CSN 102:
DATA STRUCTURES

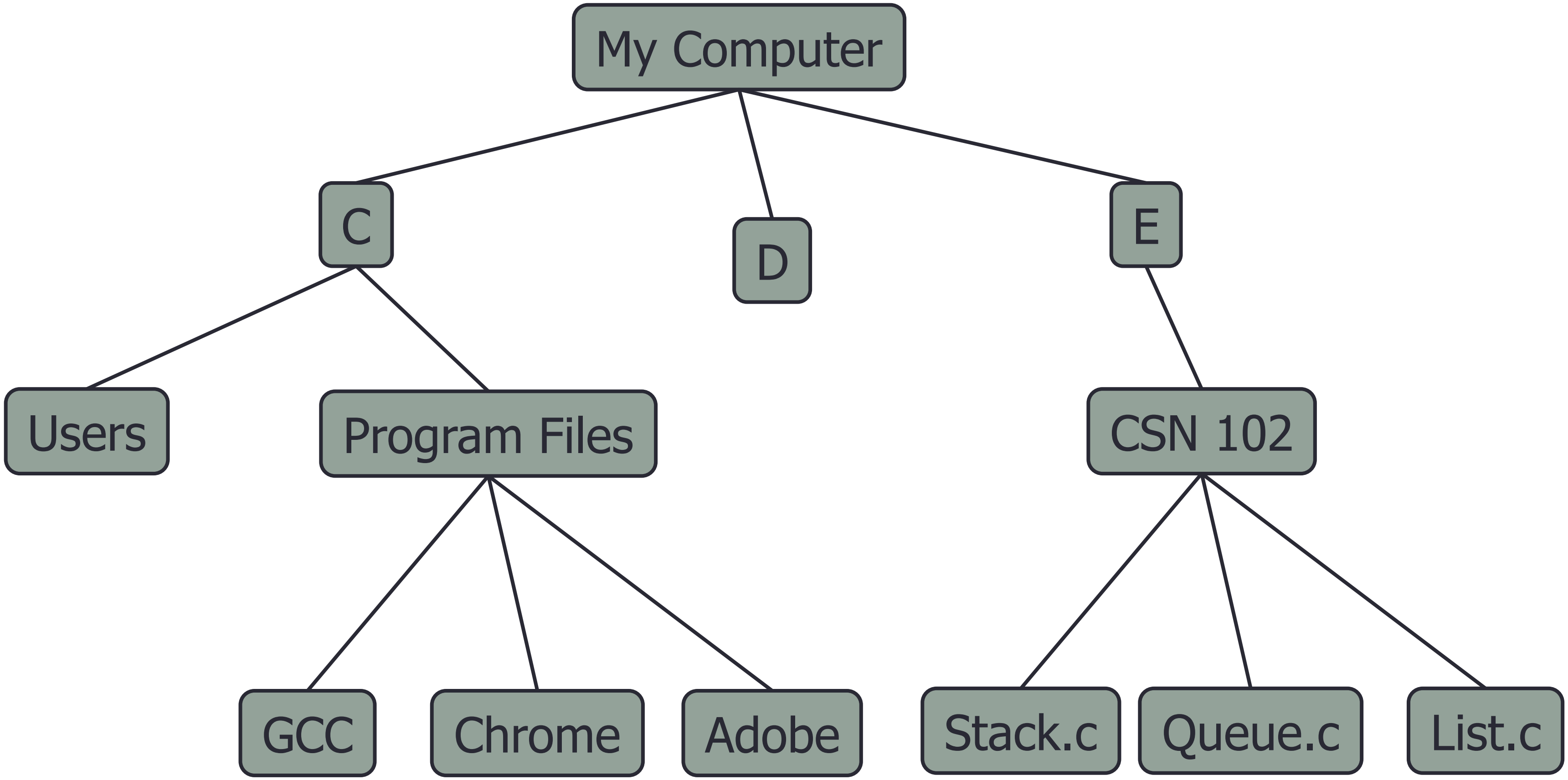
Tree: Terminology, Binary Tree, Binary Tree Traversal, Binary Search Tree, AVL tree, B Tree, Applications

What is a Tree?

- Similar to tree in general world
- Finite set of elements
- Non-linear data structure
- Used to represent hierarchical structures
- Eg. Syntax tree, Binary Search Tree, Animal Kingdom
- Application: Organization Charts, File system
- Tree can also be defined in itself as a node and list of child trees.

❖ **In Computer Science, Trees grow down! ☺**

Examples

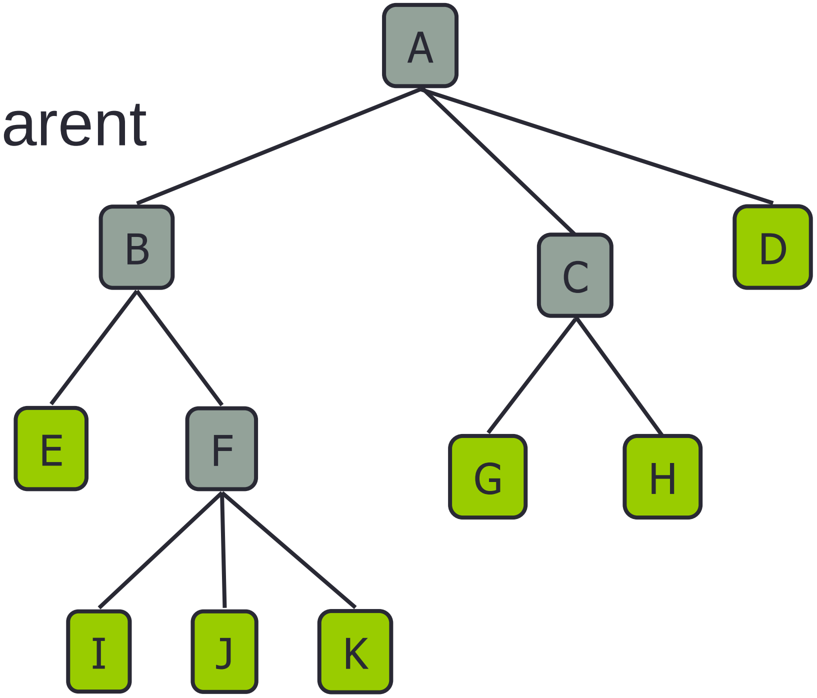


My Computer
C
D
E
Users
Program Files
CSN 102
GCC
Chrome
Adobe
Stack.c
Queue.c
List.c

Terminology

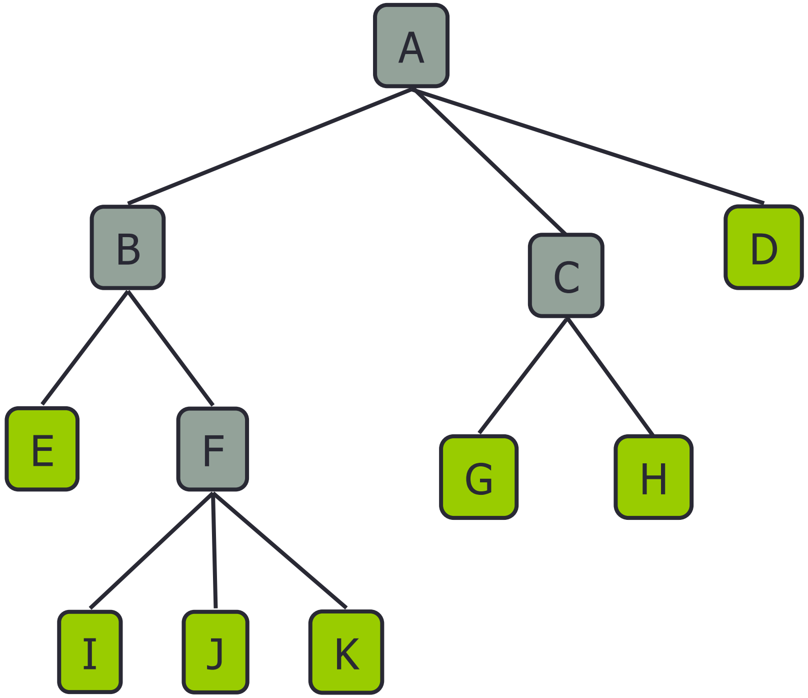
- Node: stores the key (all A, B, C… K)
- Child node: node directly connected below the parent node (B is child of A)
- Parent node: node directly connected above the child node (B is parent of E)
- Siblings: nodes sharing same parent

  (E and F are siblings)



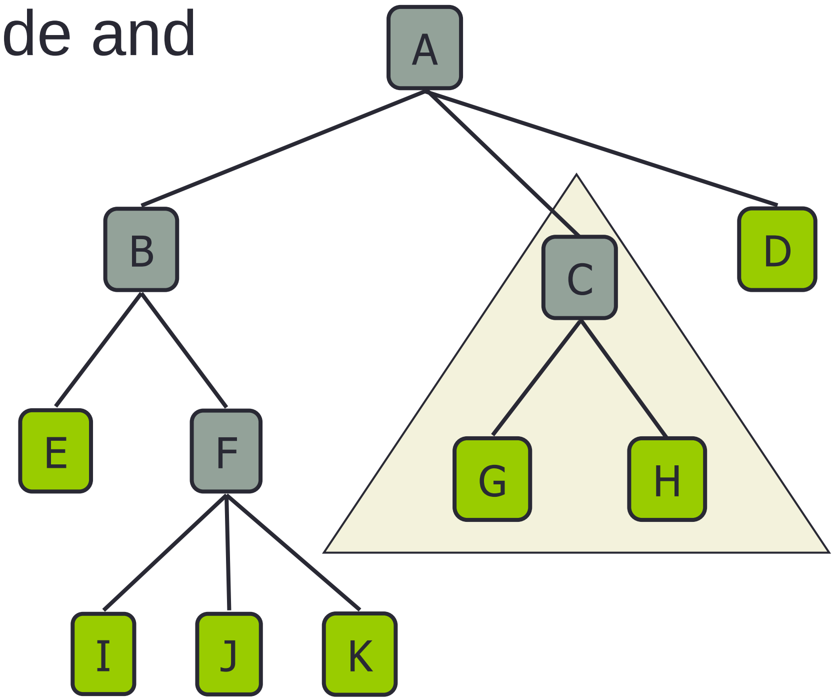
Terminology

- Root: top node in tree or node with no parent (A is root)
- Leaf: node with no child (E, I, J etc. are leaf node)
- Interior node: All non-leaf node(B, F, A etc.)



