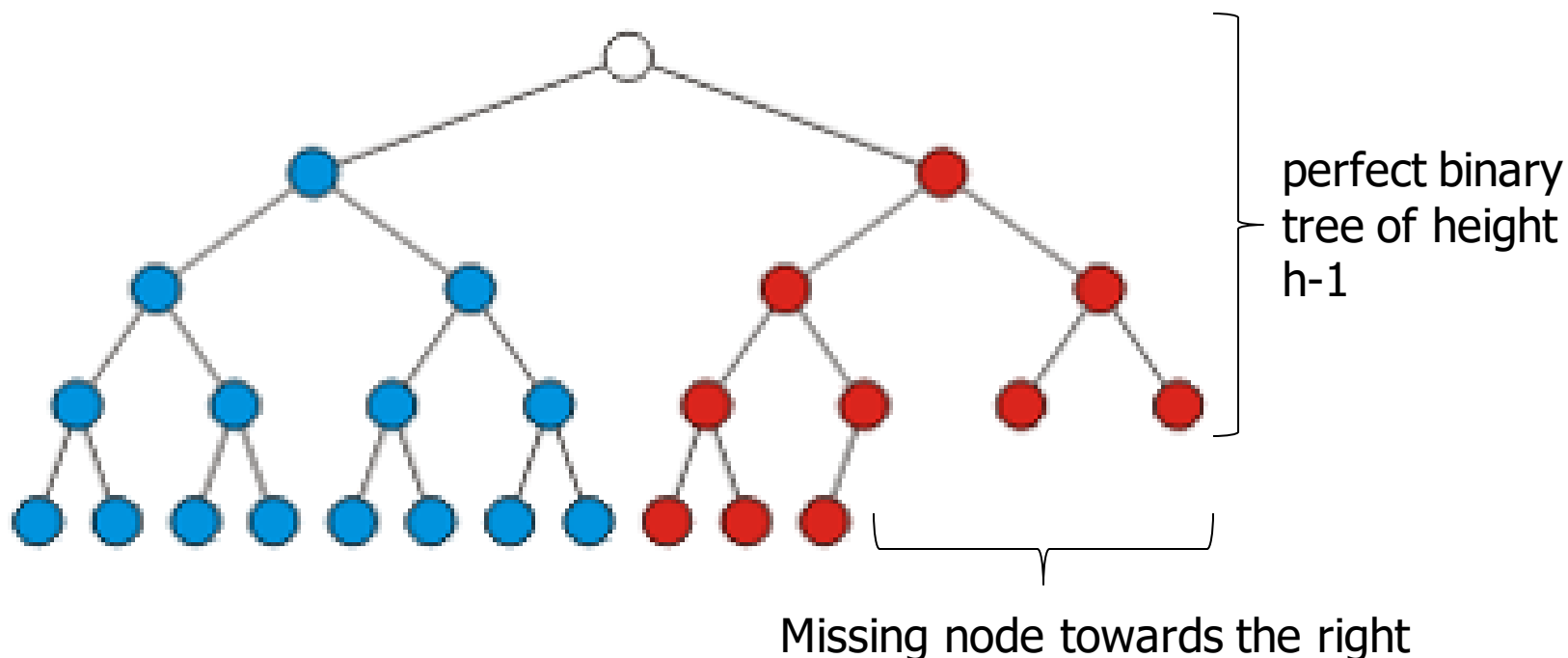
Binary Tree: Properties (4)

- A perfect binary tree with height h has 2^h leaf nodes
- A perfect 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 perfect 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] Complete Binary tree

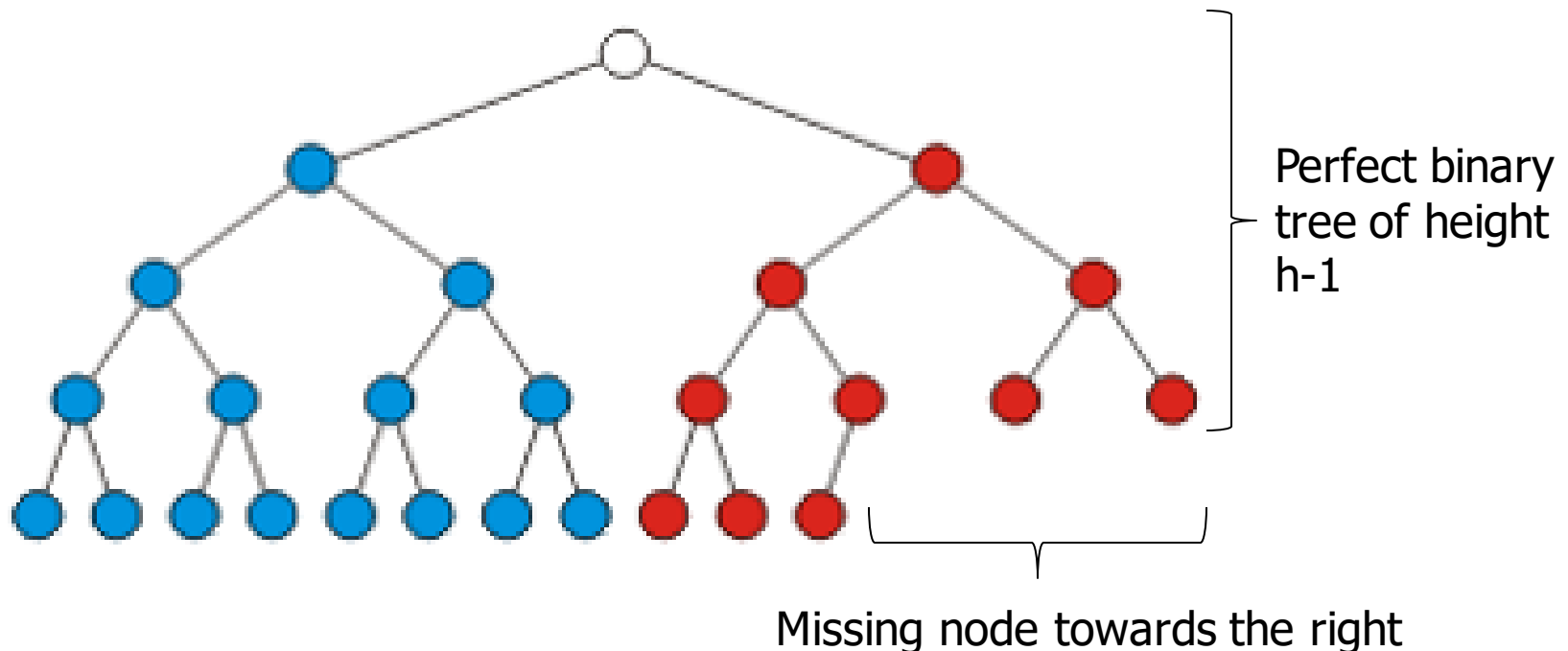
Almost (or Nearly) Complete Binary Tree

- Almost complete binary tree of height h is a binary tree in which
 1. There are 2^d nodes at depth d for $d = 1, 2, \dots, h-1$
 - 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



