Tree

CSC-114 Data Structure and Algorithms

Slides credit: Ms. Saba Anwar

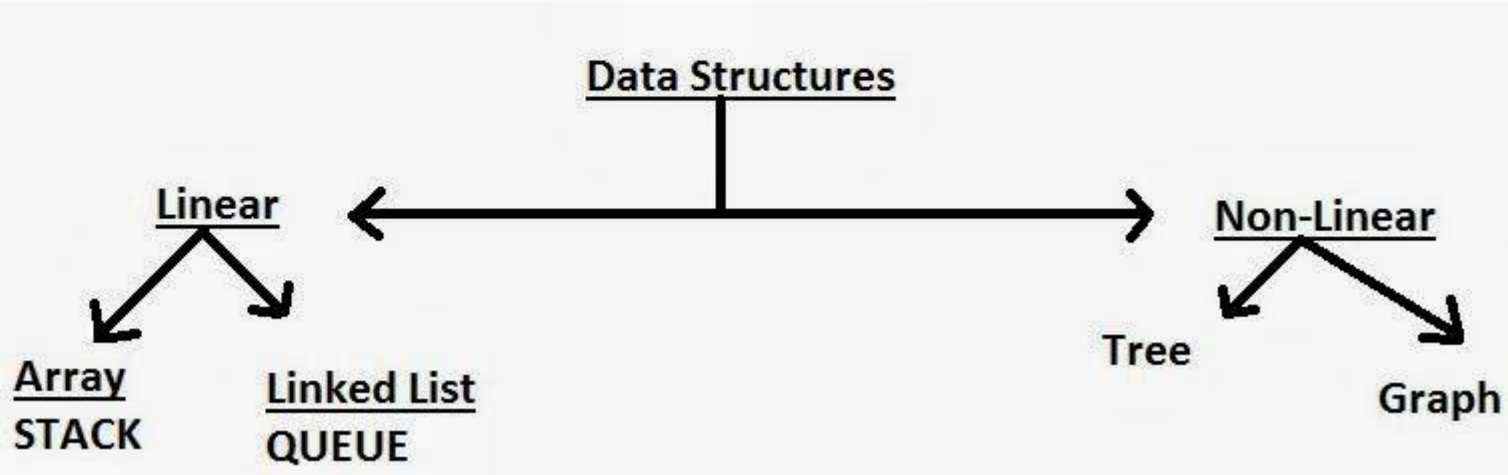
Outline

- Non-Linear Data Structures
  - Tree
    - Tree Terminologies
    - Memory Representation
    - Tree as ADT
    - Binary Tree
      - Traversal Strategies
        - BFS
        - DFS
          - Pre order
          - Post order
          - In order

Non-Linear Data Structure

Linear vs non-linear classification of data structures is dependent upon how individual elements are connected to each other.

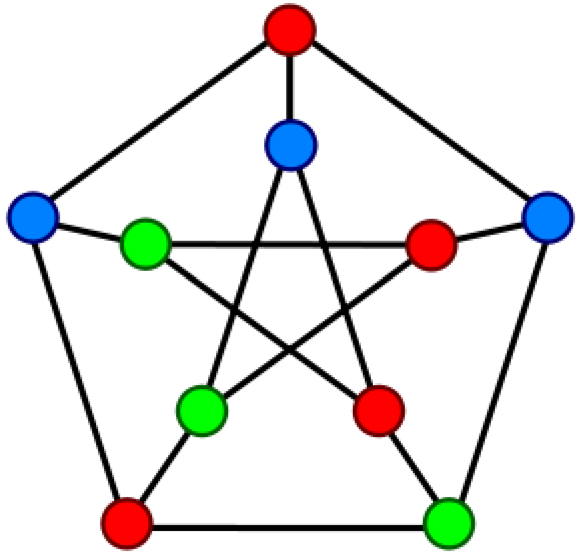
- All linear data structures have one thing in common that they are sequential
  - Lists, Stack, Queue
- In Non-Linear data structures, data elements are not sequential, an element can refer to more than one elements
  - Tree, Graphs, Tables



Graph

Graph is a non-linear mathematical structure that is defined as G= (v, e), where v is a set of vertices$\{v_1, v_2, \ldots v_n\}$ and e is a set of edges $\{e_1, e_2, e_3, \ldots e_m\}$

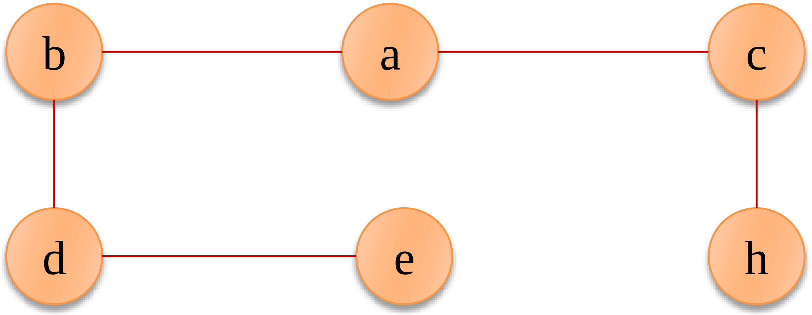
Where edge e is a pair of two vertices, means a connection between two vertices

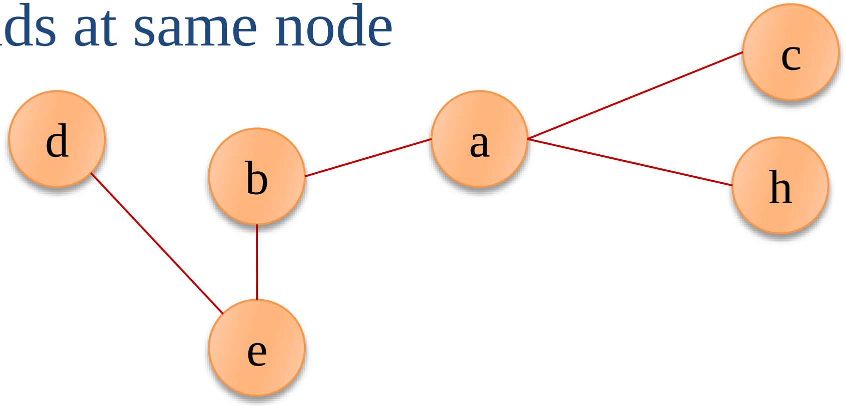


Tree

Tree is a connected graph which does not contain cycle

Cycle means a path which starts and ends at same node

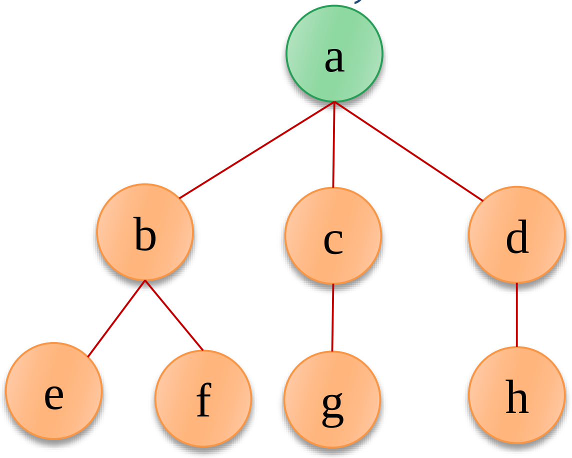




In field of computer science, a specific form of trees is more common, which is called **rooted** trees.

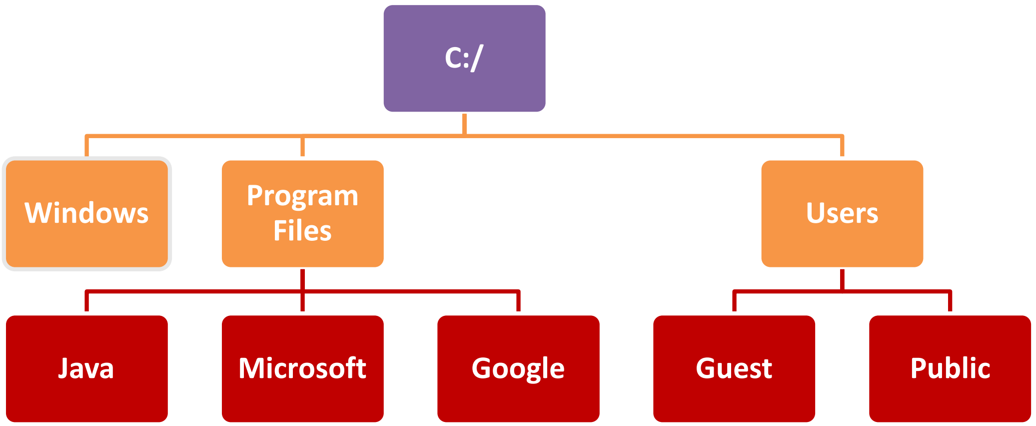
a is root vertex.