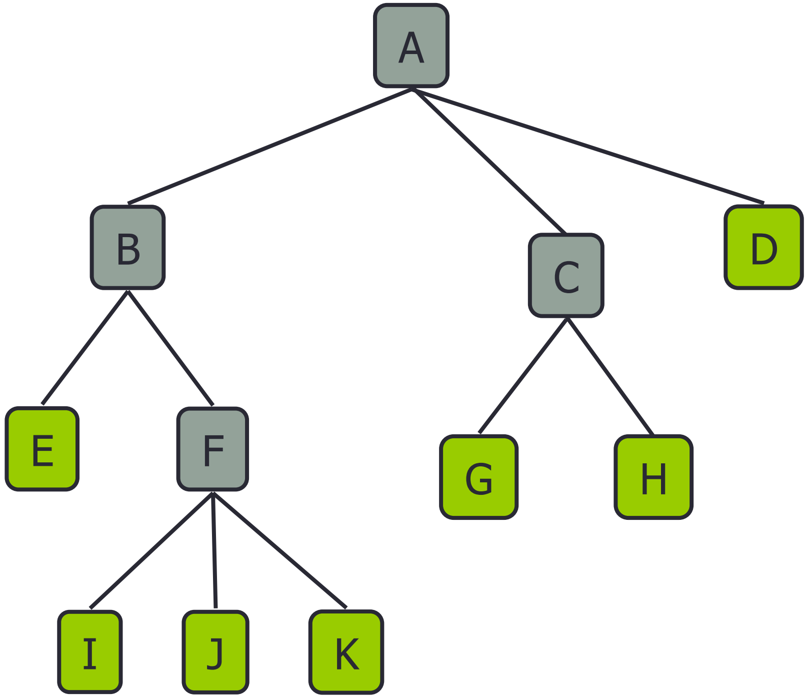
Terminology

- Ancestors: parent, parent of parent and so on. (F, B, A are ancestors of K)
- Descendants: child, child of child and so on. (E, F, I, J, K are descendants of B)
- Subtree: tree consisting of a node and its descendants



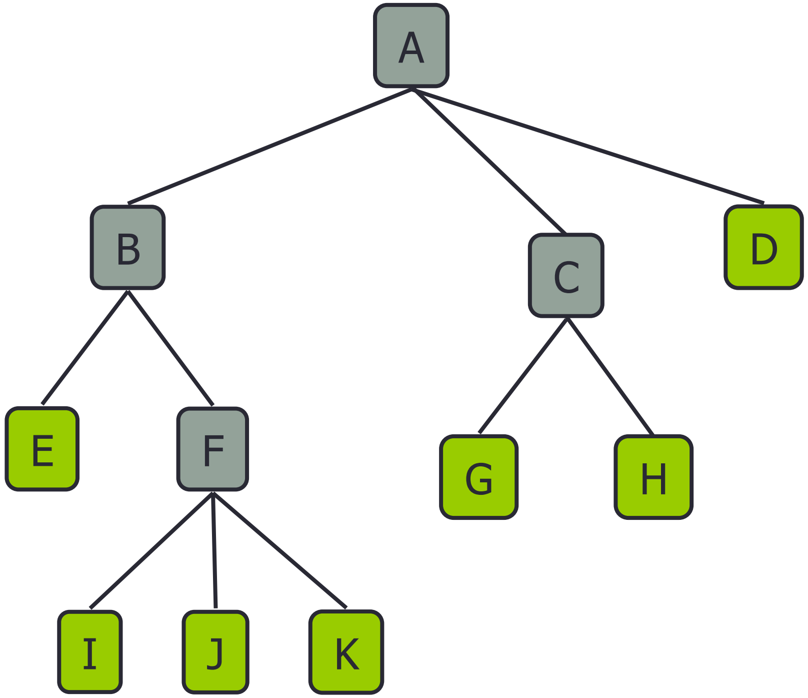
Terminology

- Degree of a node: number of its child (degree(A)=3, degree(B)=2)
- Degree of tree: maximum of degrees of all nodes
- Level: 1 + number of connection between node and root (level(F)=2)



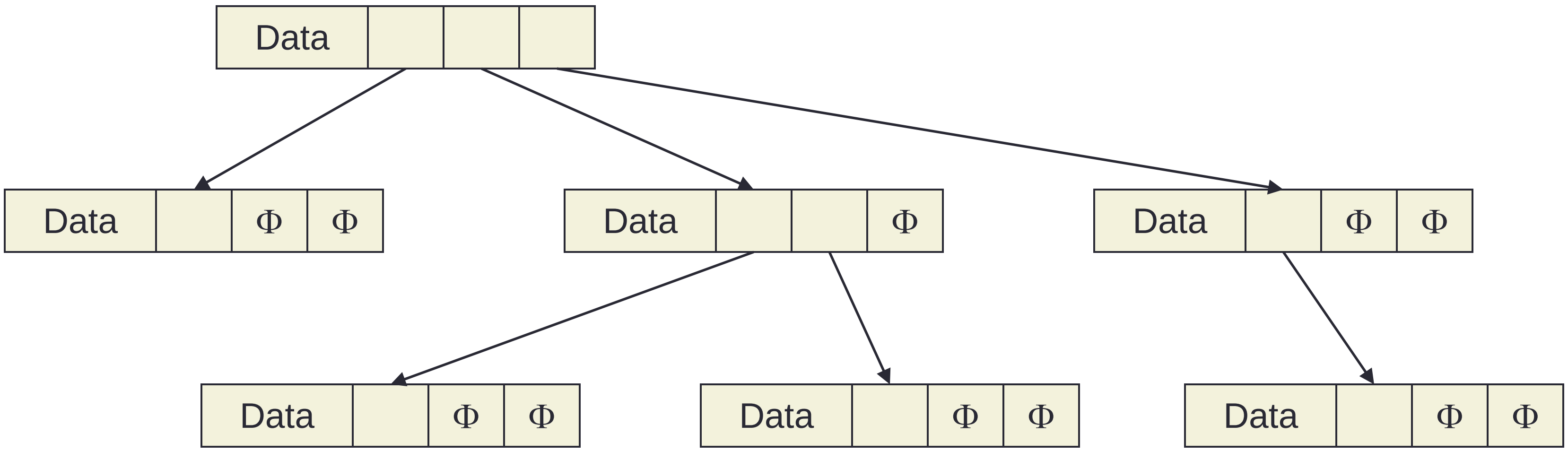
Terminology

- Height of node: number of edges between node and farthest leaf (Height(B)=2)
- Height of tree: height of root node (height of tree is 3)
- Depth of node: number of edges between root and node (depth of K is 3)



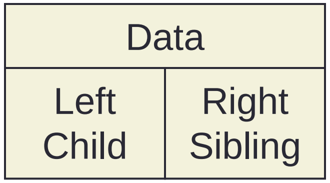
Tree Representation

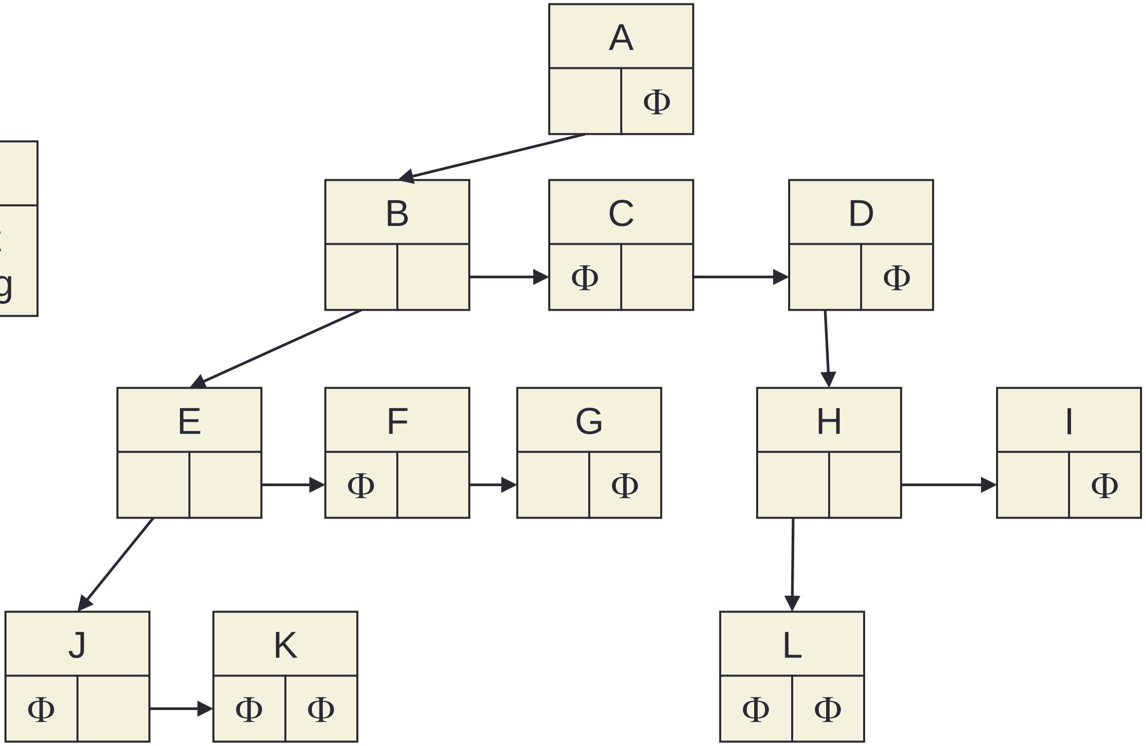
- Every node contains:
  - Key/data
  - Children nodes
  - Parent node(optional)