Complete Binary Tree

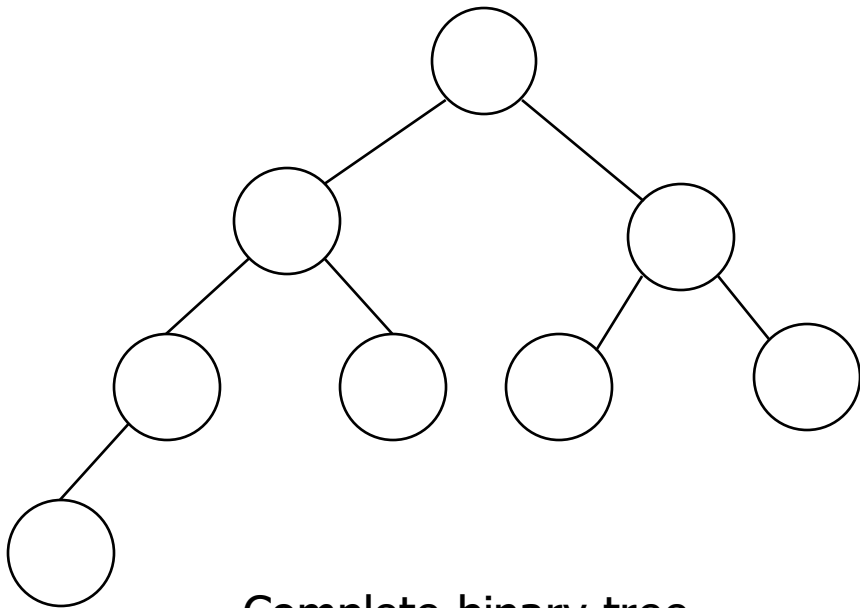
- Complete binary tree of height h is a binary tree in which
 - There are 2^d nodes at depth d for $d = 1, 2, \dots, h-1$
 - Each leaf in the tree is either at level h or at level $h - 1$
 - The nodes at depth h are as far left as possible (Formal ?)**



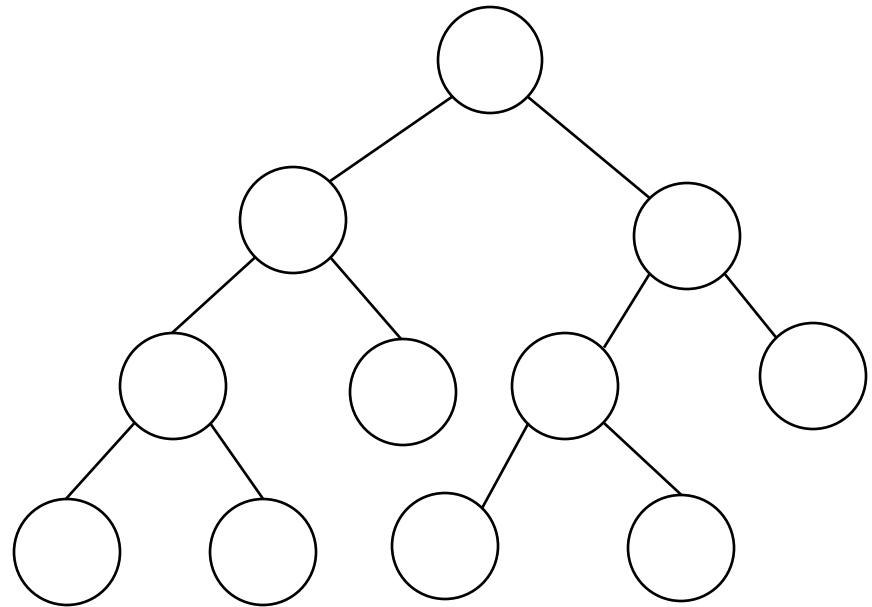
Complete Binary Tree

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

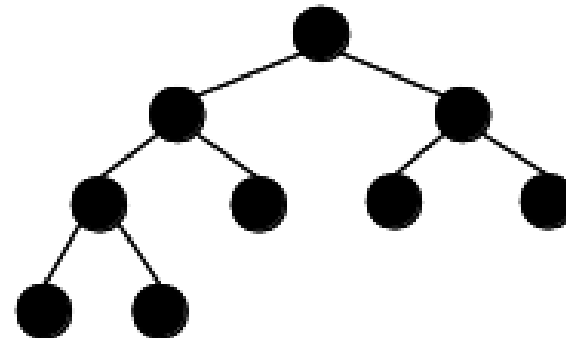
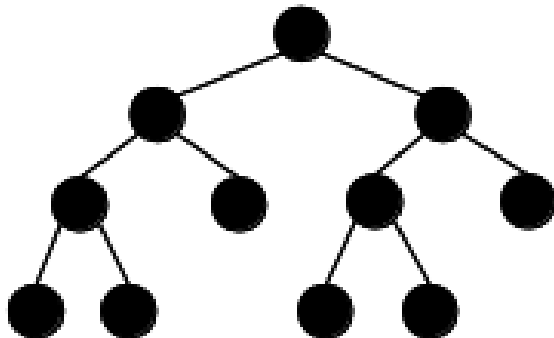
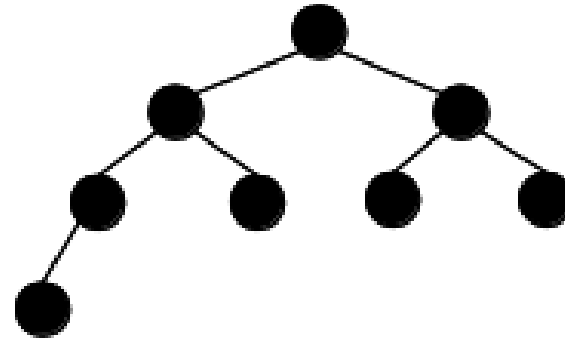
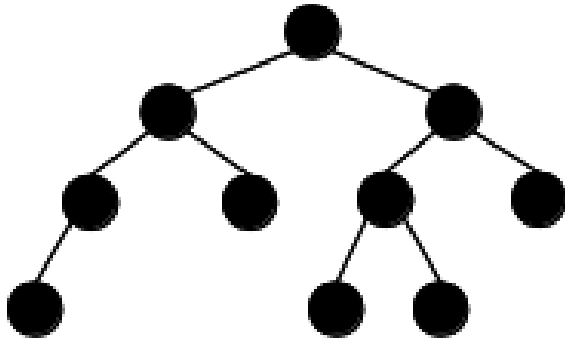