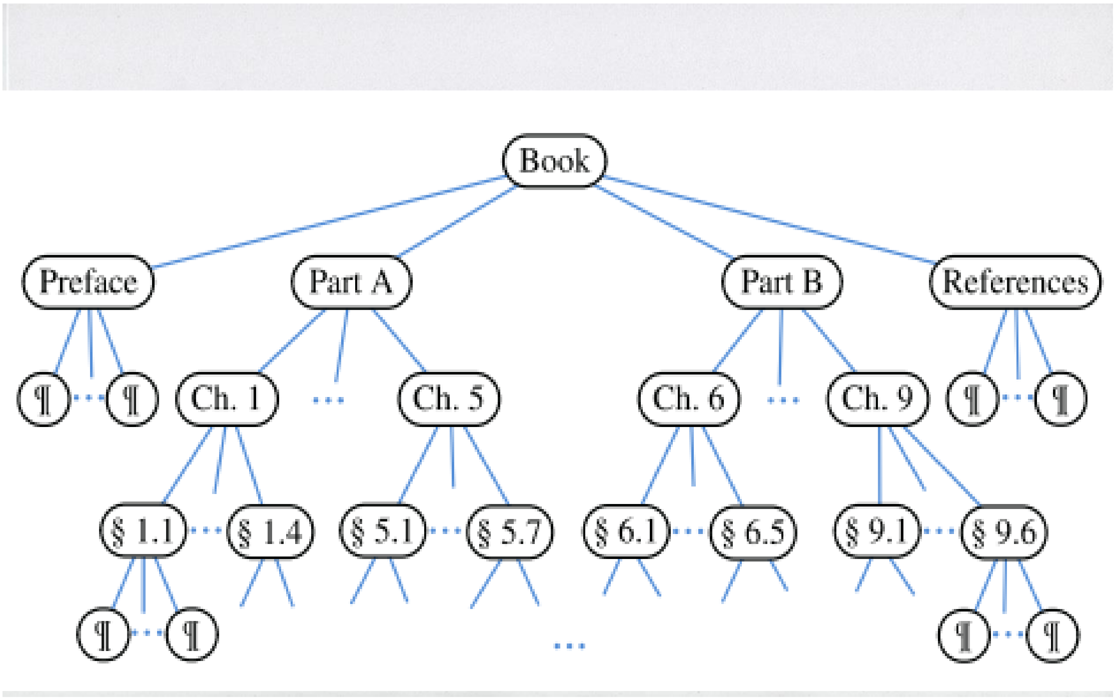
These rooted trees are directed graphs



Tree

So, Tree is defined as data structure which presents hierarchical relationship between data elements. Hierarchical means some elements are below and some are above from others. Like family tree, folder structure, table of contents





Tree: a Data Structure

Tree is a recursive data structure, it contains patterns that are themselves are trees.

A data structure is recursive if it is composed of smaller pieces of it's own data type. Such as list and trees.

a is root of all nodes like b, d etc.

b, c, d are also root of their sub trees and so on.

So, a tree **T** can be defined recursively as:

Tree T is a collection of nodes such that:

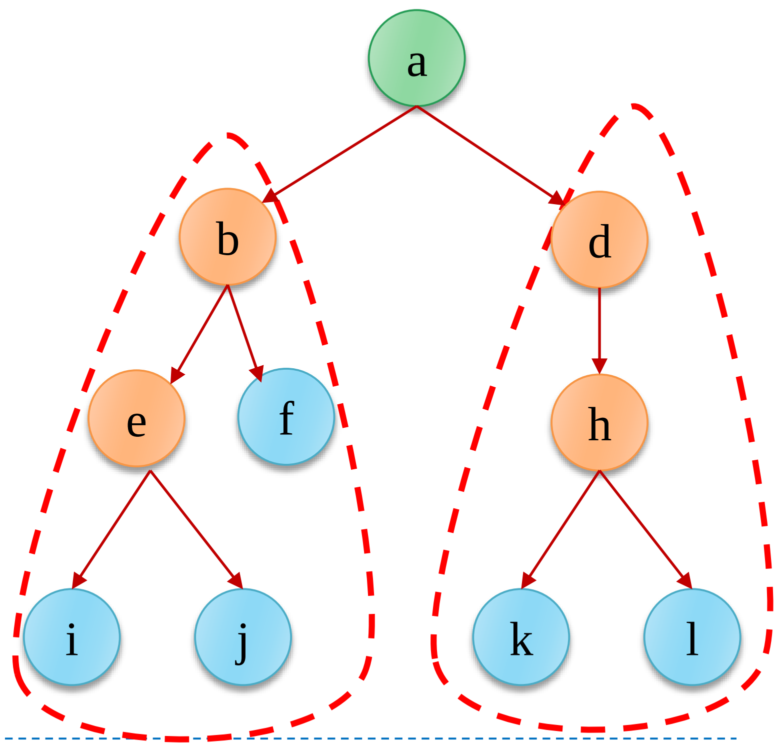
T is empty/NULL (No node) **OR**

There is a special node called **root**,

which can have 0 or more children (T_1, T_2, T_3 …T_n)

which are also sub-trees themselves.

T_1, T_2, T_3 …T_n are disjoint sub trees (no shared node)



Tree Applications

Tree is an extremely useful data structure, it provides natural organization of data which exhibits hierarchy, due to their non-linear structure they provide efficient operations with compare to linear data structures. Few uses are as follows:

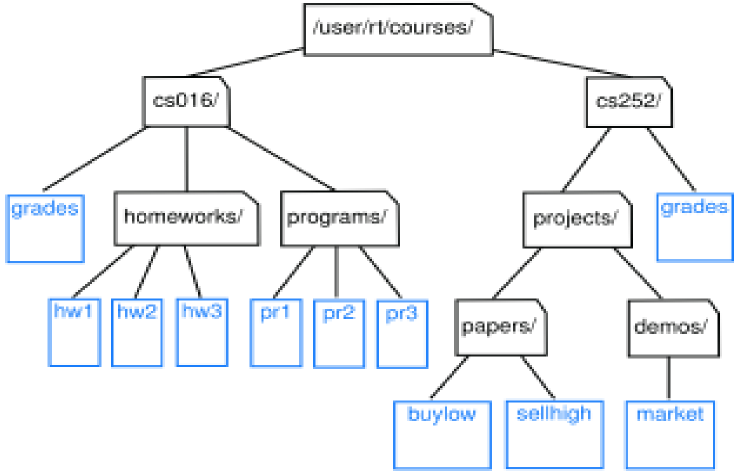


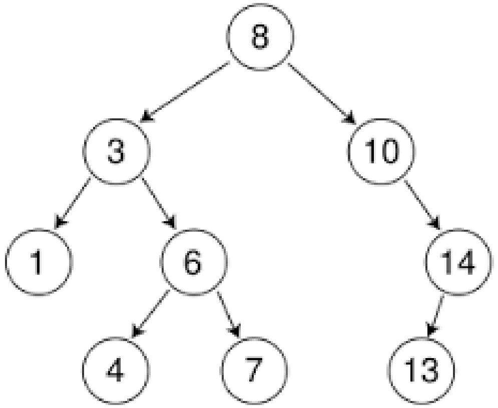
Figure 8.3: Tree representing a portion of a file system.

Disk File System

Used by operating system to stored folder hierarchy

Search trees

More efficient than sorted list



Tree Applications

Parse Trees

Used by compilers to produce machine code

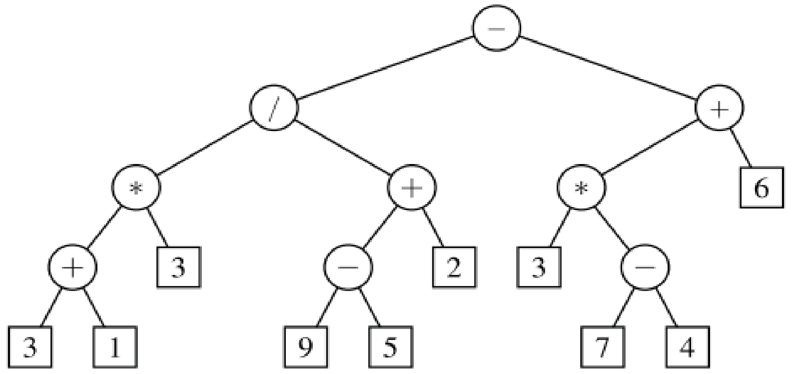


Figure 8.6: A binary tree representing an arithmetic expression. This tree repre-

Decision Trees

Used in artificial intelligence to build knowledge base

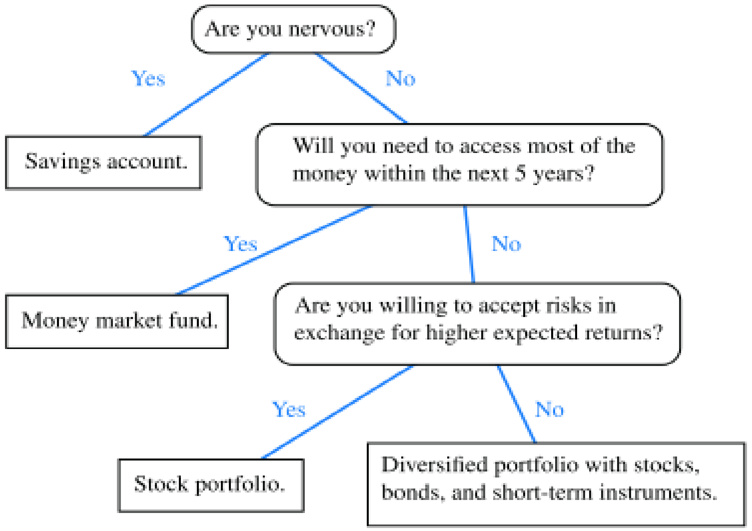
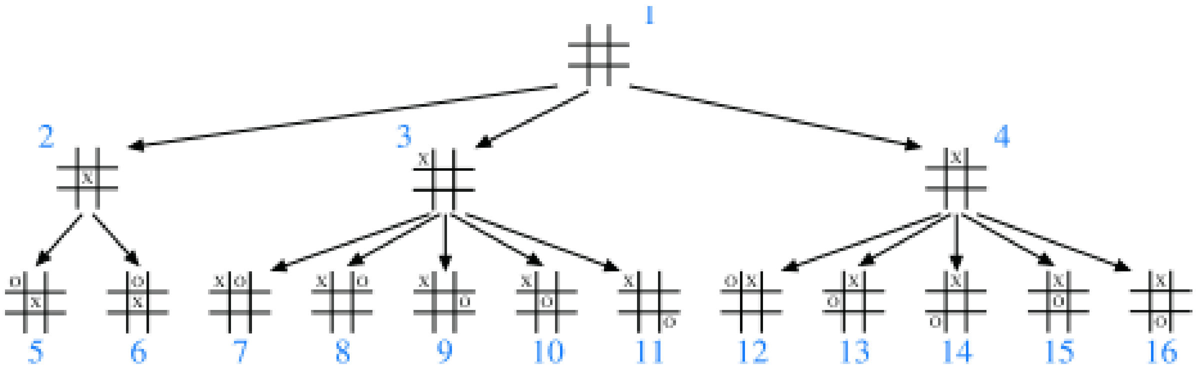


Figure 8.5: A decision tree providing investment advice.