Left Child, Right Sibling Representation

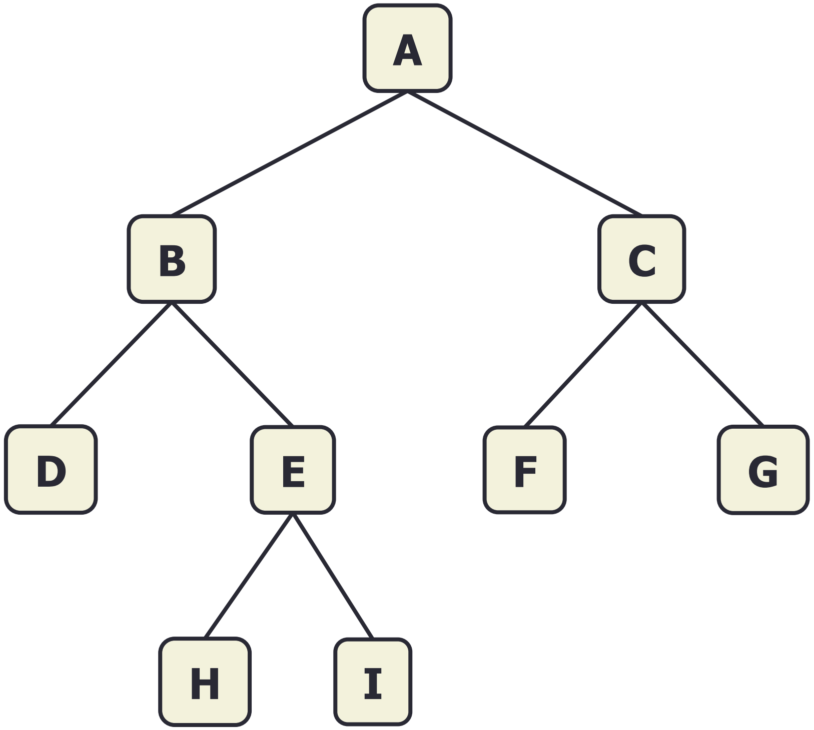
- Every node contains
  - Key/data
  - Pointer to left child and right sibling





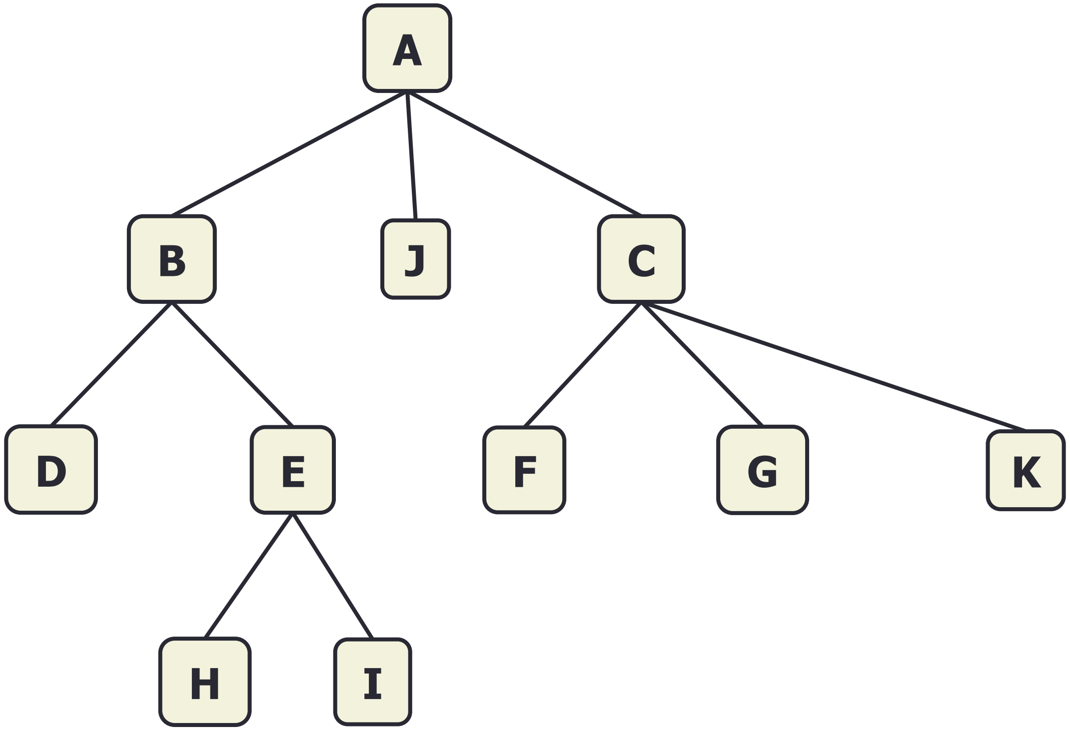
Binary Tree

- Each node has at most two children
- The children of a node are ordered pair (a left and a right child)
- Each node contains:
  - Key
  - Left
  - Right
  - Parent(optional)



K-ary Tree

- Each node has at most **K** children
- Binary tree is a special case with K=2
- Eg. 3-ary tree



Breadth First Traversal

- Traverse all the nodes at level ***i*** before progressing to level ***i+1*** starting from root node
- Add nodes in the queue as soon as their parent is visited.
- In each iteration, delete one element from queue and mark visited

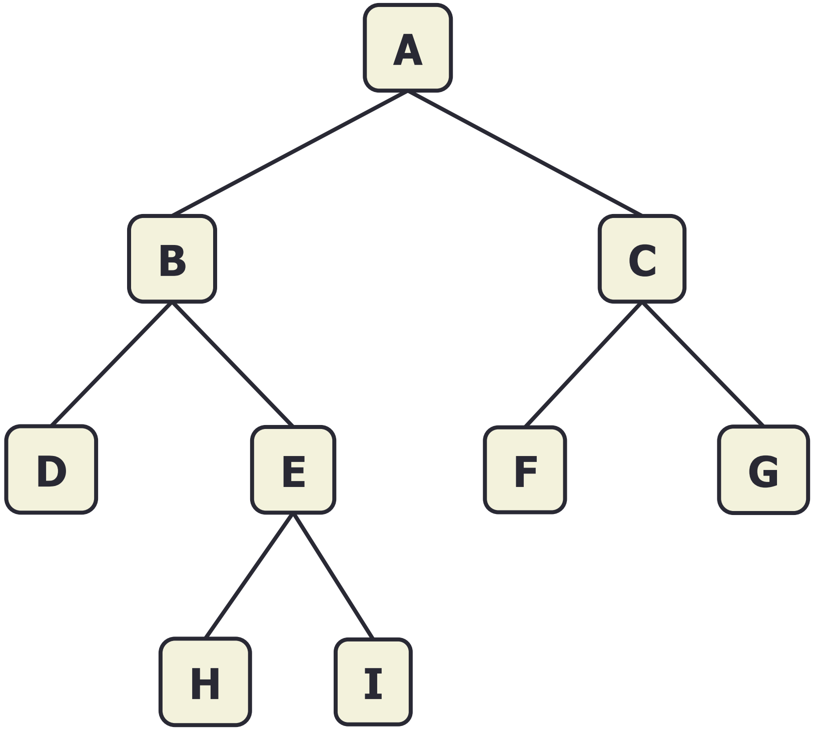
BFS Algorithm

```
BFS(Tree) {
        if (!isEmpty(Tree)) enqueue(Q, root);
        while (!isEmpty(Q)) {
                node = dequeue(Q);
                print(node->data);
                if (node->left != NULL) enqueue(Q,node->left);
                if (node->right != NULL) enqueue(Q,node->right);
        }
}
```

BFS Example

Output:

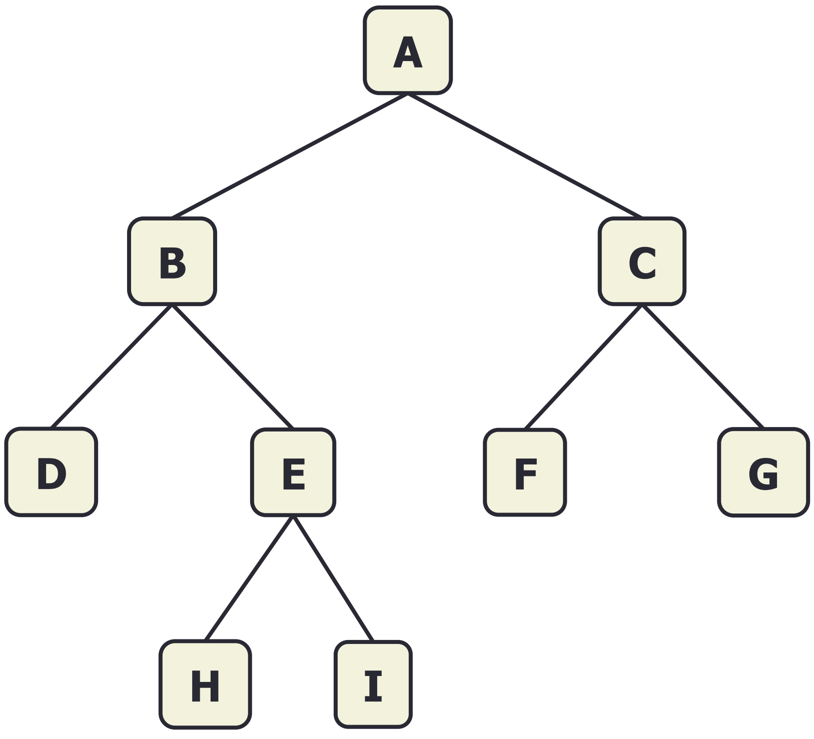
Queue(Q): A



BFS Example

Output: A

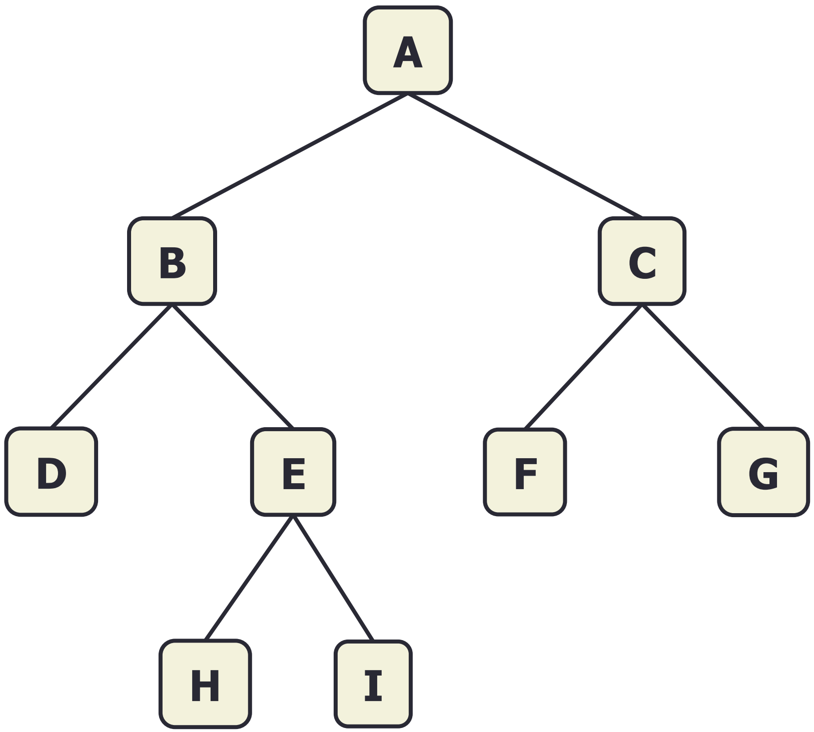
Queue(Q): B, C



BFS Example

Output: A B

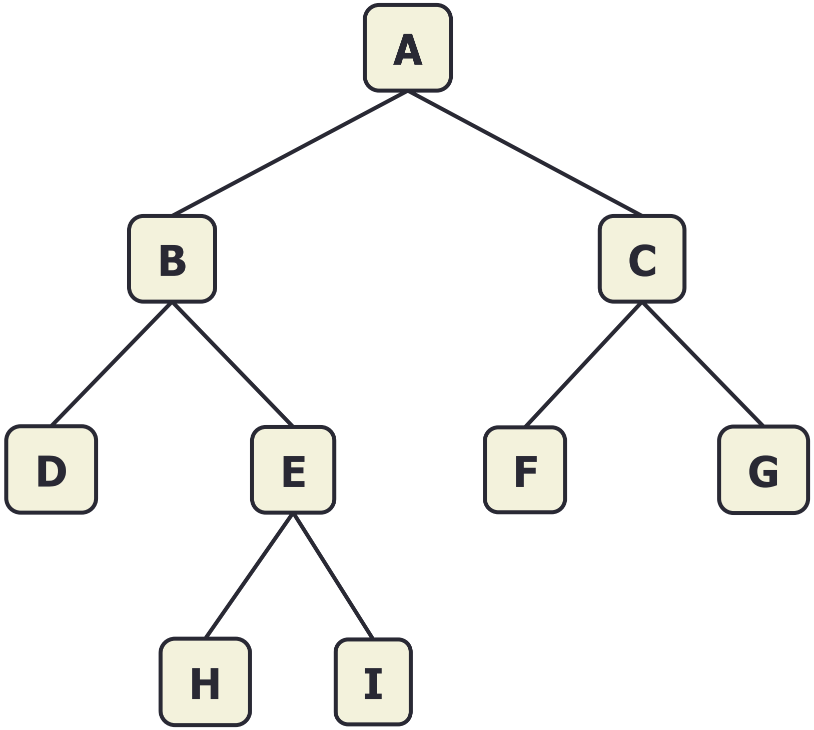
Queue(Q): C, D, E



BFS Example

Output: A B C

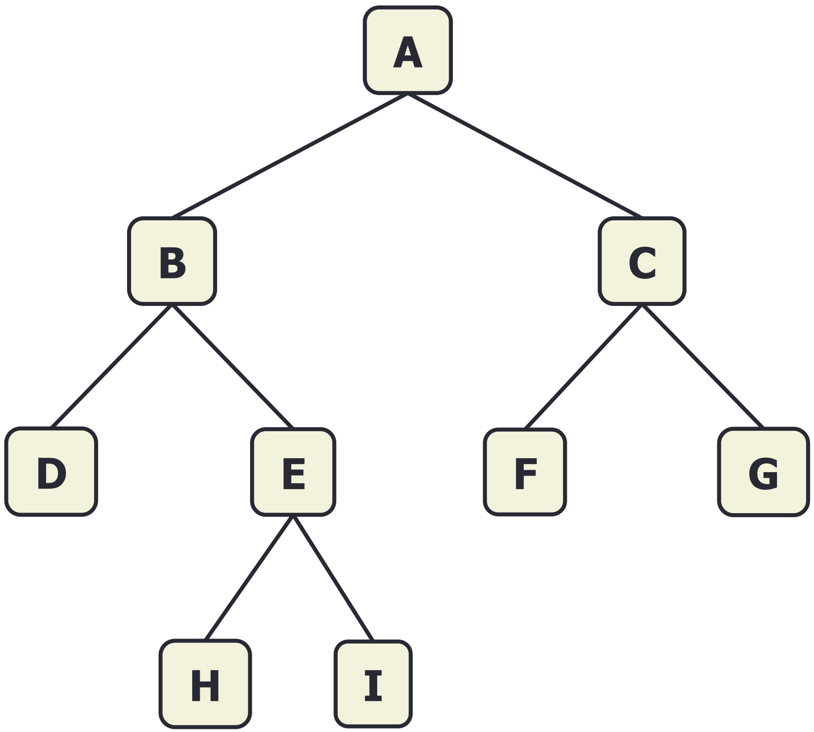
Queue(Q): D, E, F, G



BFS Example

Output: A B C D

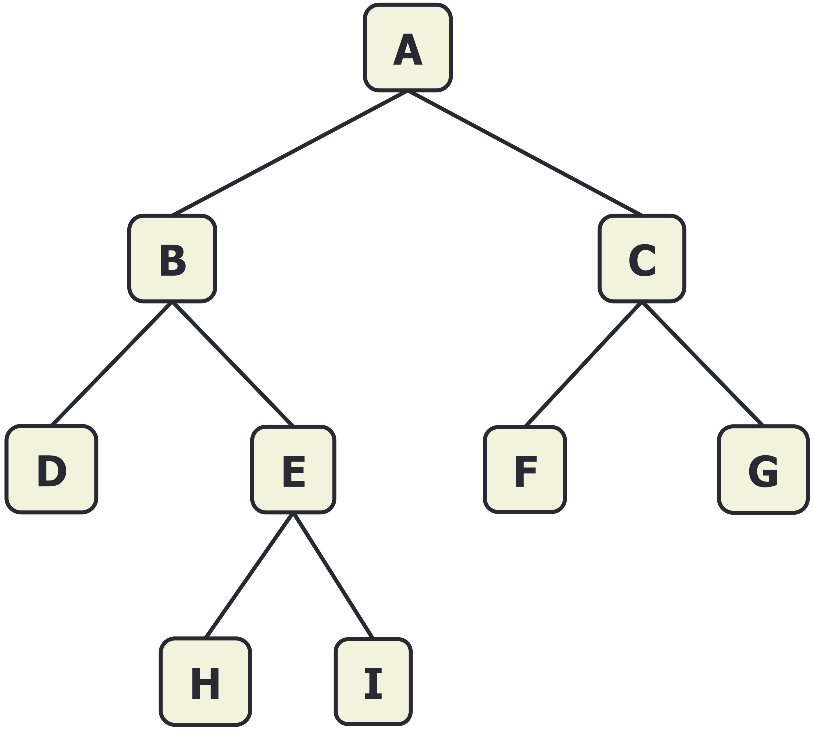
Queue(Q): E, F, G



BFS Example

Output: A B C D E

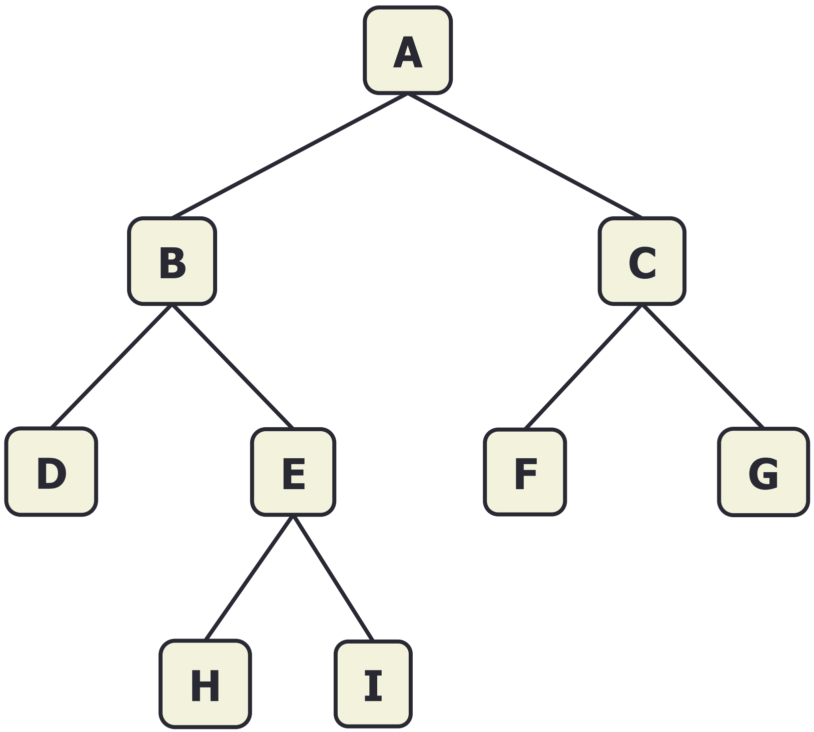
Queue(Q): F, G, H, I



BFS Example

Output: A B C D E F

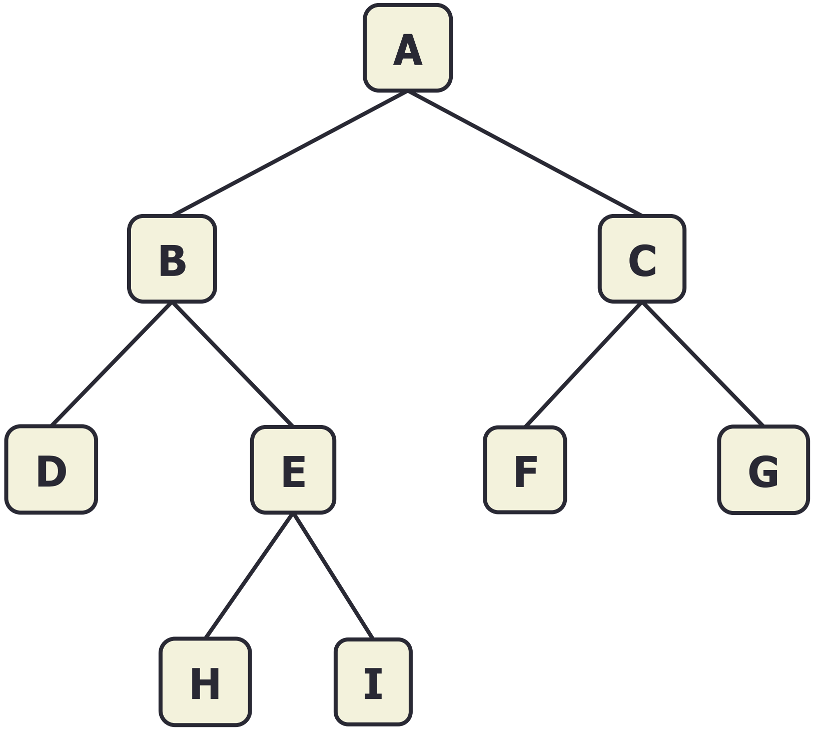
Queue(Q): G, H, I



BFS Example

Output: A B C D E F G

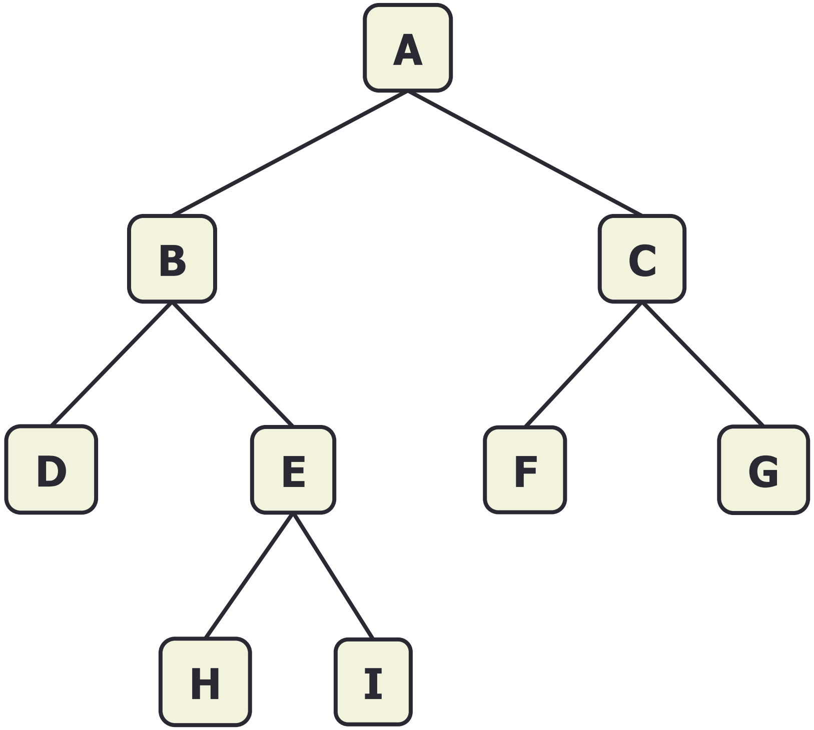
Queue(Q): H, I



BFS Example

Output: A B C D E F G H

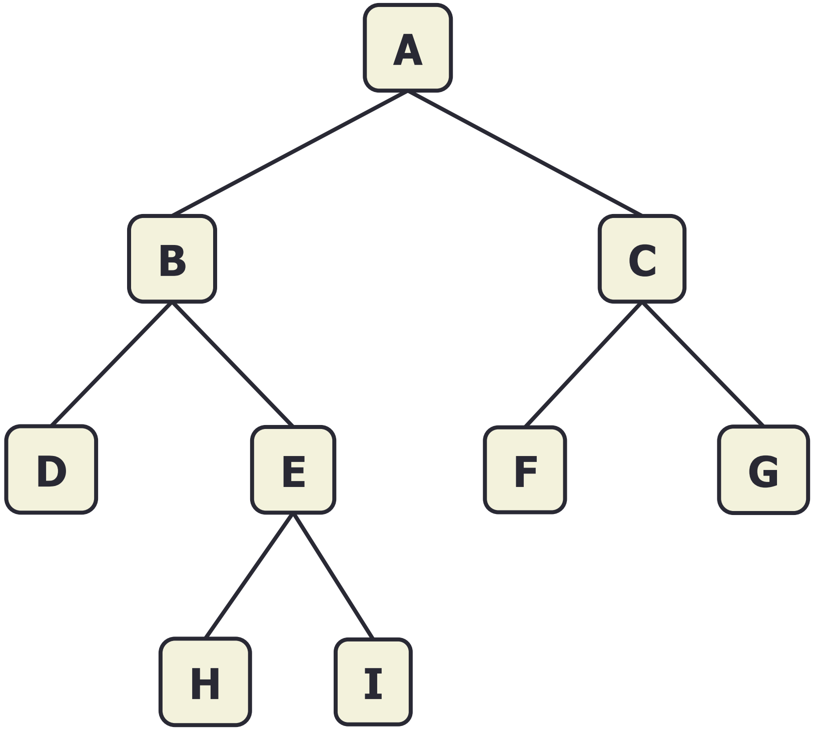
Queue(Q): I



BFS Example

Output: A B C D E F G H I