Complete binary tree

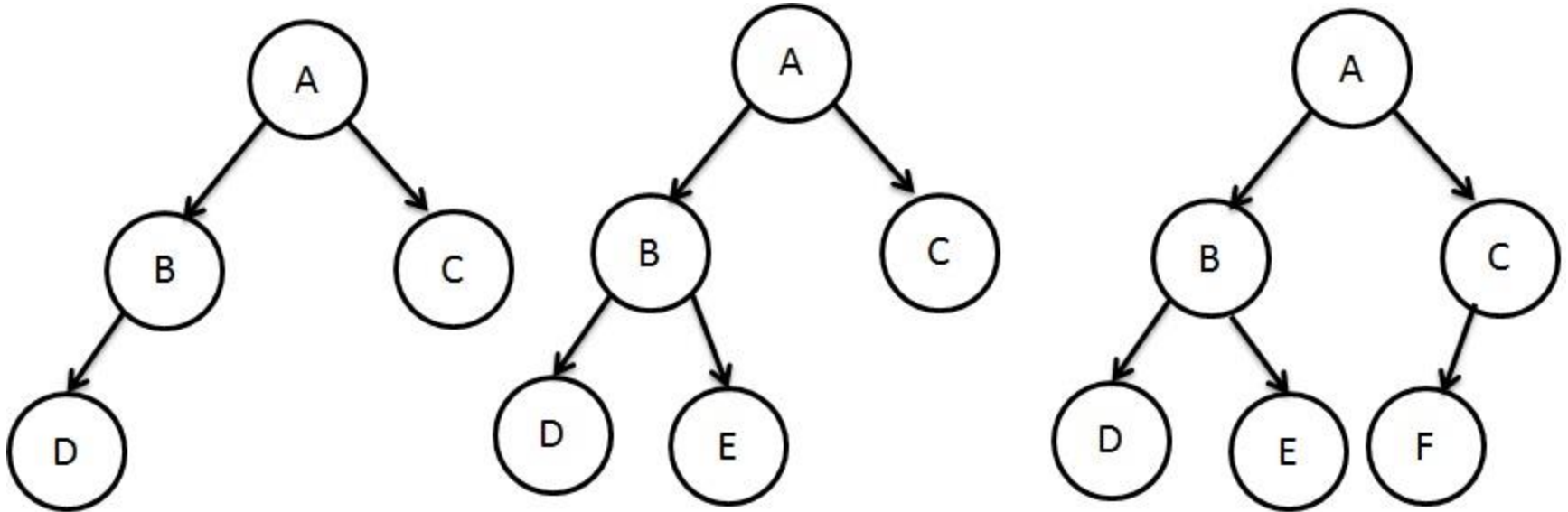


Not a Complete binary tree
(condition 2 violated)

Full vs. Complete Binary Tree



Complete Binary Trees...



What is the height and number of nodes for each tree?

Complete Binary Tree: Properties

- Total number of nodes n are between
 - **At least:** perfect binary tree of height $h-1 + 1$ (i.e., 1 node in the next level) $\rightarrow 2^h - 1 + 1 = 2^h$ nodes
 - **At most:** perfect binary tree of height h , i.e., $2^{h+1} - 1$ nodes
- Height h is equal to $\lfloor \log_2(n) \rfloor$

Balanced Binary Tree

- **Balanced binary tree**
 - For each node, the difference in height of the right and left sub-trees is no more than one
 - Both Perfect binary trees and complete binary trees are balanced as well
- **Completely balance binary tree**
 - Left and right sub-trees of every node have the same height
 - A perfect binary tree is completely balanced