Tree Applications

Games

- are used in logic games

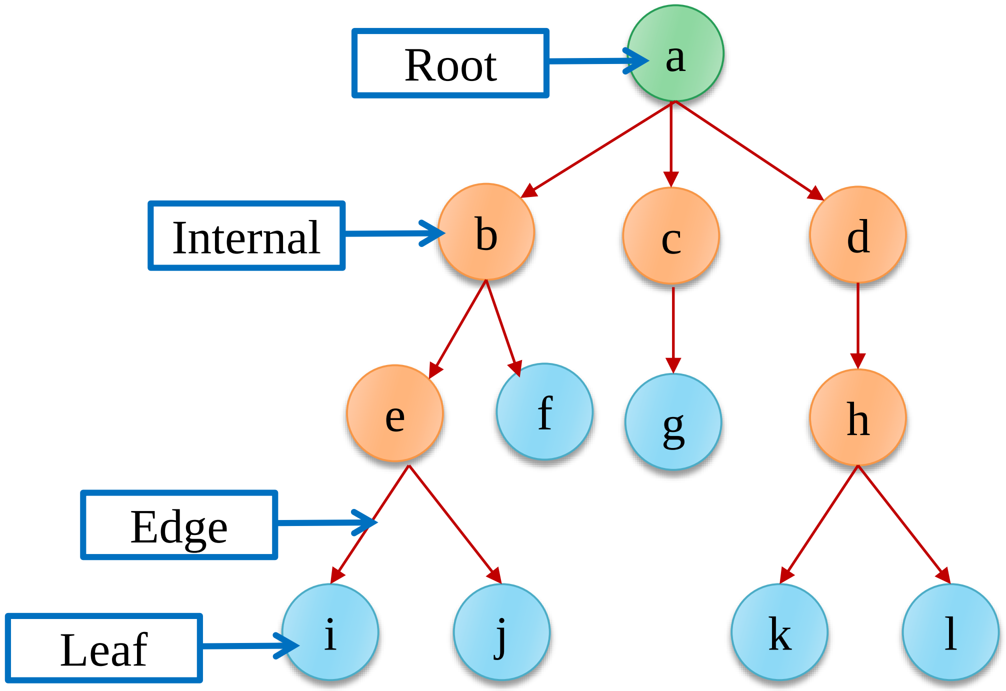


- Data Compression
  - Huffman coding trees
- Priority Queue
  - Heap Tree

And many more other applications.

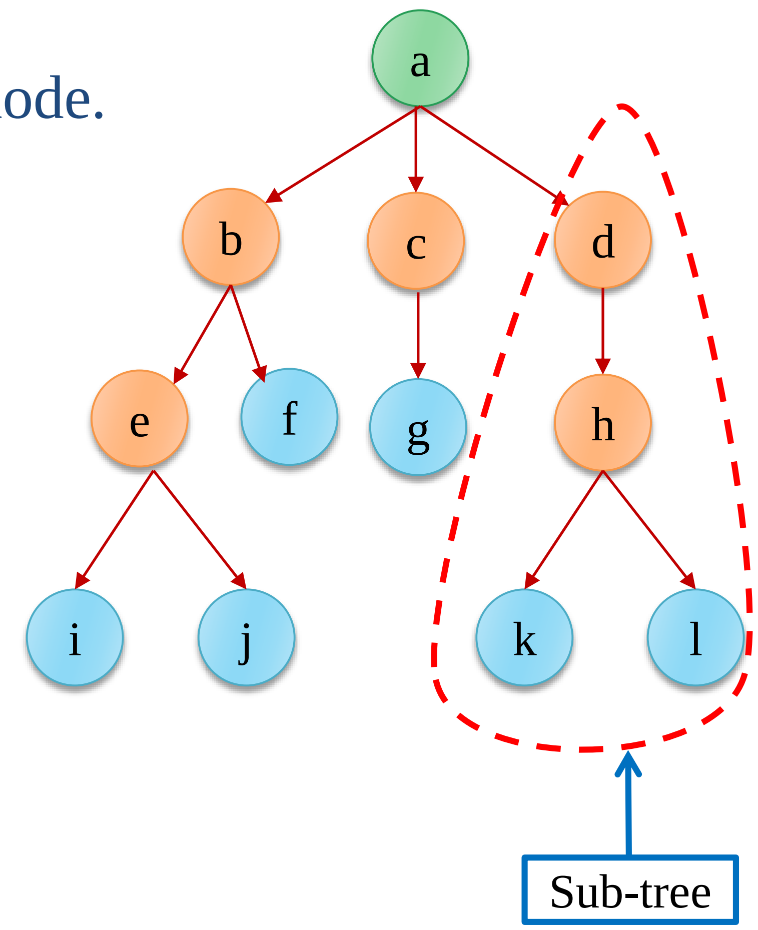
Tree Terminologies

- Node/Vertex
  - One data unit of tree
- Edge
  - Arc/link from one node to other
- Root node
  - The top node of tree. A node with no parent
- Leaf/External node
  - Node with no child
- Internal Node:
  - Node with child
- Ancestors of Node
  - Parent, all grand parents and all great grand parents of node.
    - a, b and e are ancestors of i.



Tree Terminologies

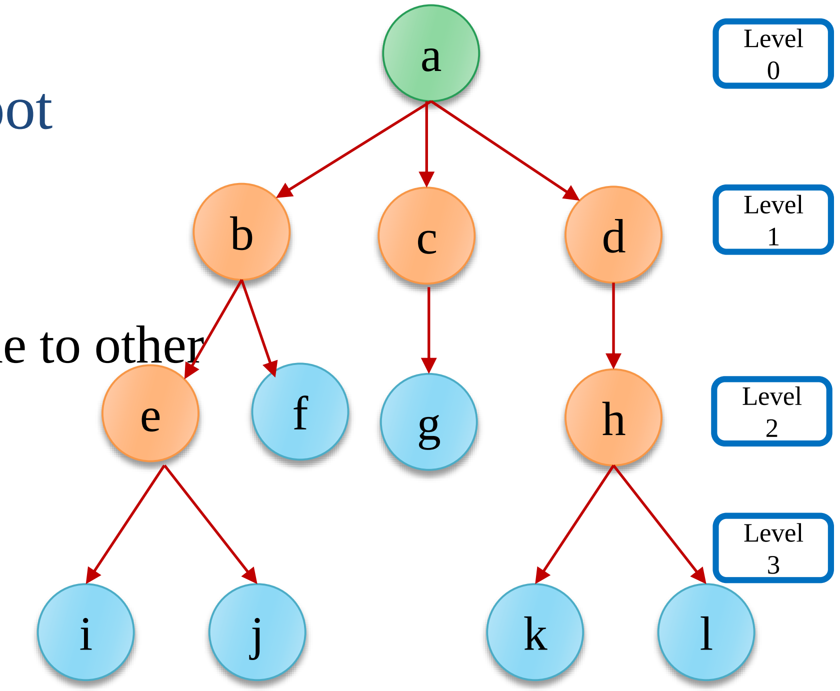
- Descendants of Node
  - Child , all grand children and great grand children of node.
    - i, j, e and f are descendants of b.
- Sub Tree
  - A node within tree with descendants
- Degree of Node:
  - Number of its children
    - a's degree is 3
    - b, h and e's degree is 2
    - c's degree is 1
- Degree of Tree
  - Maximum degree of any node
    - Since a has degree 3 that is maximum so degree of tree is 3



Tree Terminologies

Depth/Level of Node

- Number of ancestors or length of path from node to root
  - j has depth 3
  - c has depth 1
  - **Length of Path** means # of edges on the path from one node to other



Siblings

- Nodes with same parent and at same level
  - i and j
  - b and c and d

Height of Tree

1. Maximum depth of any node→3
2. Longest path from root to any leaf node → 3

Tree as ADT

A tree T provides following basic operations:

Tree Methods:

- size(root): returns total number of nodes
- isEmpty(root): if tree is empty or not
- root(): returns root node of tree

Node Methods:

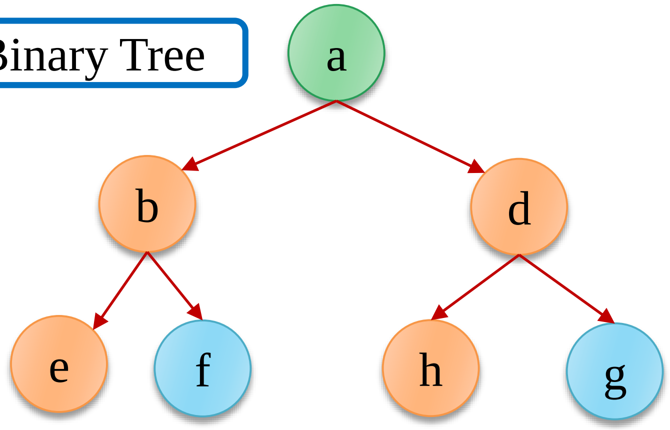
- parent(node): returns parent of node
- children(node): returns list of all child’s of node
- isInternal(node): if node is non-leaf
- isExternal(node):if node is leaf
- isRoot(node): if node is root

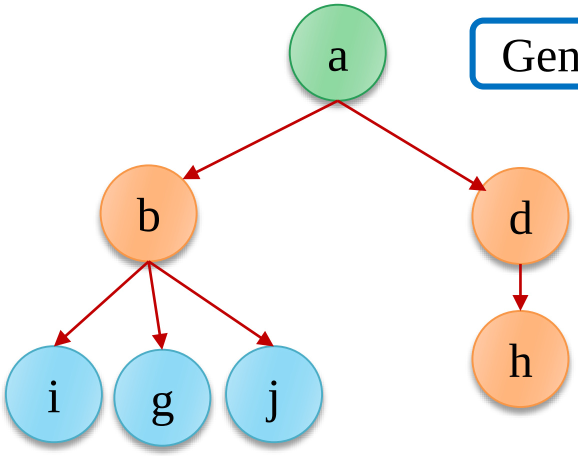
Binary Tree

Binary Tree is a special tree where each node can have maximum two children. In other words maximum degree of any node is 2.

Each node has a left child and a right child. Even if a node has only one child, other child is still mentioned with NULL.

Binary Tree





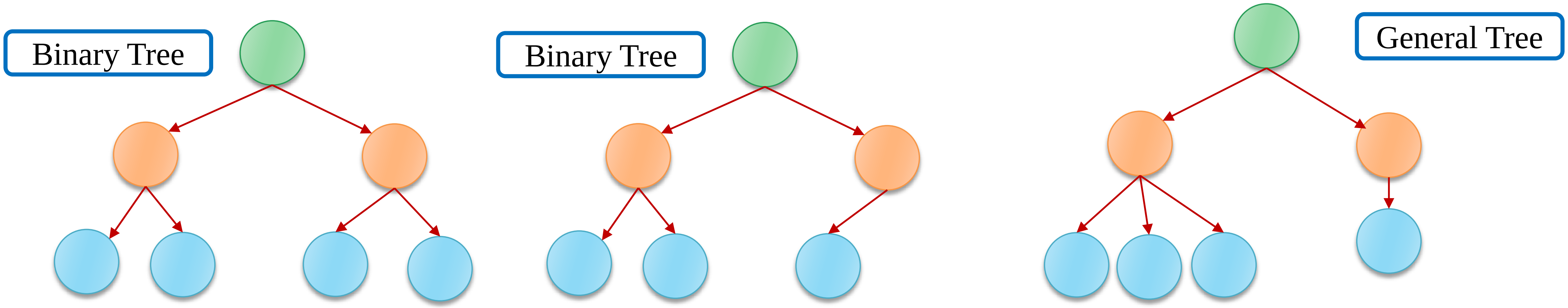
General Tree

Binary Tree

Recursive Definition:

T is a binary tree if

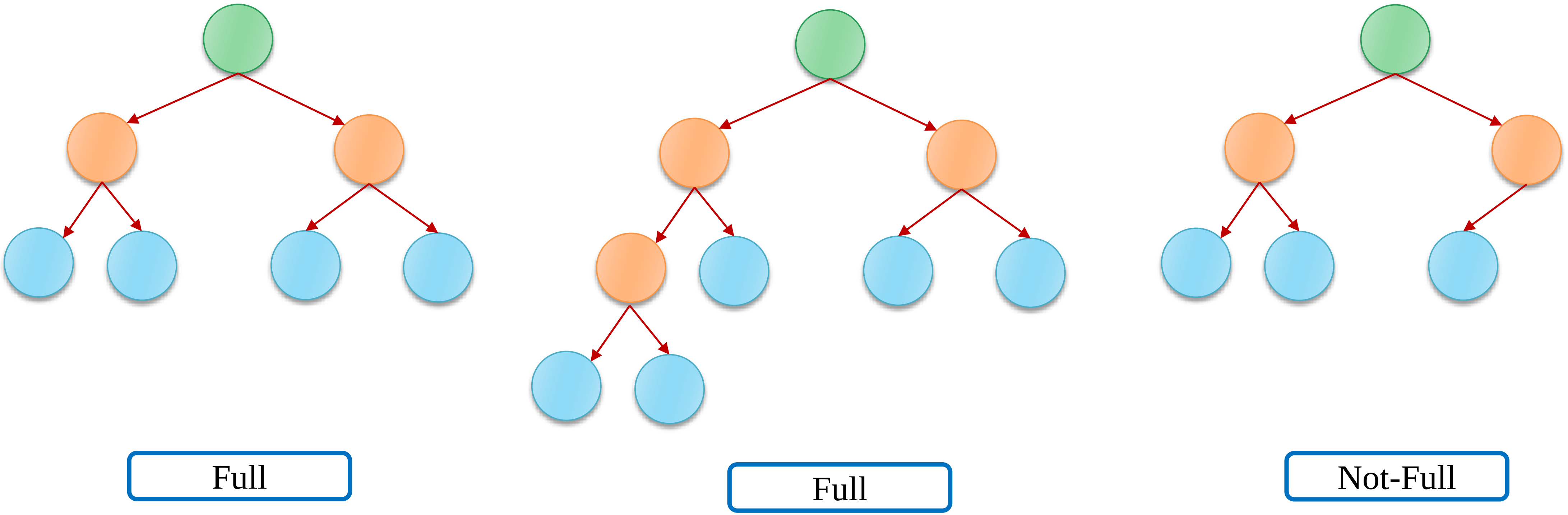
- T is empty (NULL) **OR**
- T’s root node has maximum two children's, where each child is itself a binary tree.
  - Left child is called left subtree and right child is called right subtree



Full Binary Tree

Degree of each node is either 0 or 2.

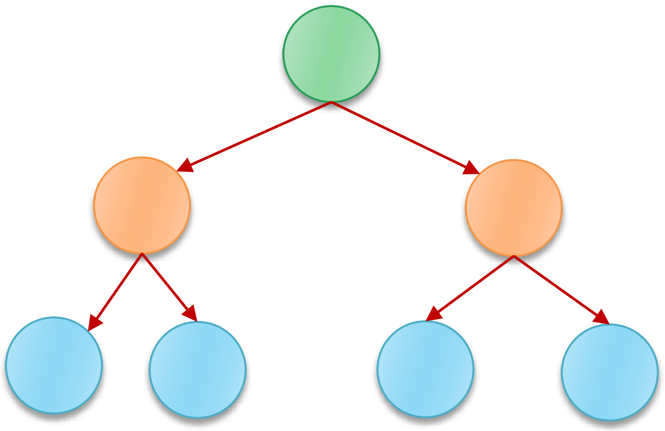
Full Tree is also referred as Proper Tree



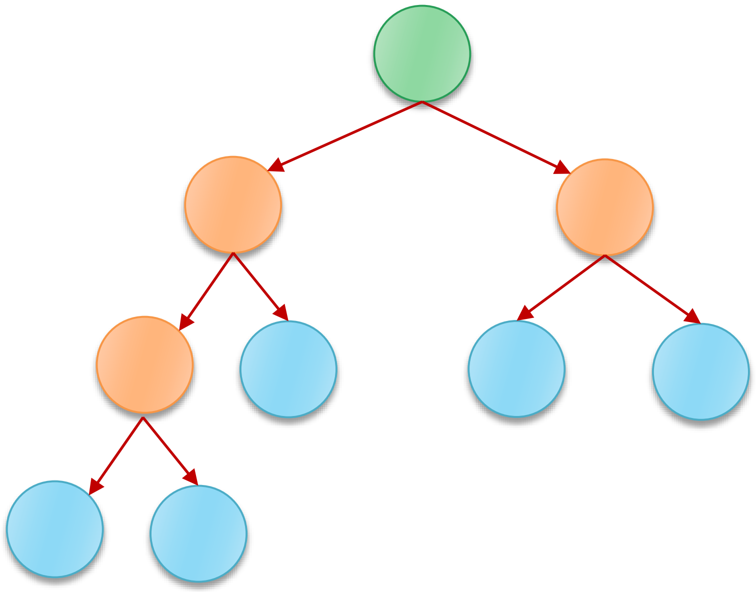
A tree that is not Full/Proper , is called improper or not-full

Perfect

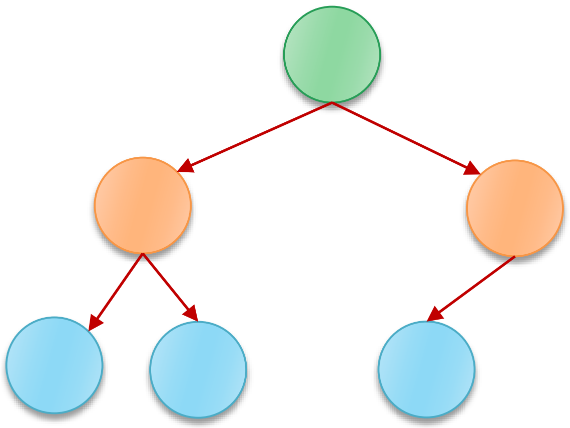
A Full/Proper binary tree in which each leaf node has same depth/level.