Queue(Q):



Depth First Search

- Travel All the nodes of one sub-tree of binary search before travelling other sub-tree
- DFS is recursively implemented on a tree to visit nodes
- **Note: Many of the tree algorithms are recursively implemented for the reason that tree itself is implemented recursively**
- DFS on binary tree can be implemented in 3 ways
  - PreOrder: Root-Left-Right
  - InOrder: Left-Root-Right
  - PostOrder: Left-Right-Root

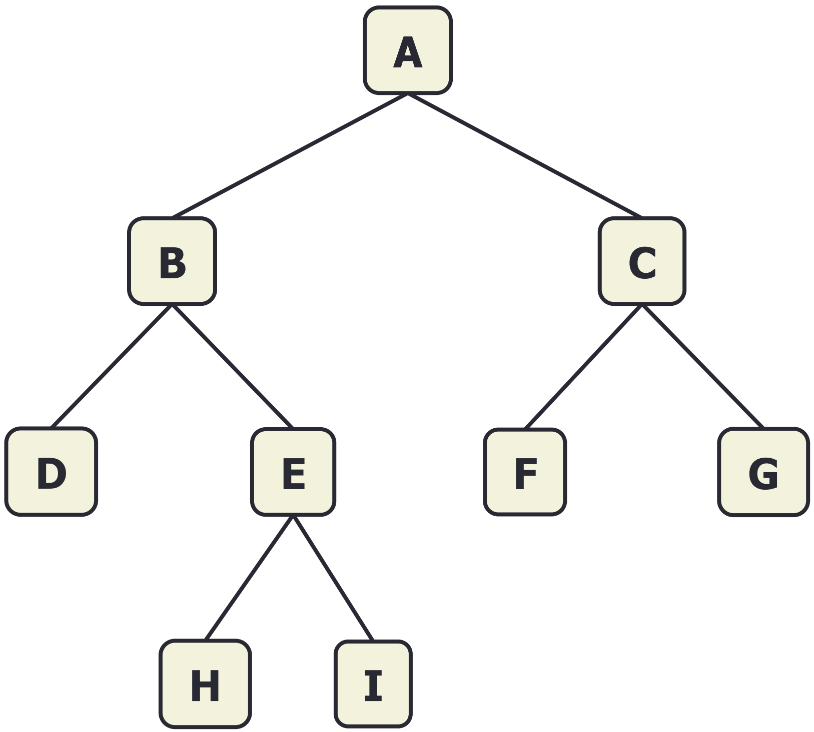
PreOrder Traversal

- Visit the root node first
- Visit left sub-tree in PreOrder
- Visit right sub-tree in PreOrder

```
PreOrderTraversal(Tree) {
        if (isEmpty(Tree)) return;
        else {
                print (tree->data);
                PreOrderTraversal(tree->left);
                PreOrderTraversal(tree->right);
        }
}
```