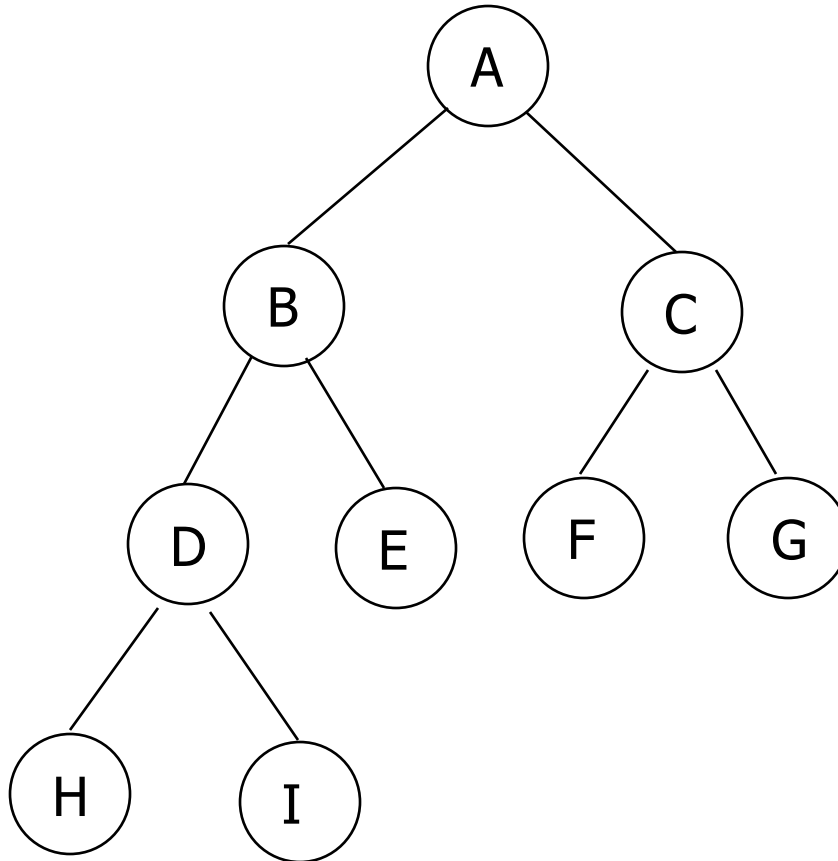
Tree ADT

Binary Tree Storage

- Contiguous storage
- Linked-list based storage

Contiguous Storage

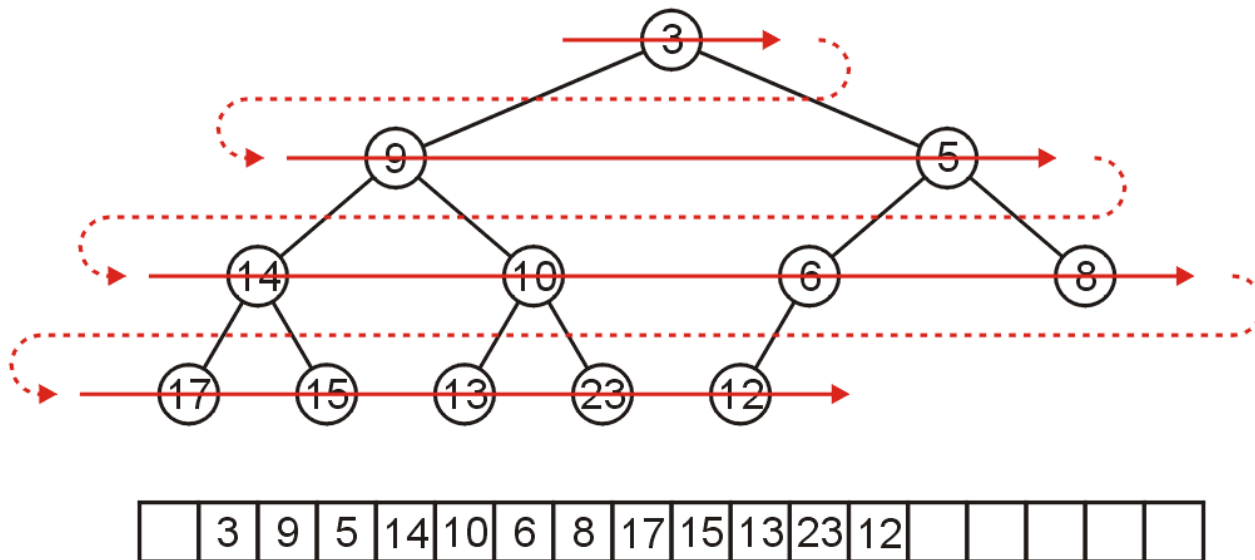
Array Storage Example (1)



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

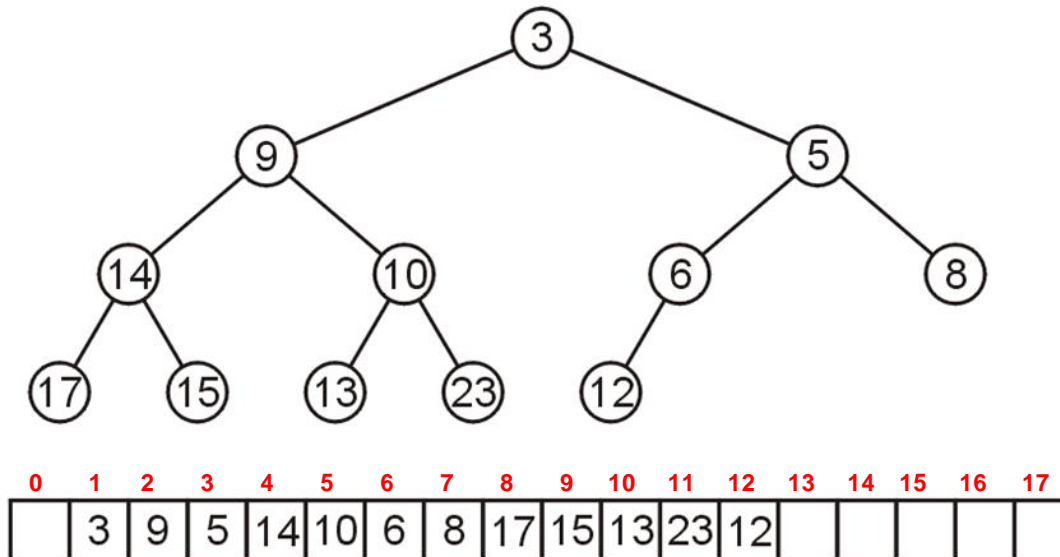
Array Storage (1)

- We can 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



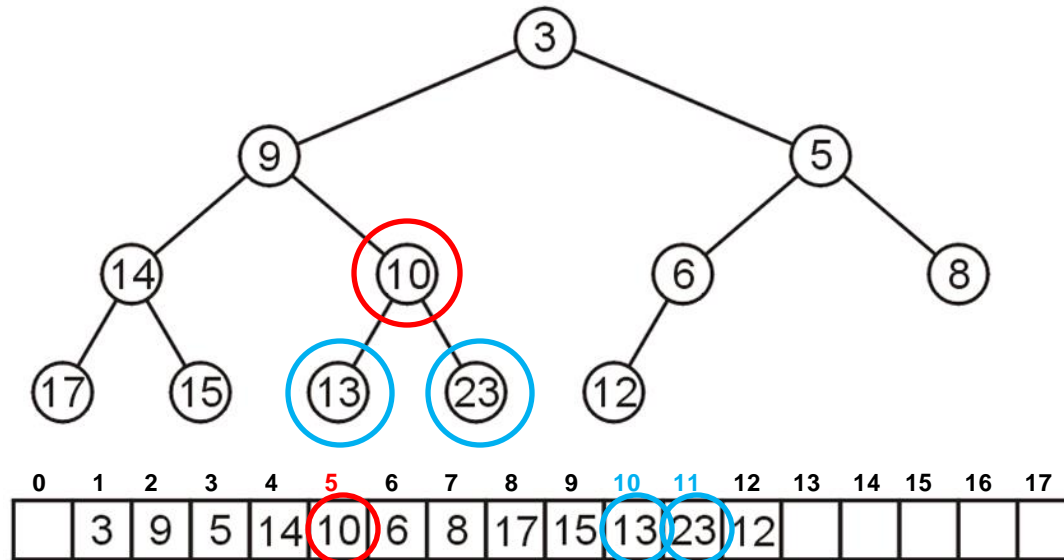
Array Storage (2)

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



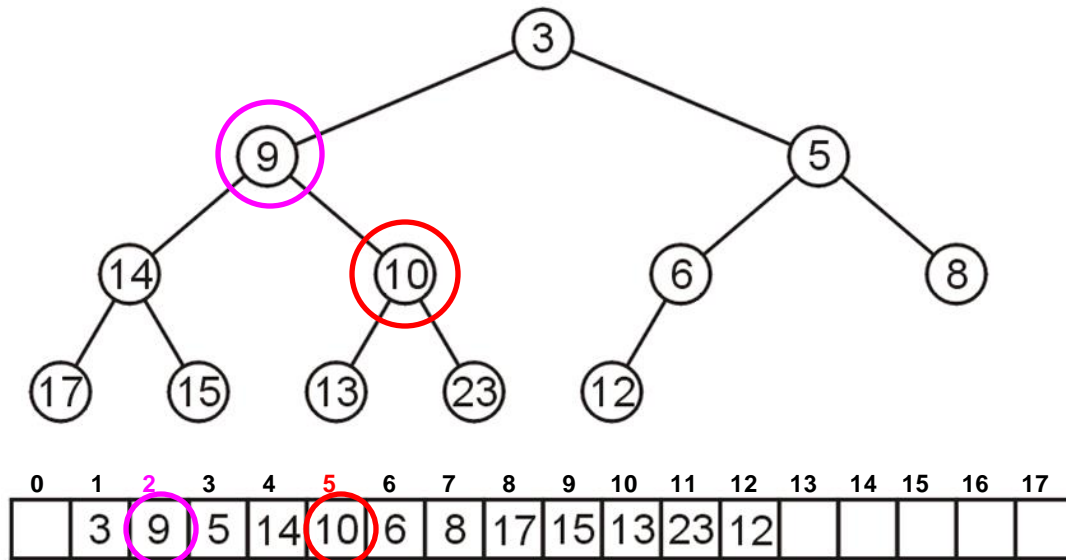
Array Storage Example (3)