Perfect & Full



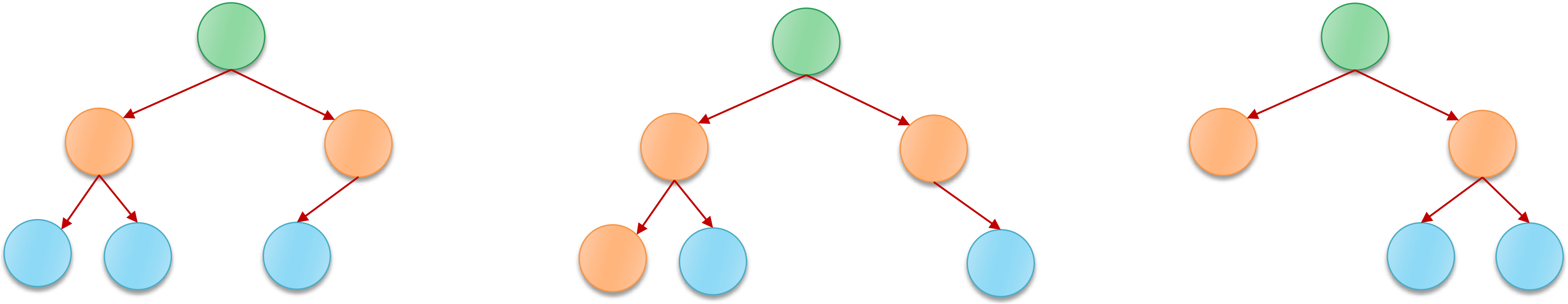
Not-Perfect But Full



Not Perfect Nor Full

Complete Binary Tree

A tree that is completely filled at all levels, except the last level which is filled from left to right



Complete But Not Full

Not-Complete

Not-Complete But Full

Binary Tree

Maximum nodes at level i of binary tree?

2^i

Maximum nodes in a binary tree?

$2^{h+1}-1$

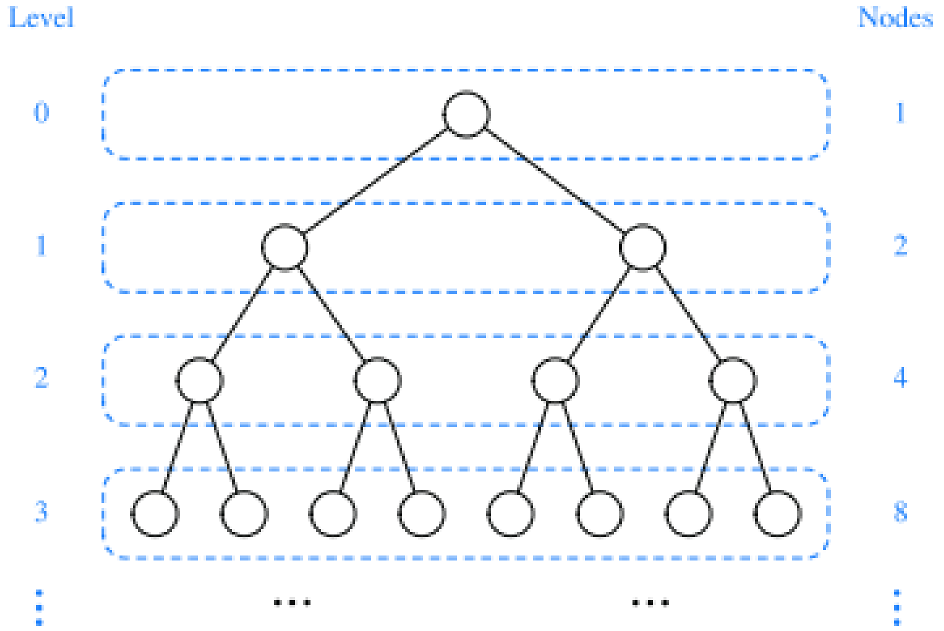


Figure 8.7: Maximum number of nodes in the levels of a binary tree.

Binary Tree ADT

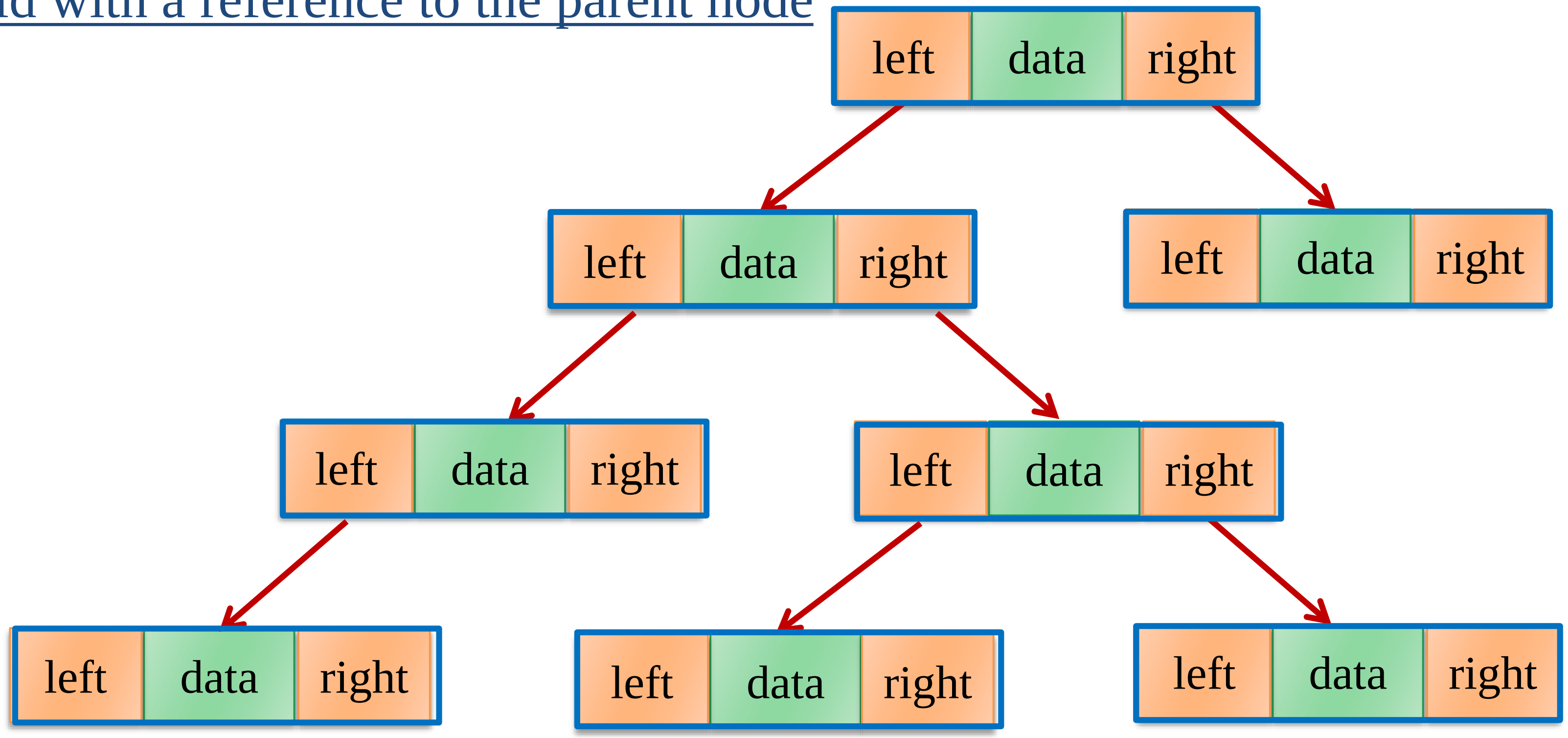
In addition to previous function Binary Tree provides additional functions:

- left(node): returns left child of node
- right(node): returns right child of node
- hasLeft(node): tells if a node has left child or not
- hasRight(node): tells if a node has right child or not
- sibling(node): returns sibling of given node
  - First find parent, then see it node itself is left or right child

Binary Tree Implementation

Linked representation

- Each node has two links left and right
- If root node is null, means tree is empty
- If node's left, right links are NULL, it means its leaf node
- **Optionally**, a parent field with a reference to the parent node



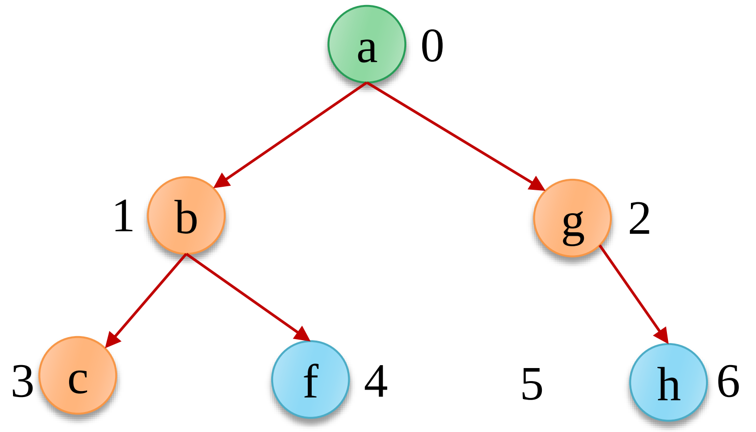

```
class Node{
data;
Node left;
Node right;
}
```

Binary Tree Implementation

Array representation

A fixed size tree can be represented using 1-D array.