PreOrder Traversal: Example

Output: A B D E H I C F G



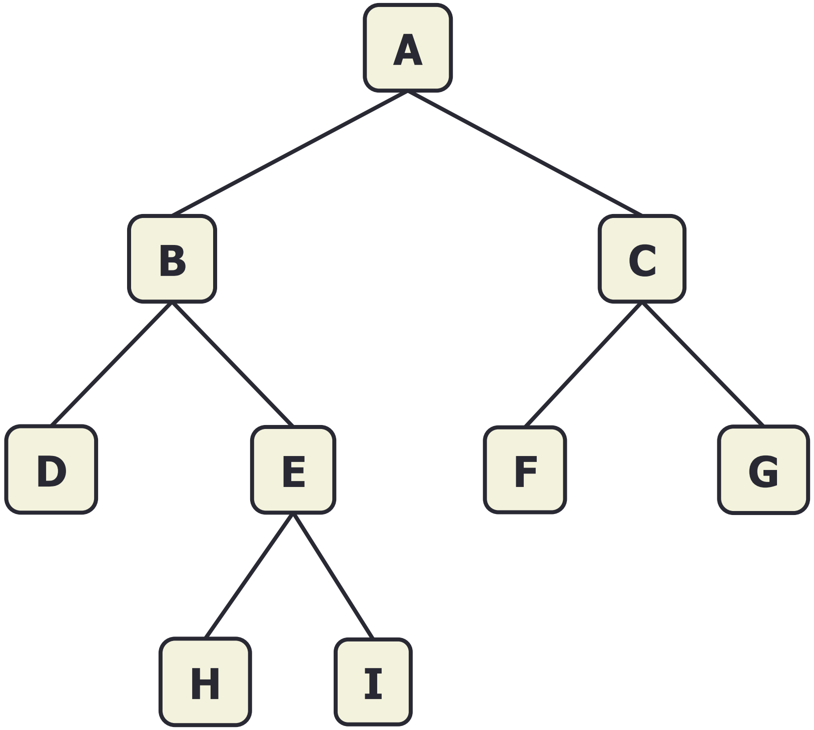
InOrder Traversal

- Visit the left sub-tree in InOrder
- Visit root node
- Visit right sub-tree in InOrder

```
InOrderTraversal(Tree) {
        if (isEmpty(Tree)) return;
        else {
                InOrderTraversal(tree->left);
                print (tree->data);
                InOrderTraversal(tree->right);
        }
}
```

InOrder Traversal: Example

Output: D B H E I A F C G



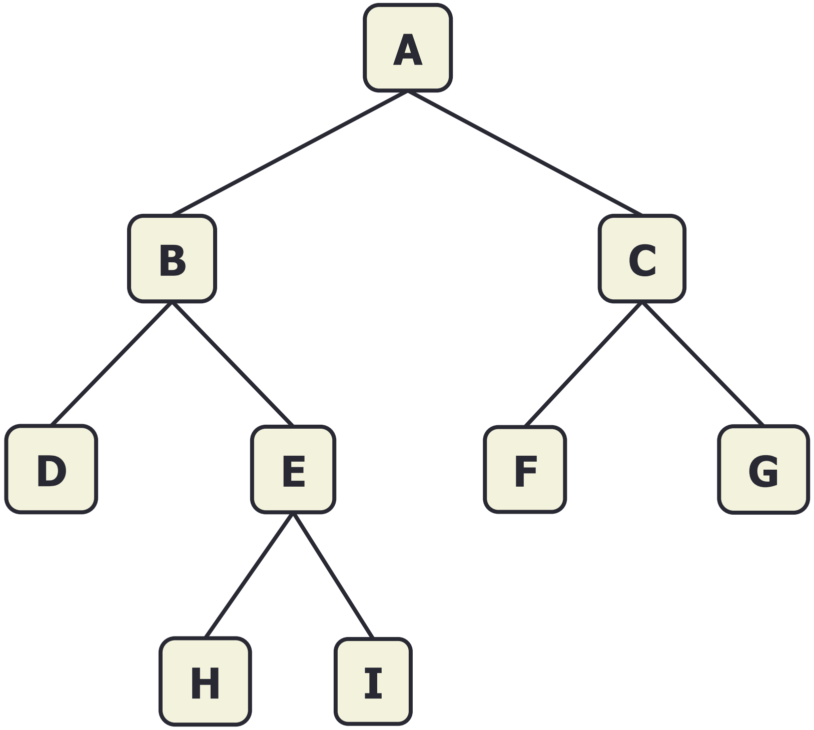
PostOrder Traversal

- Visit the left sub-tree in PostOrder
- Visit right sub-tree in PostOrder
- Visit root node

```
PostOrderTraversal(Tree) {
        if (isEmpty(Tree)) return;
        else {
                PostOrderTraversal(tree->left);
                PostOrderTraversal(tree->right);
                print (tree->data);
        }
}
```

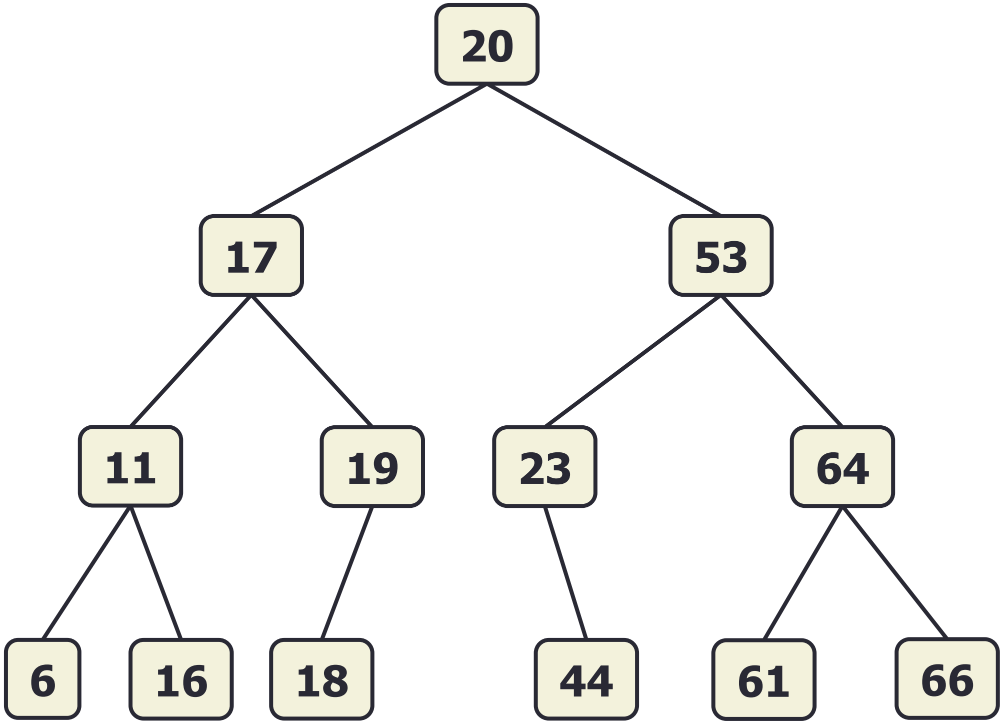
PostOrder Traversal: Example

Output: D H I E B F G C A



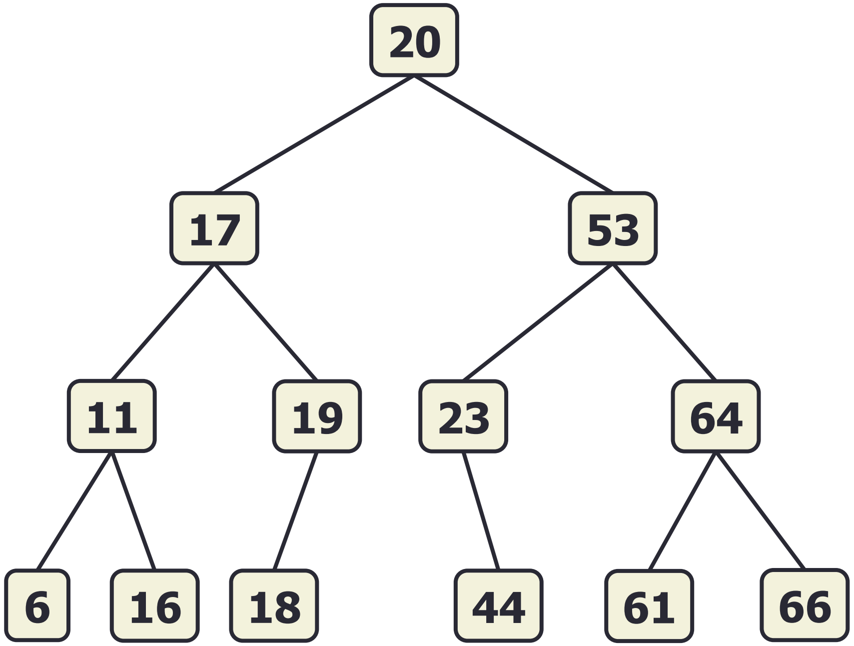
Exercise

- What will be the order of BFS, PreOrder, InOrder and PostOrder traversals on below tree?