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



Array Storage Example (4)

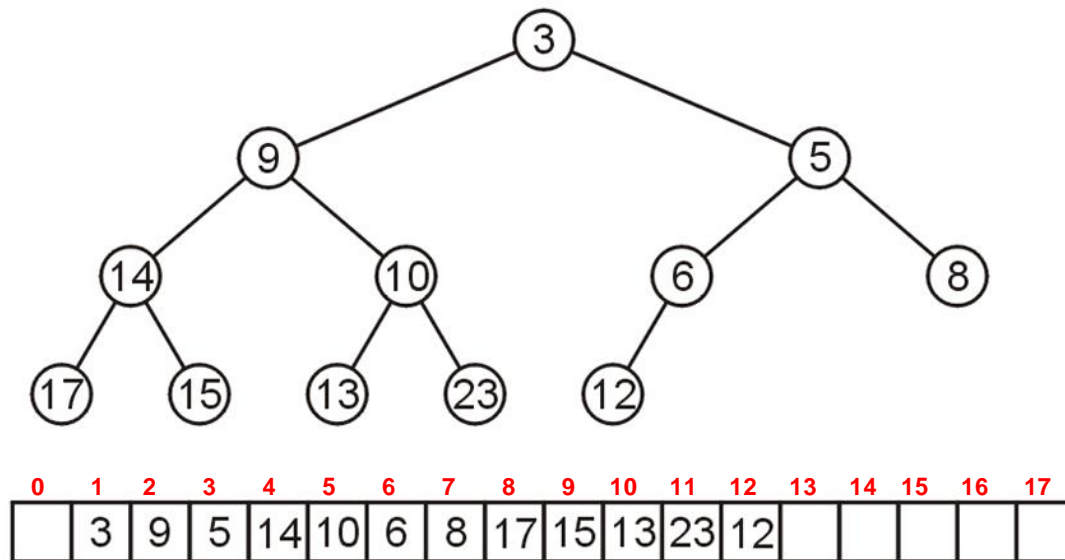
- Node 10 has index **5**
 - Its children 13 and 23 have indices **10** and **11**, respectively
 - Its parent is node 9 with index $5/2 = 2$



Array Storage (3)

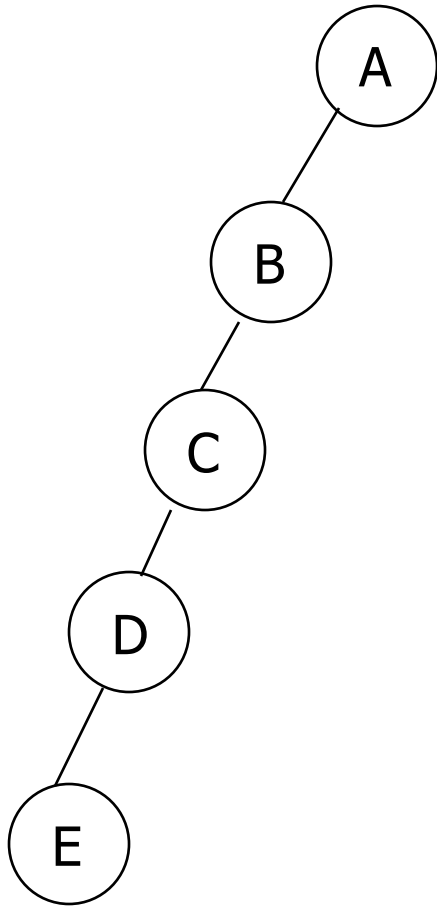
- Why array index is not started from 0
 - In C++, this simplifies the calculations

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



Array Storage Example (2)

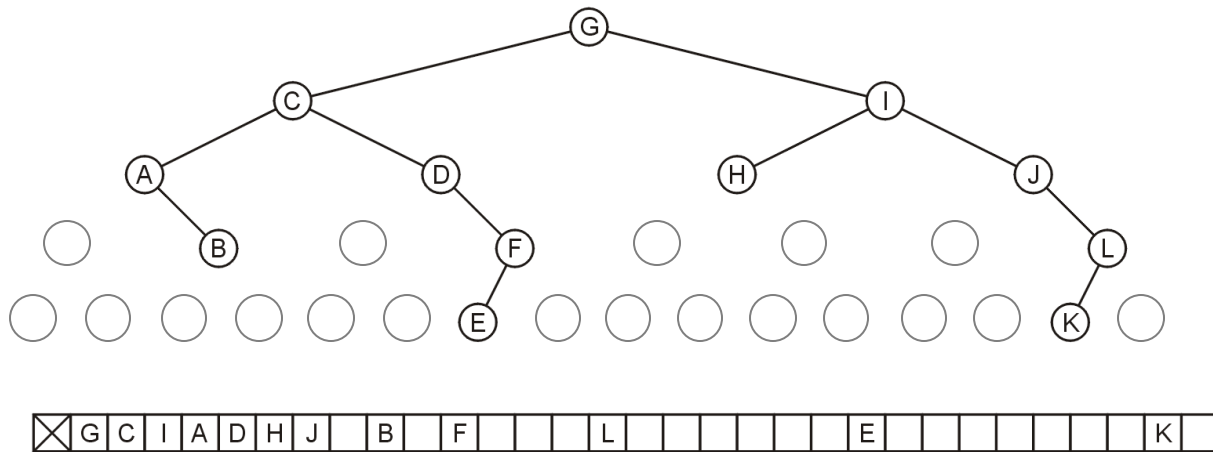
- Unused nodes in tree represented by a predefined bit pattern