If we know the height of tree, we can define size of array to hold maximum possible number of nodes → $2^{h+1}-1$



Root of tree → array[0]

Left child of root →array[1]

Right child of root →array[2]

Left child of node at index k →array[2k+1]

Right child of node at index k→ array[2k+2]

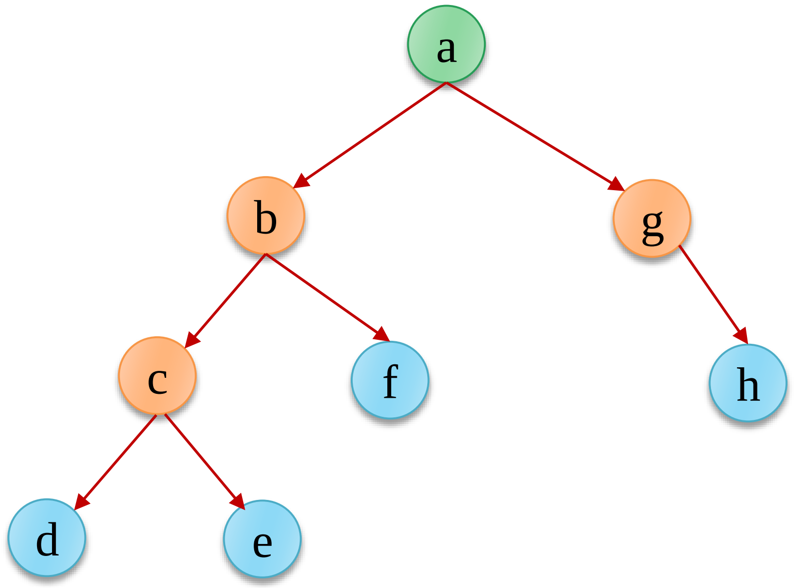
0	**1**	**2**	**3**	**4**	**5**	**6**
a	b	g	c	f	NULL	h

Tree Traversal

A tree traversal means visiting each node of tree once.

Due to non-linear structure of tree there is not a single way to traverse node:

1. Breadth First Search
2. Depth First Search
   - Pre-Order
   - In-Order
   - Post-Order



Breadth First Search (BFS)

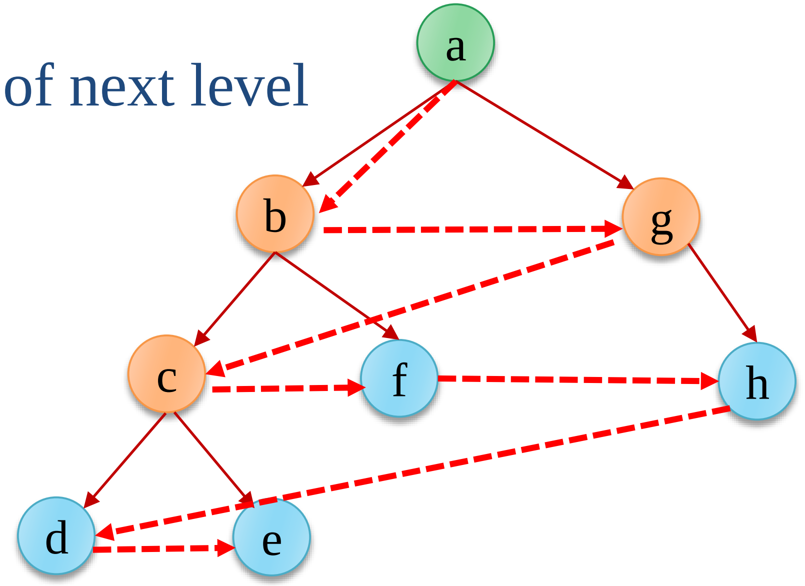
Starting from root node, visit all of its children, all of its grand children and all of its great grand children

- Order of nodes: a b g c f h d e
- Nodes at same level must be visited first before nodes of next level

Also known as level order traversal

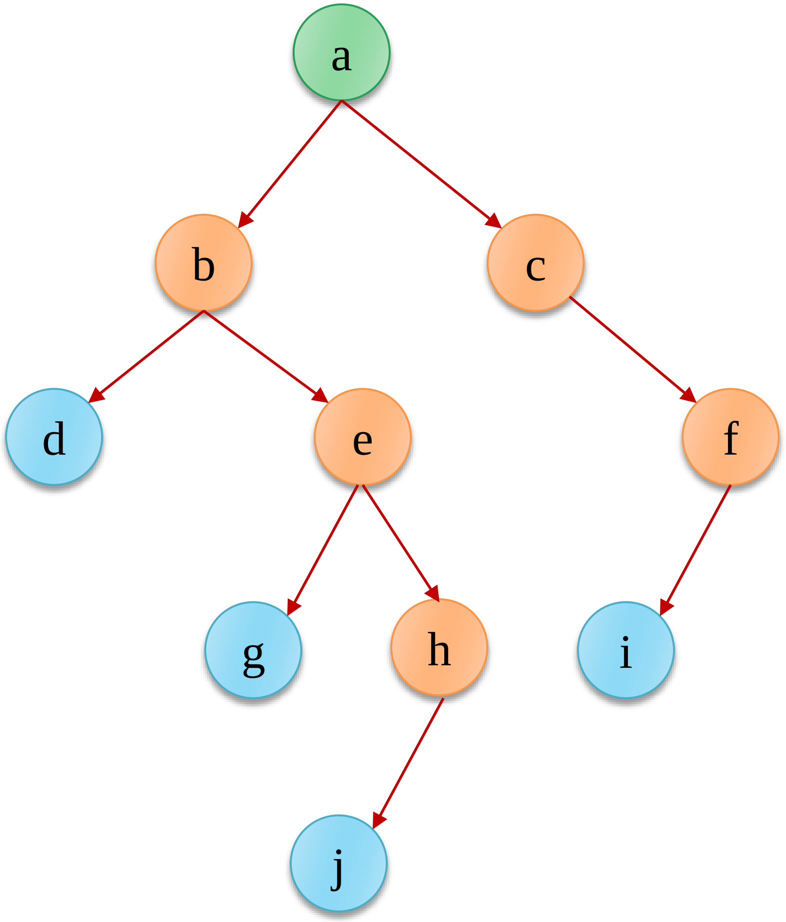
Implementation?

- We should store nodes to keep track of them.
- The sequence in which we store them effects the the sequence in which we retrieve them back

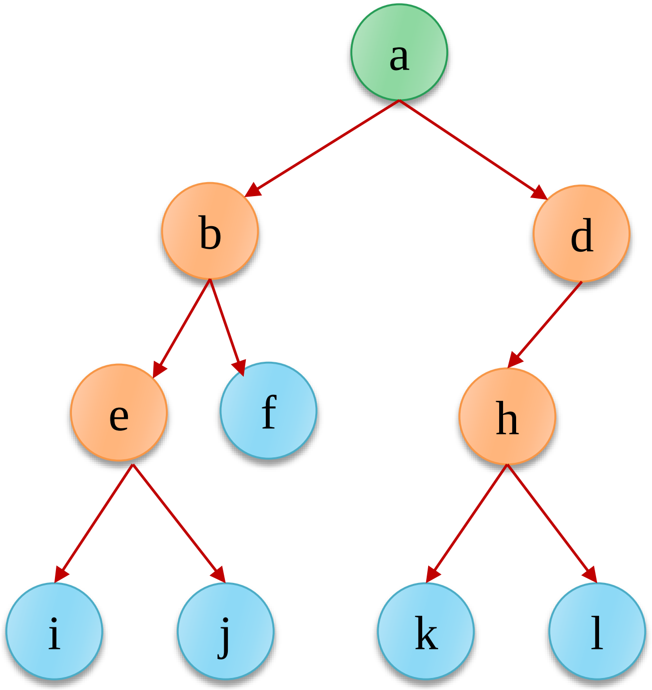


- Which data structure can be used to store nodes?
  - array, stack or queue

Breadth First Search (BFS)



a b c d e f g h i j



a b d e f h i j k l

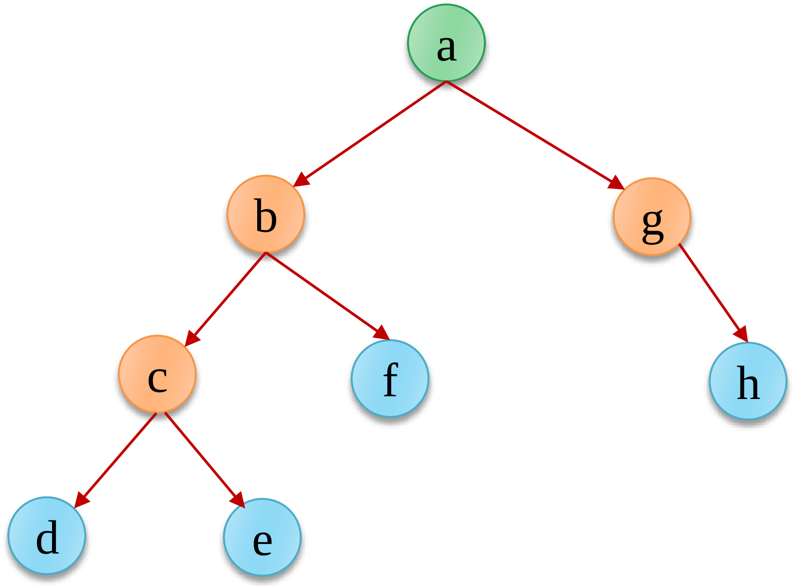
Breadth First Search (BFS)

Algorithm: Iterative_BFS(Tree root)

Input: root node of Tree.