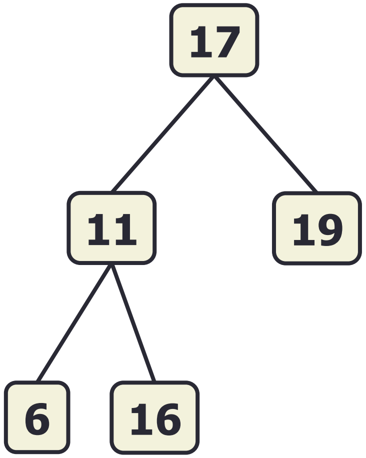
Exercise: Solution

BFS:	20 17 53 11 19 23 64 6 16 18 44 61 66
PreOrder:	20 17 11 6 16 19 18 53 23 44 64 61 66
InOrder:	6 11 16 17 18 19 20 23 44 53 61 64 66
PostOrder:	6 16 11 18 19 17 44 23 61 66 64 53 20

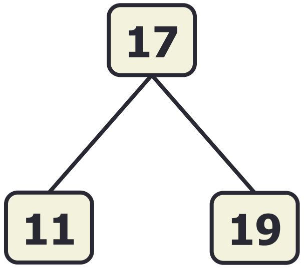


Full Binary Tree

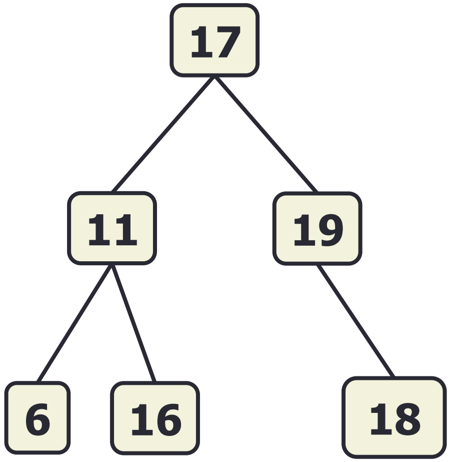
- Either a node has 2 children or no child in a binary tree



Full Binary Tree



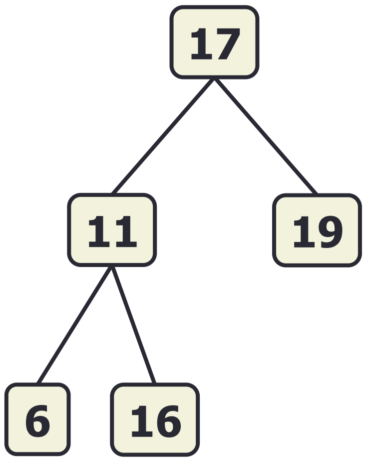
Full Binary Tree

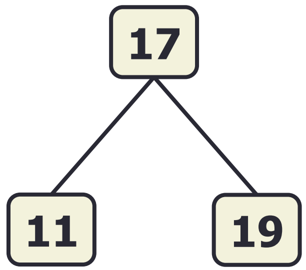


Not Full Binary Tree

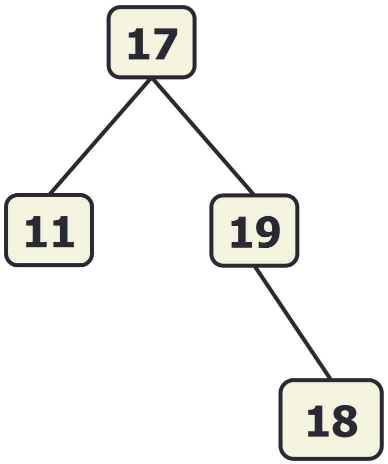
Perfect Binary Tree

- All internal nodes has two children and all leaves are at same level/depth





Full; perfect



full; **not perfect**

Not full; not perfect

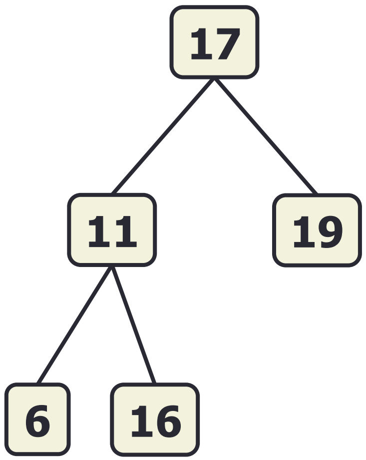
Perfect Binary Tree

- Level i has 2^i nodes
- If leaves are level h
  - Number of leaves is 2h
  - Number of internal node = $1+2+2^2+2^3+...+ 2^{h-1} = 2^h – 1$
    $= \text{number of leaves} - 1$
  - Total number of nodes = $2^h + 2^h – 1 = 2^{h+1} – 1$

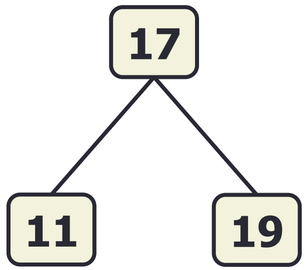
- If total number of nodes is n
  - Number of leaves = $(n+1)/2$
  - Height of tree = $\log_2$ (number of leaves) = $\log_2(n+1)/2$

Complete Binary Tree

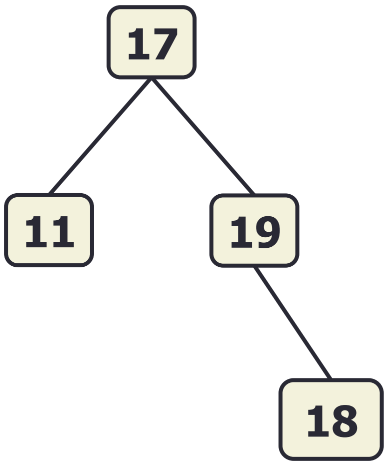
- All levels are completely filled except last. Also, all nodes in last level are as far left as possible



Full;
Not perfect;
Complete



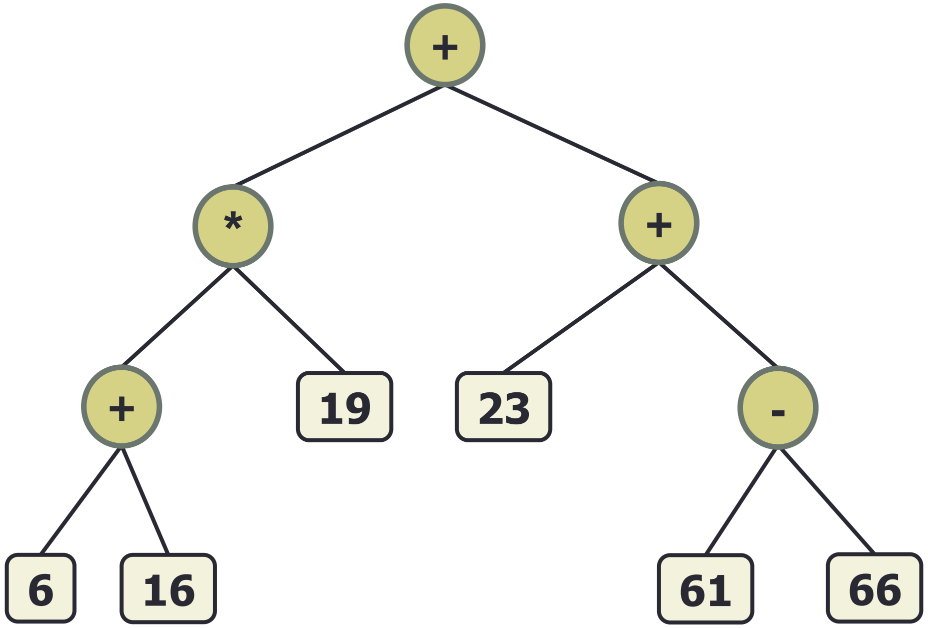
Full;
Perfect;
Complete



Not full;
Not perfect;
Not Complete

Example : Expression Tree

- An arithmetic expression can be represented as binary tree where internal nodes are operators and leaves are operands
- Eg. ((6+16) * 19) + (23 + (61-66))

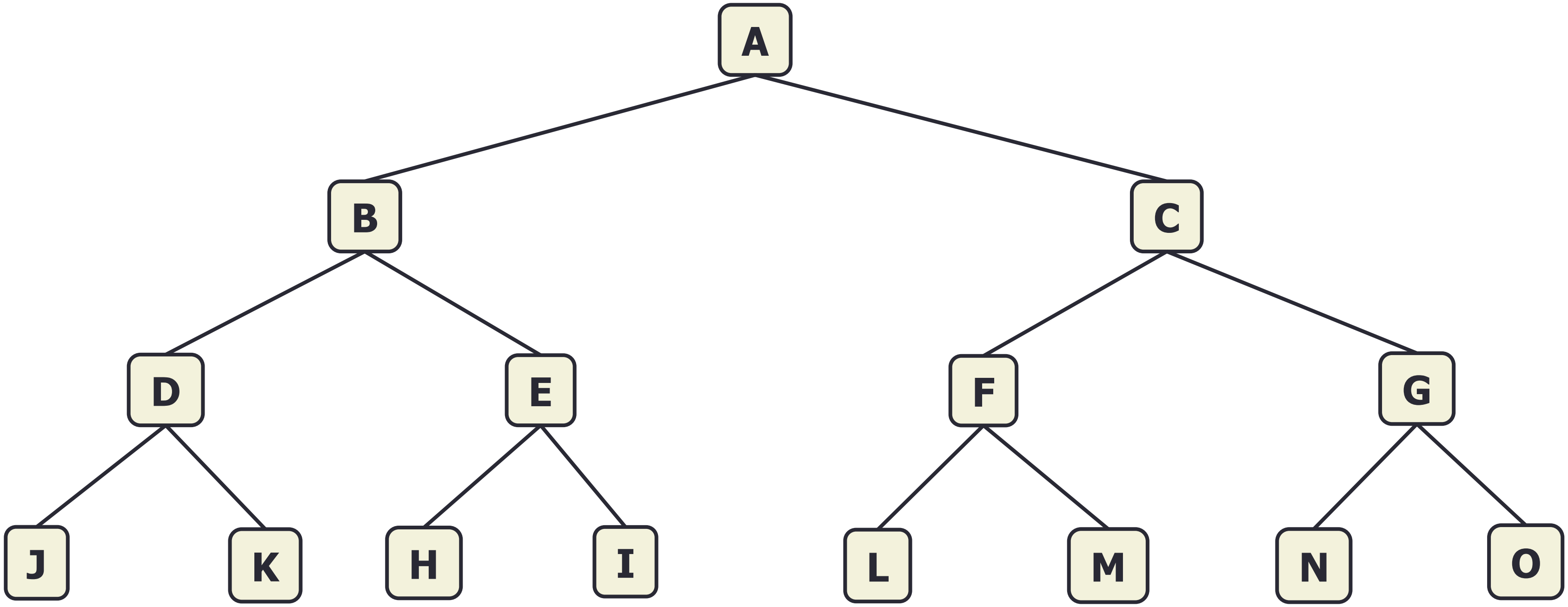


Complete Binary Tree

- Perfect tree is a special case of complete tree with last level completely filled.
- In some literature, Perfect binary tree is referred as Complete binary tree. In that case, Complete binary tree is referred Almost Complete binary tree.