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

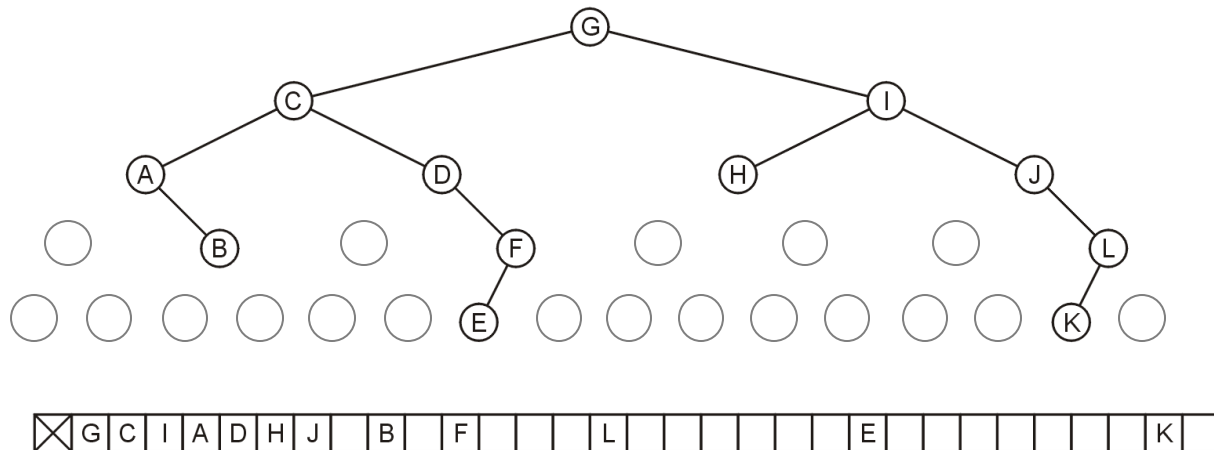
Array Storage: Disadvantage

- Why not store any tree as an **array** using breadth-first traversals?
 - Because 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?



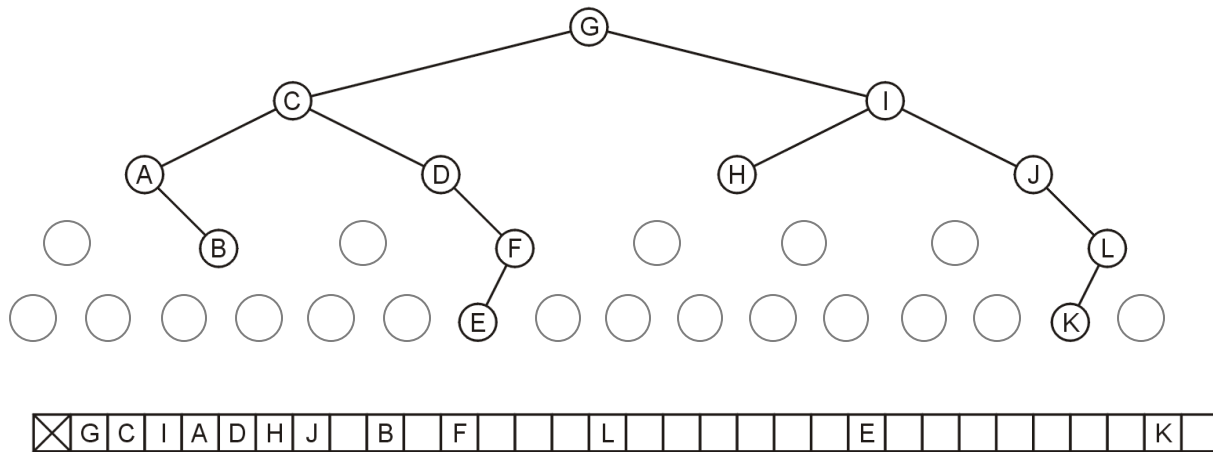
Array Storage: Disadvantage

- 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?



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**



Linked List Storage

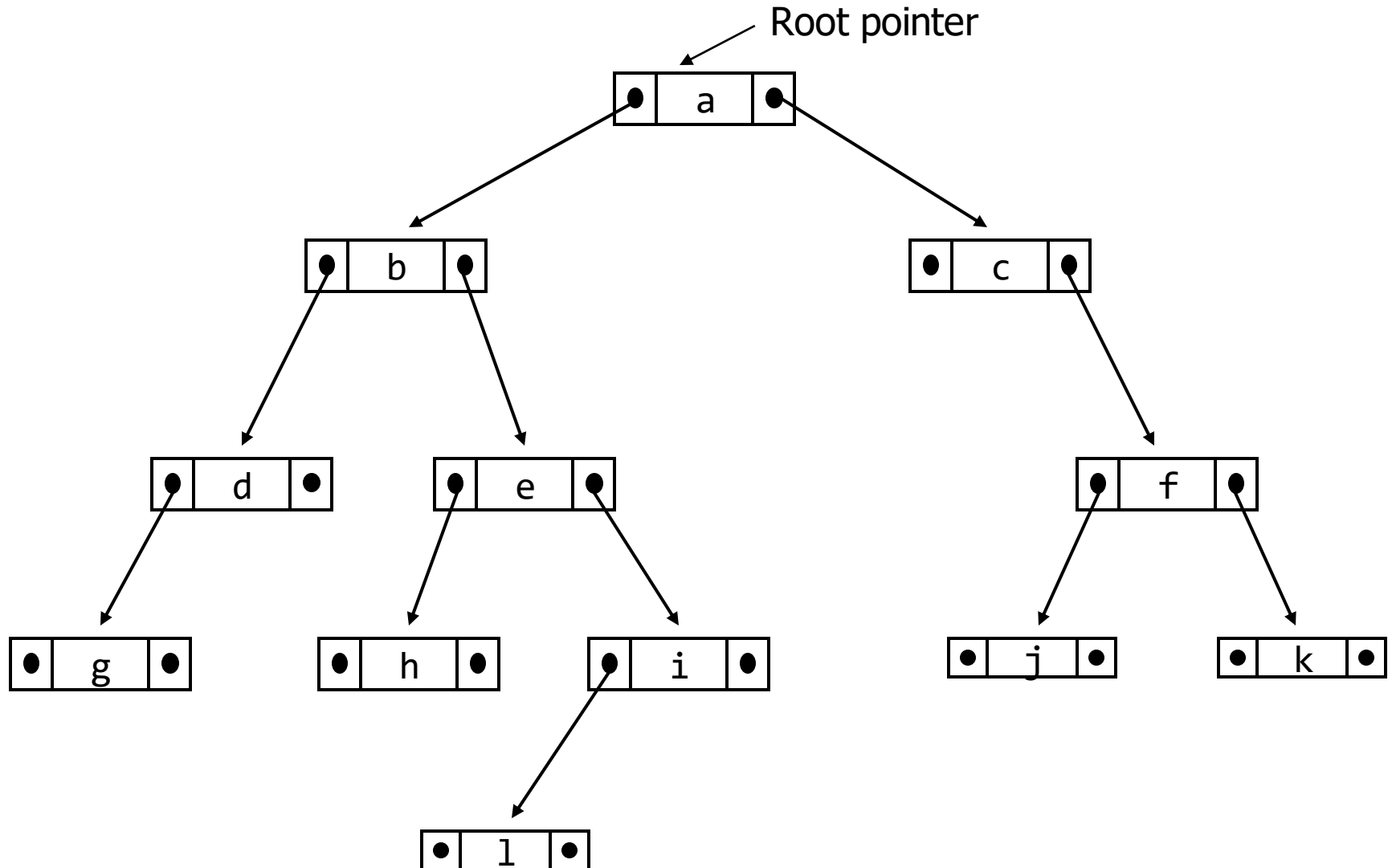
As Linked List Structure (1)