Steps:

1. If root is not NULL
2. Let Q =new Queue ()
3. Set node = root
4. Q.enqueue(node)
5. While(Q is not empty)
6. node=Q.dequeue()
7. print(node)//print node's data
8. If hasLeft(node)
9. Q.enqueue(node.left)
10. If hasRight(node)
11. Q.enqueue(node.right)
12. End While
13. End If



a						
b	**g**					
	g	**c**	**f**			
		c	**f**	**h**		
			f	**h**	**d**	**e**

				h	**d**	**e**
					d	**e**
						e

Recursive BFS

```
Recursive_BFS(Tree node, Queue Q)
  If(node is not NULL)
    print(node)
    If hasLeft(node)
     Q.enqueue(node.left)
    If hasRight(node)
     Q.enqueue(node.right)
    If Q is not Empty
     Recursive_BFS(Q.dequeue(),Q)
  End if
```

Depth First Search (DFS)

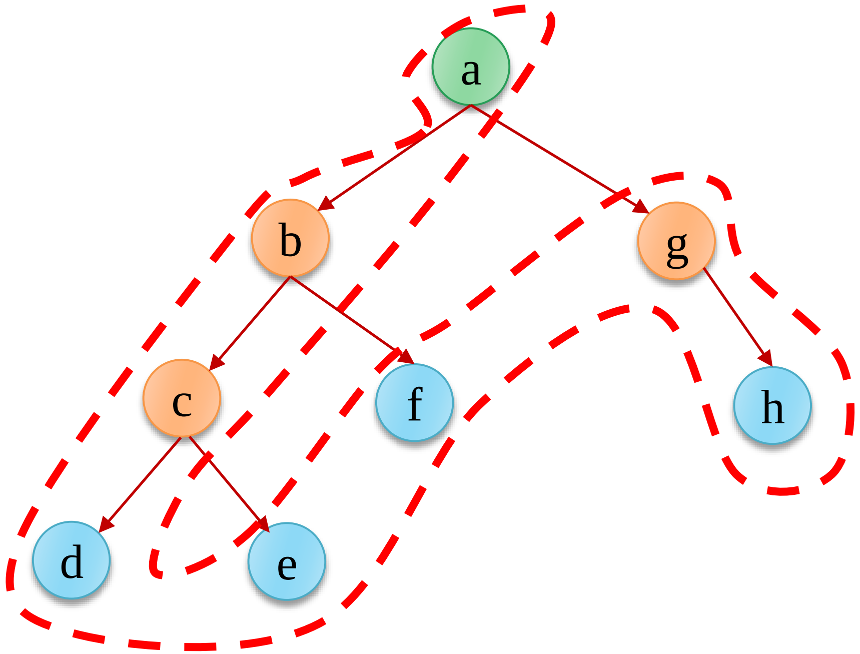
Using the top-down view of the tree, starting from root, go to each sub tree as far as possible, then back track

Possible Orders:

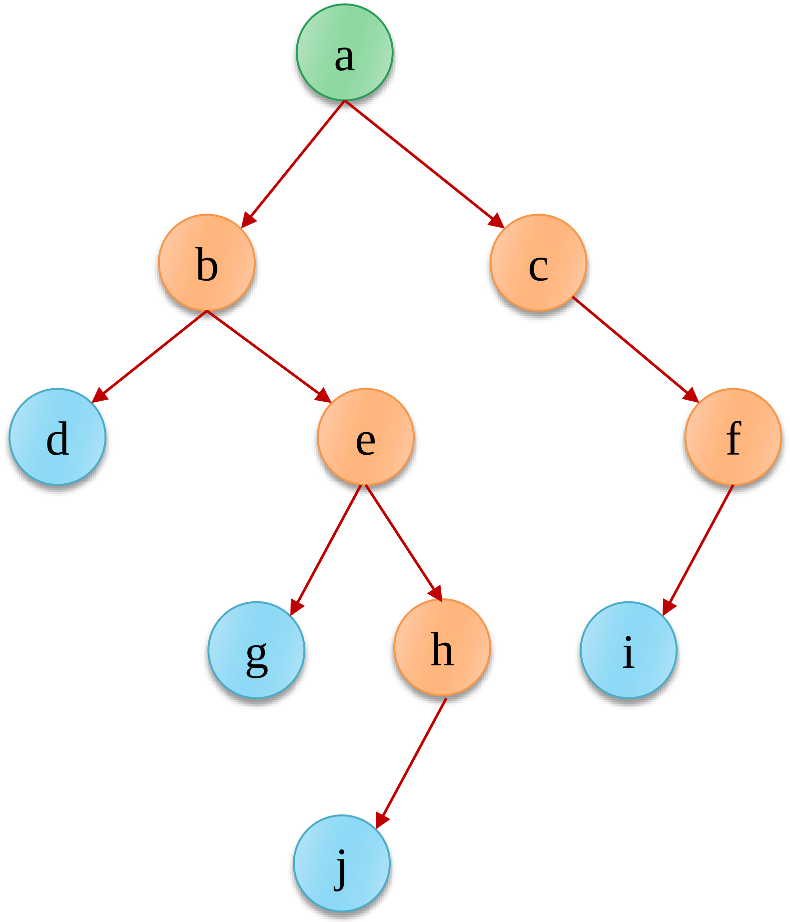
- Left sub tree and then right sub tree
  - a b c d e f g h
- right sub tree and then left sub tree
  - a g h b f c e d

Implementation:

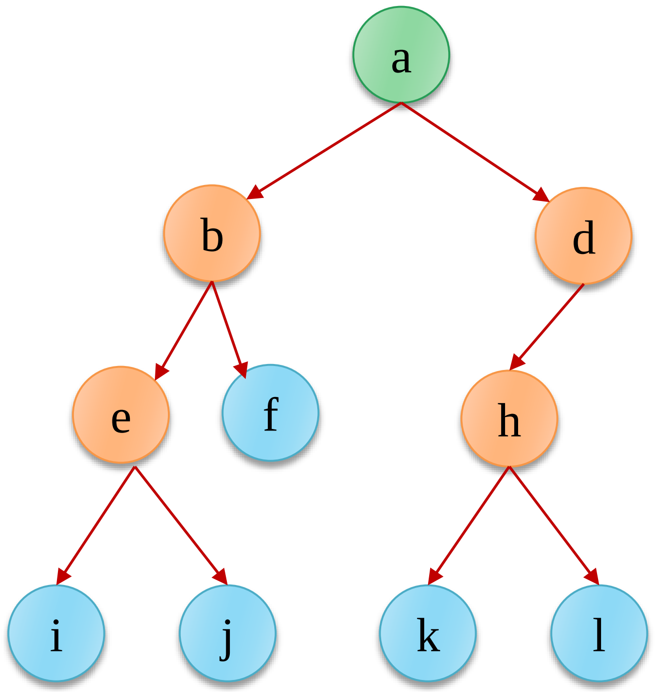
- Can we use a stack instead of queue



Depth First Search (DFS)



a b d e g h j c f i



a b e i j f d h k l

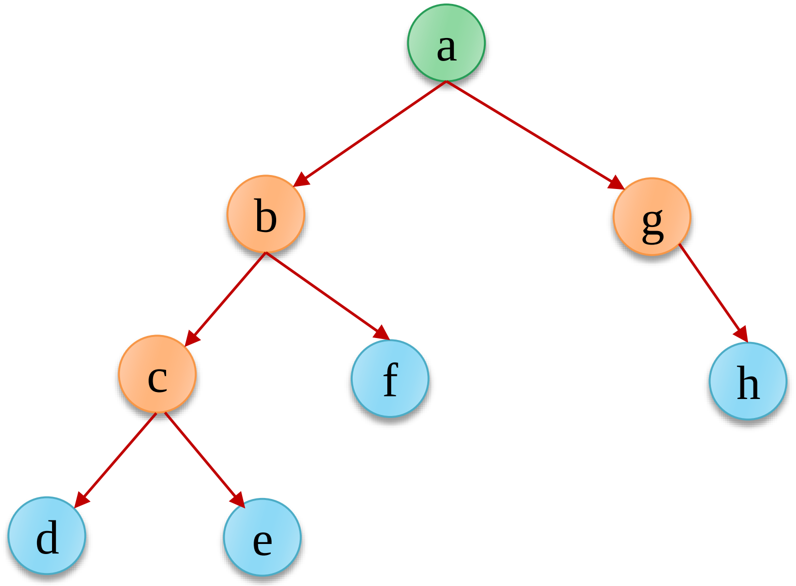
Depth First Search (DFS)

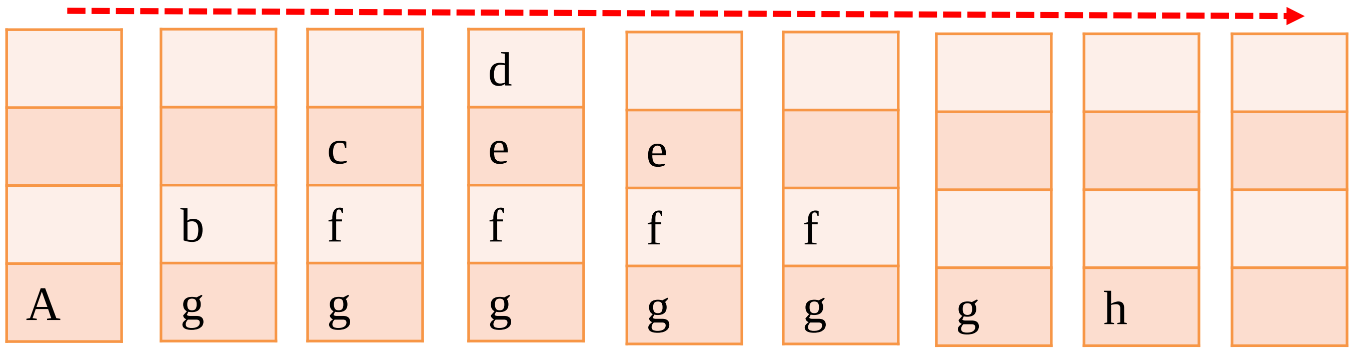
Algorithm: Iterative_DFS(Tree root)

Input: root node of Tree.