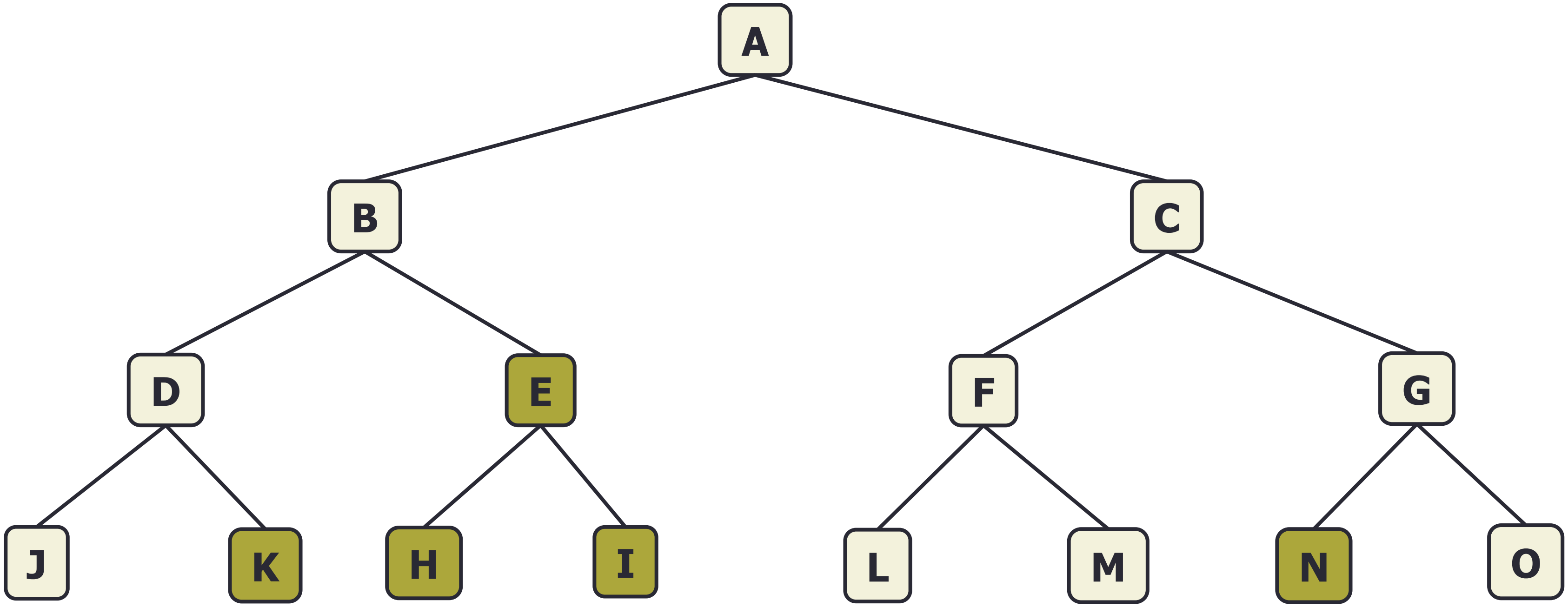
Binary Tree

- Any binary tree can be thought of as a tree obtained by pruning some nodes of a perfect binary tree



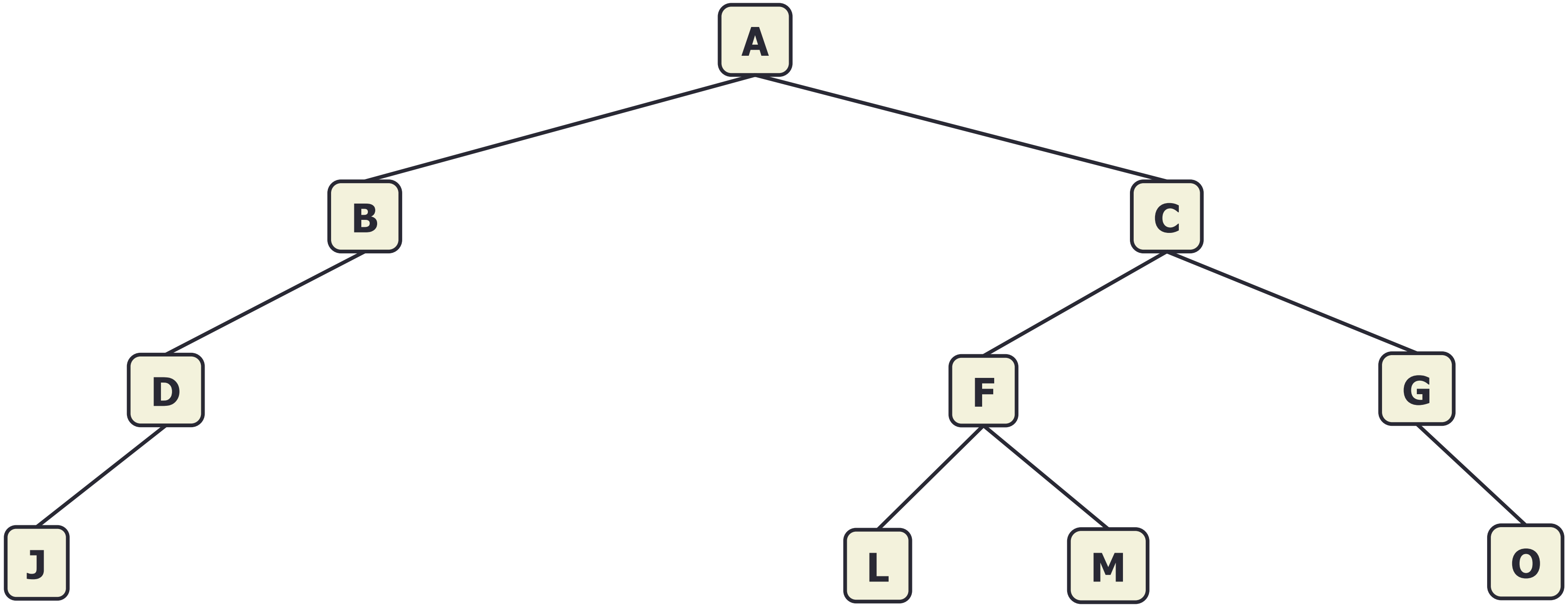
Binary Tree

- Any binary tree can be thought of as a tree obtained by pruning some nodes of a perfect binary tree



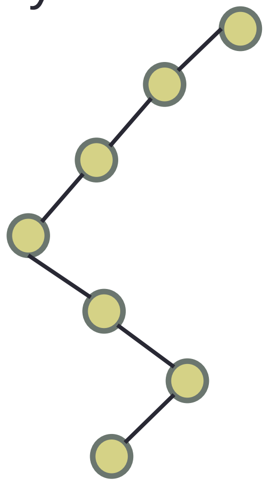
Binary Tree

- Any binary tree can be thought of as a tree obtained by pruning some nodes of a perfect binary tree



Height of a Binary Tree

- If a binary tree has n nodes and height h, then
  - Level i has at most 2^i nodes
  - $n <= 2^{h+1} – 1$
  - Hence, $h >= \log_2(n+1)/2$ i.e. minimum height of a tree with n nodes is $O(\log_2 n)$
  - Maximum height of a tree with n nodes is n-1 which is obtained when every non-leaf node has exact one child

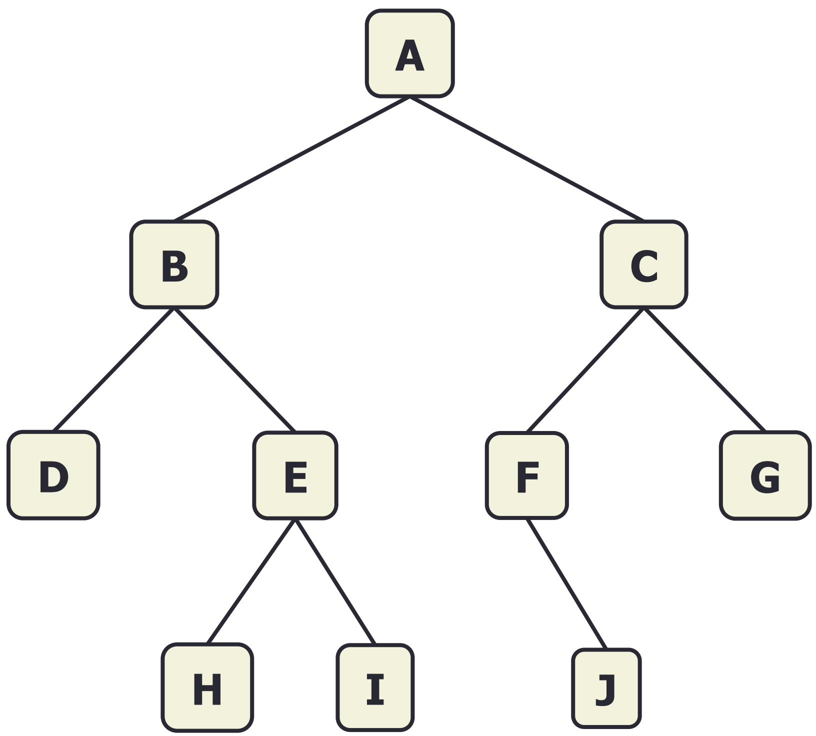


Linear Representation of Binary Tree

- Binary tree can also be represented using arrays
- Store root node at index ***0***
- Store left child of a parent node is at ***2*i+1*** where ***i*** is the index of parent node in array
- Store right child of a parent node is at ***2*i+2*** where ***i*** is the index of parent node in array
- Parent of a node at index ***i*** can be found at ***(i-1)/2*** except for root node

Example

A	B	C	D	E	F	G			H	I		J						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	…	26



Linear Representation of Binary Tree

- If a node doesn’t have a left or/and right child, indices for left or/and right child are empty
- If index of a child is greater than size of array, child of that node does not exist