- We can implement a binary tree by using a struct which stores:
 - An element
 - A left child pointer (pointer to first child)
 - A right child pointer (pointer to second child)

```
struct 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



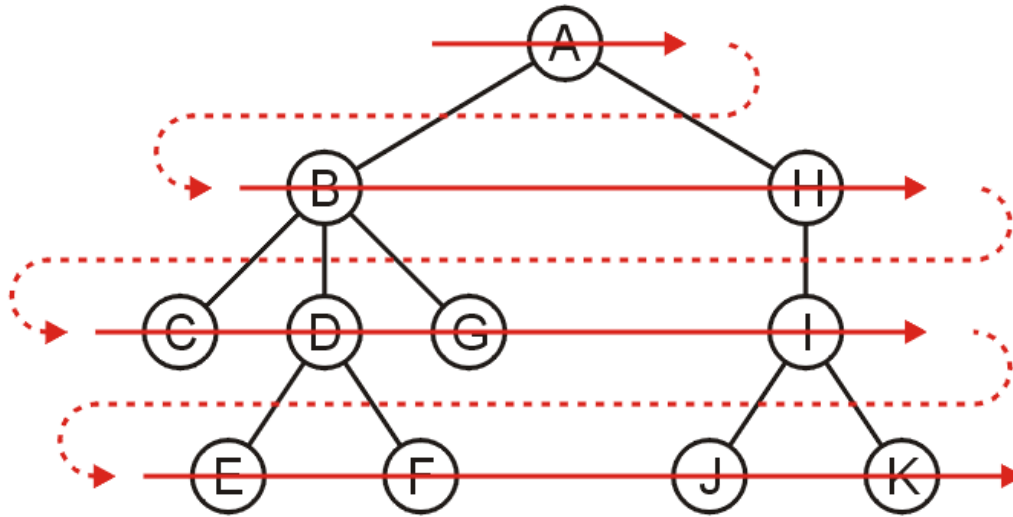
Tree Traversal

Tree Traversal

- To **traverse** (or **walk**) the tree is to visit (printing or manipulating) 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)

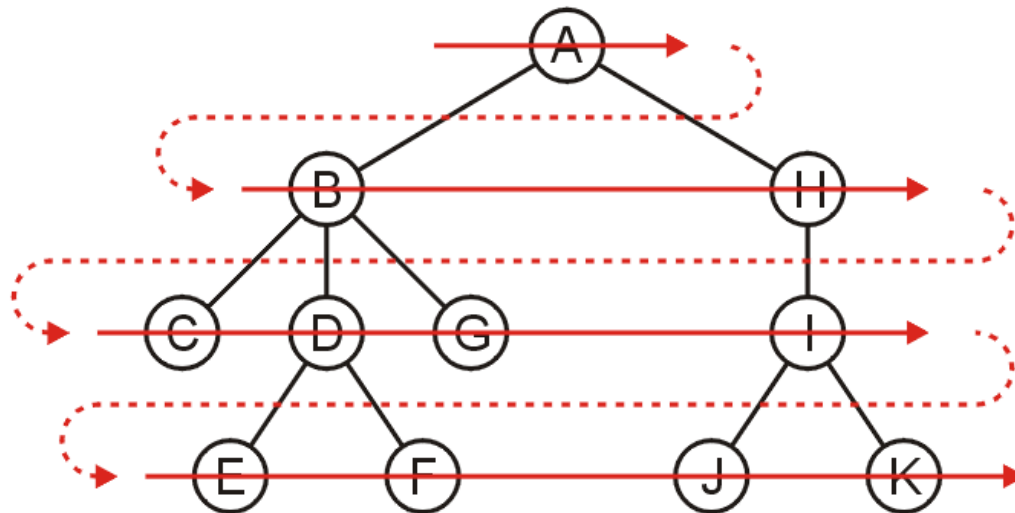
- All nodes at a given depth d are traversed before nodes at $d+1$
- Can be implemented using a queue



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

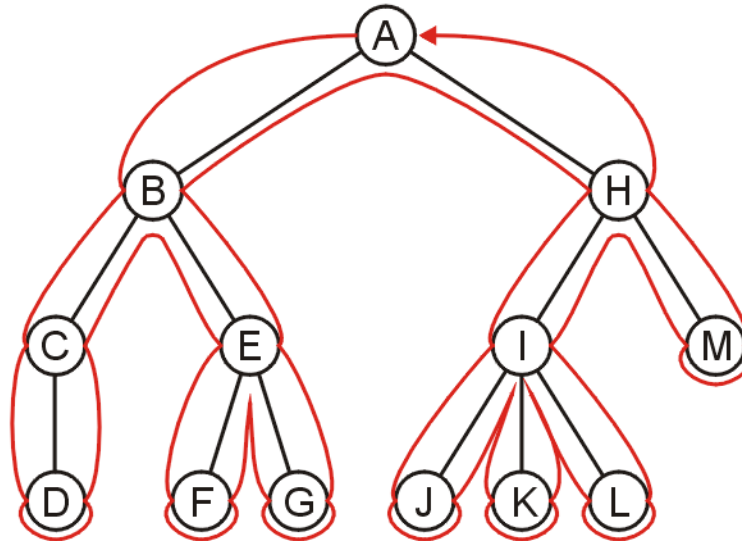
Breadth-First Traversal – Implementation

- Create a queue and push the root node onto the queue
- While the queue is not empty:
 - Enqueue all of its 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 **approached 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)

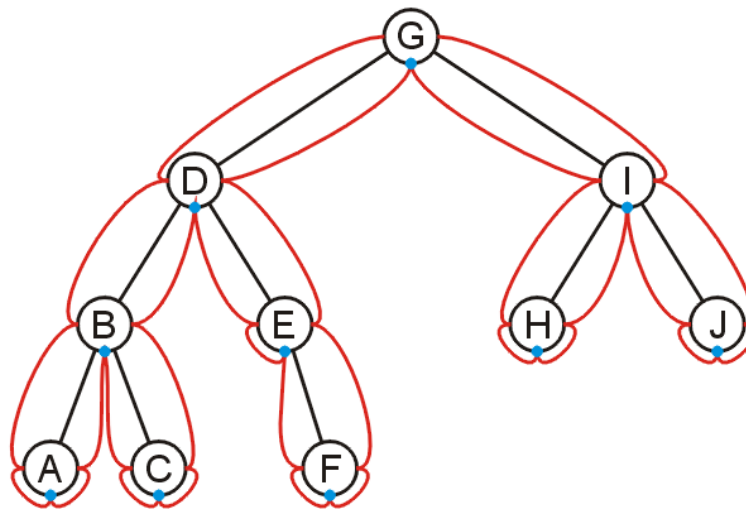


Depth-First Tree Traversal (Binary Trees)

- For each node in a binary tree, there are three choices
 - Visit the node first
 - Visit the node after left subtree
 - Visit the node after both the subtrees
- These choices lead to three commonly used traversals
 - **Preorder traversal:** visit Root (Left subtree) (Right subtree)
 - **Inorder traversal:** (Left subtree) visit Root (Right subtree)
 - **Postorder traversal:** (Left subtree) (Right subtree) visit Root

Inorder Traversal

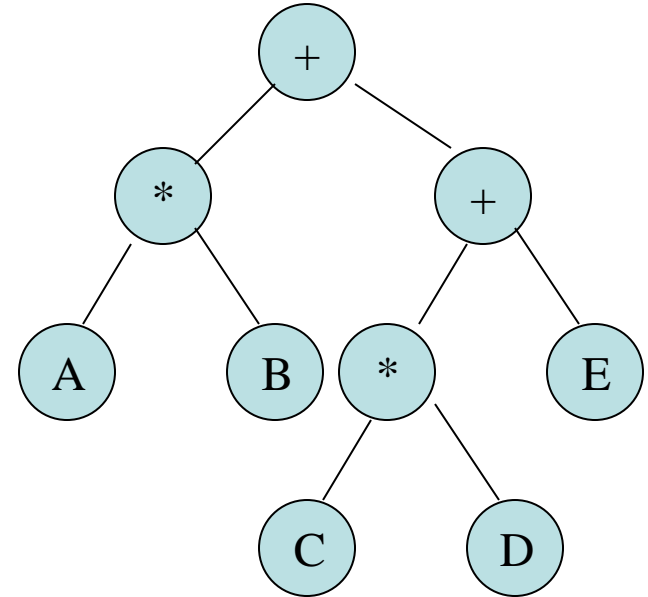
- Algorithm
 1. Traverse the left subtree in inorder
 2. Visit the root
 3. Traverse the right subtree in inorder



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);
}
```