Steps:

```
If root is not NULL
  S=new Stack()
   set node = root
  S.push(node)
  While( S is not empty)
     node=S.pop()
     print(node)
     If hasRight(node)
        S.push(node.right)
     If hasLeft(node)
        S.push(node.left)
  End While
End If
```



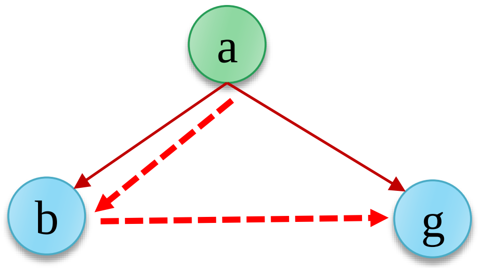


Depth First Variations

Depth First Search can also be implemented with recursive approach. And depending upon the order in which we go in depth can bring different variations in order of node traversal. Which are:

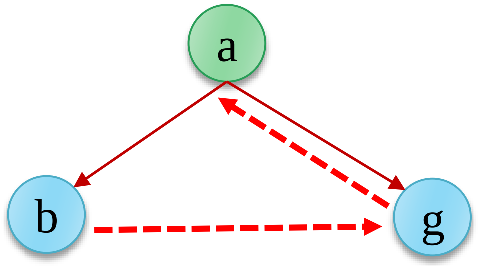
Pre-Order (simple DFS)

1. Visit node
2. Visit left child of node
3. Visit right child of node



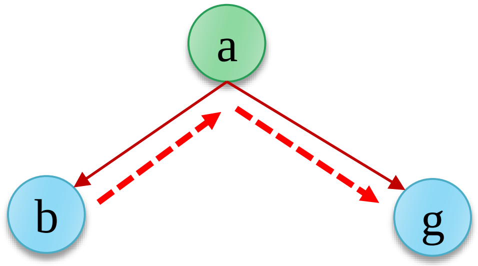
Post-Order

1. Visit left child of node
2. Visit right child of node
3. Visit node



In-Order

1. Visit left child of node
2. Visit node
3. Visit right child of node



Pre-Order vs. Post-Order vs. In-Order

Pre-order (node-left-right)

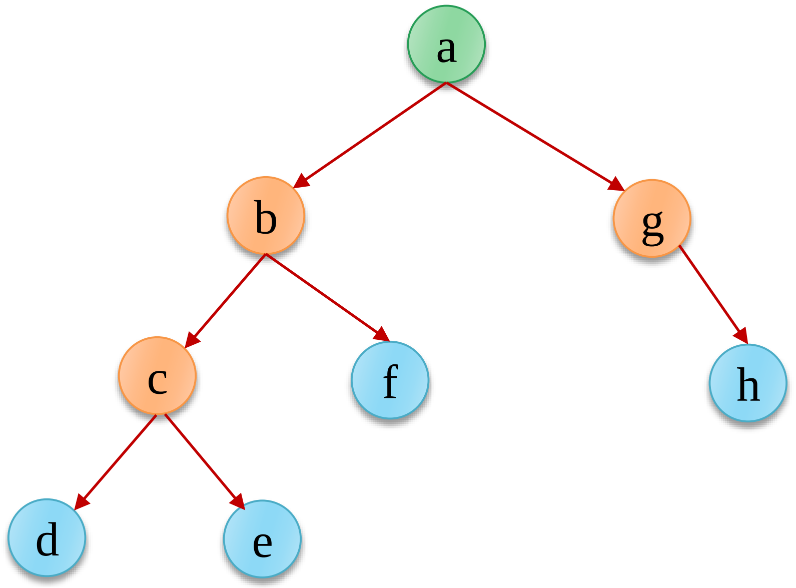
a b c d e f g h

Post-order (left-right-node)

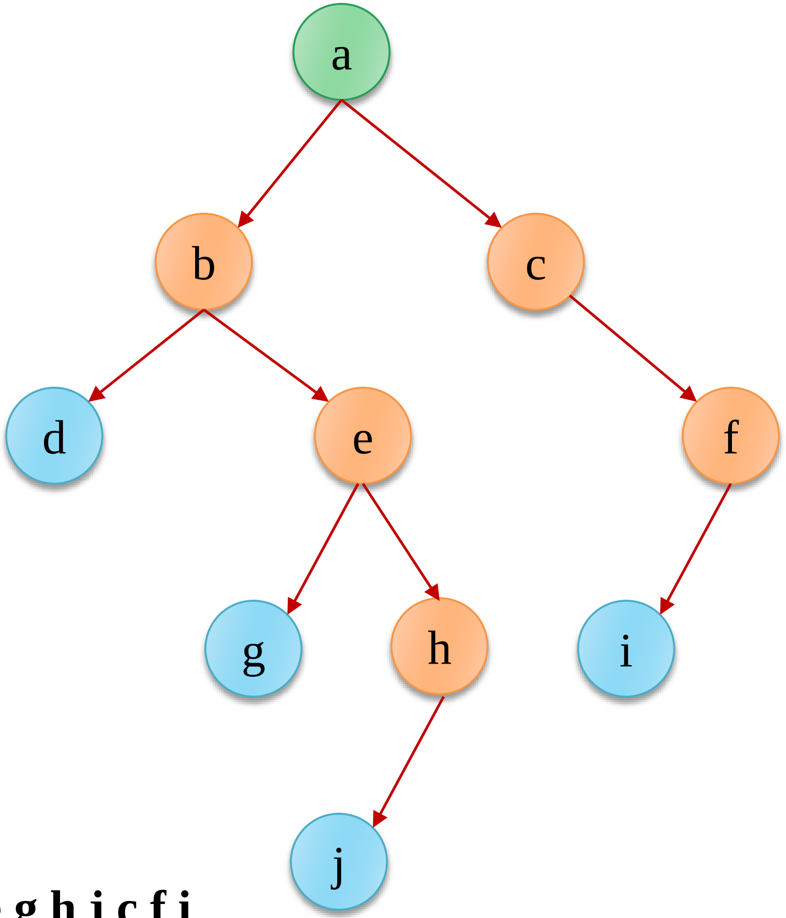
d e c f b h g a

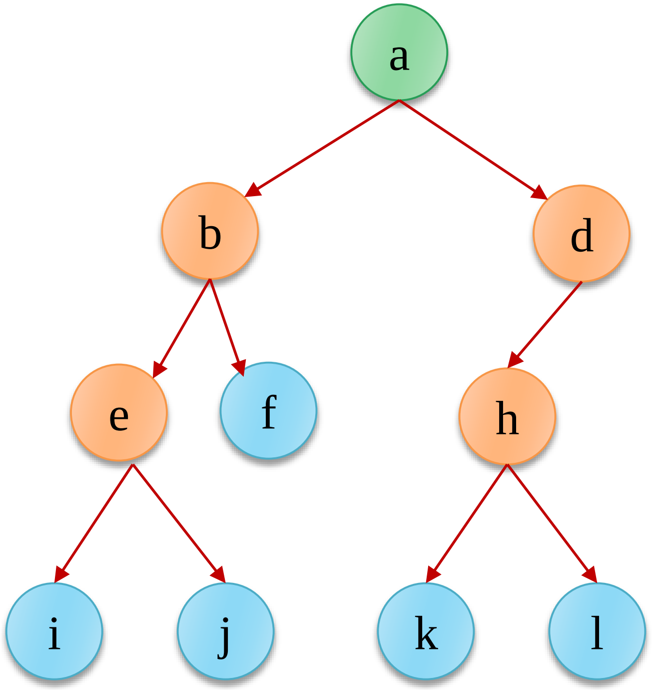
In-order (left-node-right)

d c e b f a g h



Pre-Order vs. Post-Order vs. In-Order





Pre-Order: **a b d e g h j c f i**
Post-Order: **d g j h e b i f c a**
In-Order: **d b g e j h a c i f**

Pre-Order**: a b e i j f d h k l**
Post-Order: **i j e f b k l h d a**
In-Order: **i e j b f a k h l d**

Tree Traversal-Recursive Algorithms

Recursive_PreOrder(Tree node)

```
If node is not NULL
    print(node)
    Recursive_PreOrder(node.left)
    Recursive_PreOrder(node.right)
End If
```

This is traditional DFS

Recursive_PostOrder(Tree node)

```
If node is not NULL
    Recursive_PostOrder(node.left)
    Recursive_PostOrder(node.right)
    print(node)
End If
```

Recursive_InOrder(Tree node)

```
If node is not NULL
    Recursive_InOrder(node.left)
    print(node)
    Recursive_InOrder(node.right)
End If
```