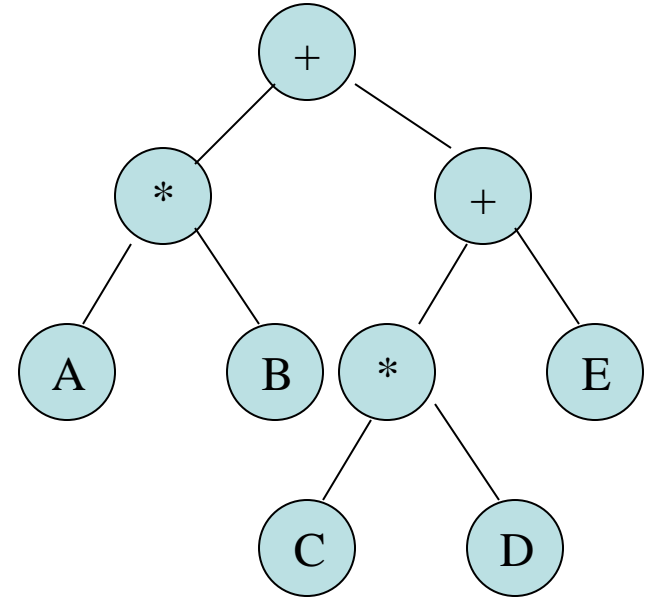
Preorder Traversal

- 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);
}
```

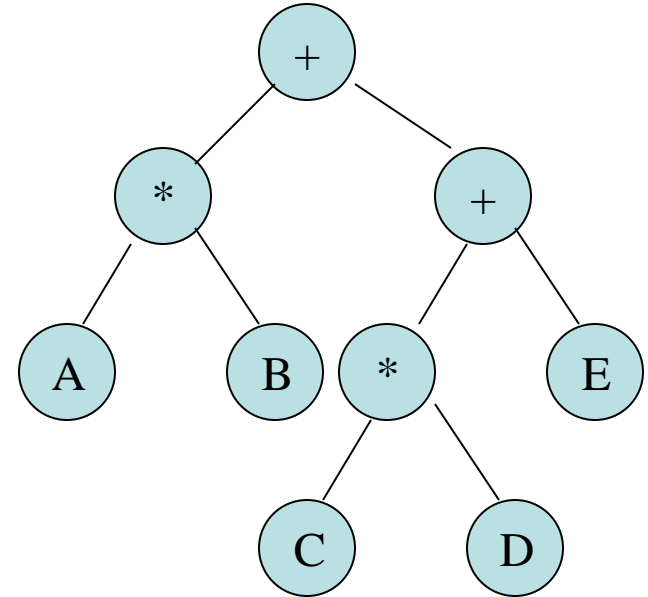
Postorder Traversal

- 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 + +



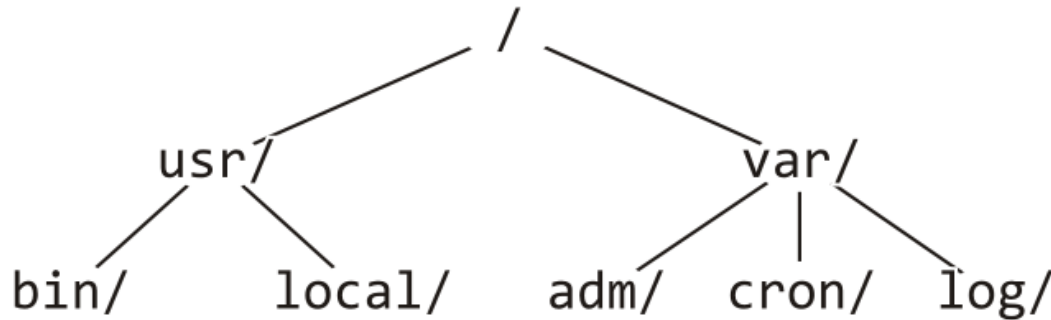
Postorder Traversal – Implementation

```
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/
```

Any Question So Far?

