Binary Tree & Search

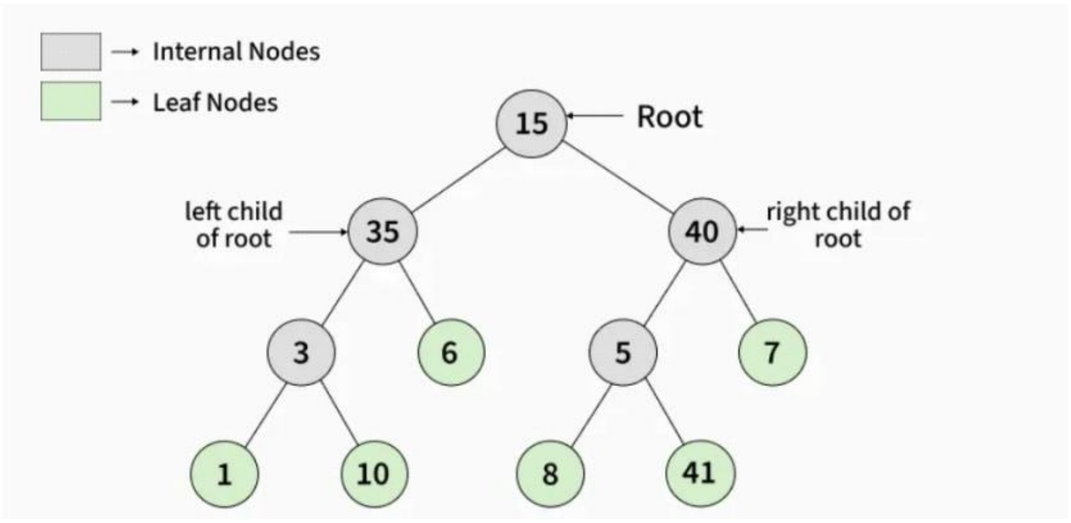
Saturday, 27 December 2025 12:10 pm

https://www.geeksforgeeks.org/dsa/introduction-to-binary-tree/

Introduction to Binary Tree

Last Updated : 09 Oct, 2025

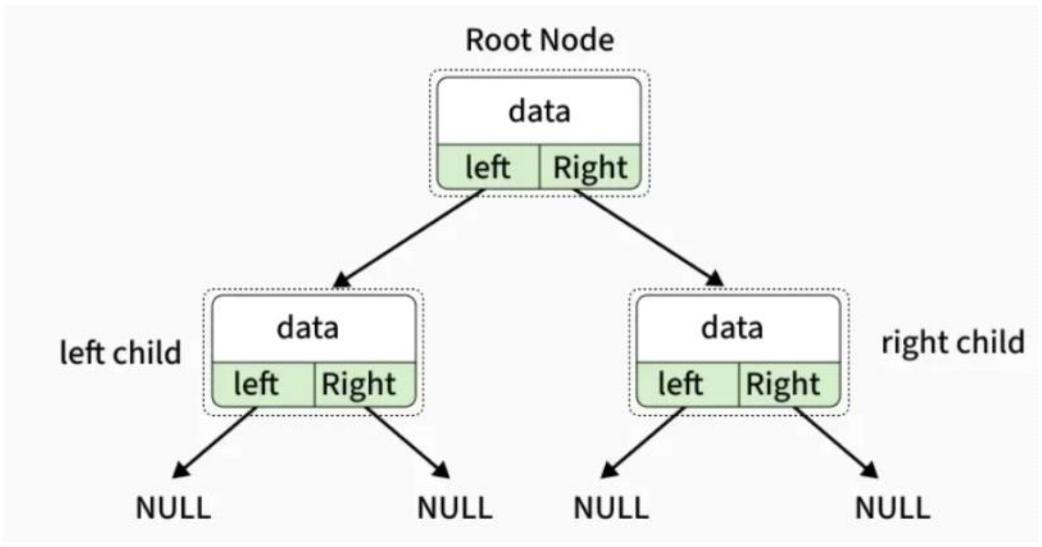
Binary Tree is a non-linear and hierarchical data structure where each node has **at most two children** referred to as the left child and the right child. The topmost node in a binary tree is called the root, and the bottom-most nodes(having no children) are called leaves.



Representation of Binary Tree

Each node in a Binary Tree has three parts:

- Data
- Pointer to the left child
- Pointer to the right child



Properties of Binary Tree

- The maximum number of nodes at level **L** of a binary tree is **2^L.**
- The maximum number of nodes in a binary tree of height **H** is **$2^{H+1} - 1$.**
- Total number of leaf nodes in a binary tree = total number of nodes with 2 children + 1.
- In a Binary Tree with **N** nodes, the minimum possible height or the minimum number of levels is **$\lfloor \log_2 N \rfloor$.**
- A Binary Tree with **L** leaves has at least **$\lceil \log_2 L \rceil + 1$** levels.

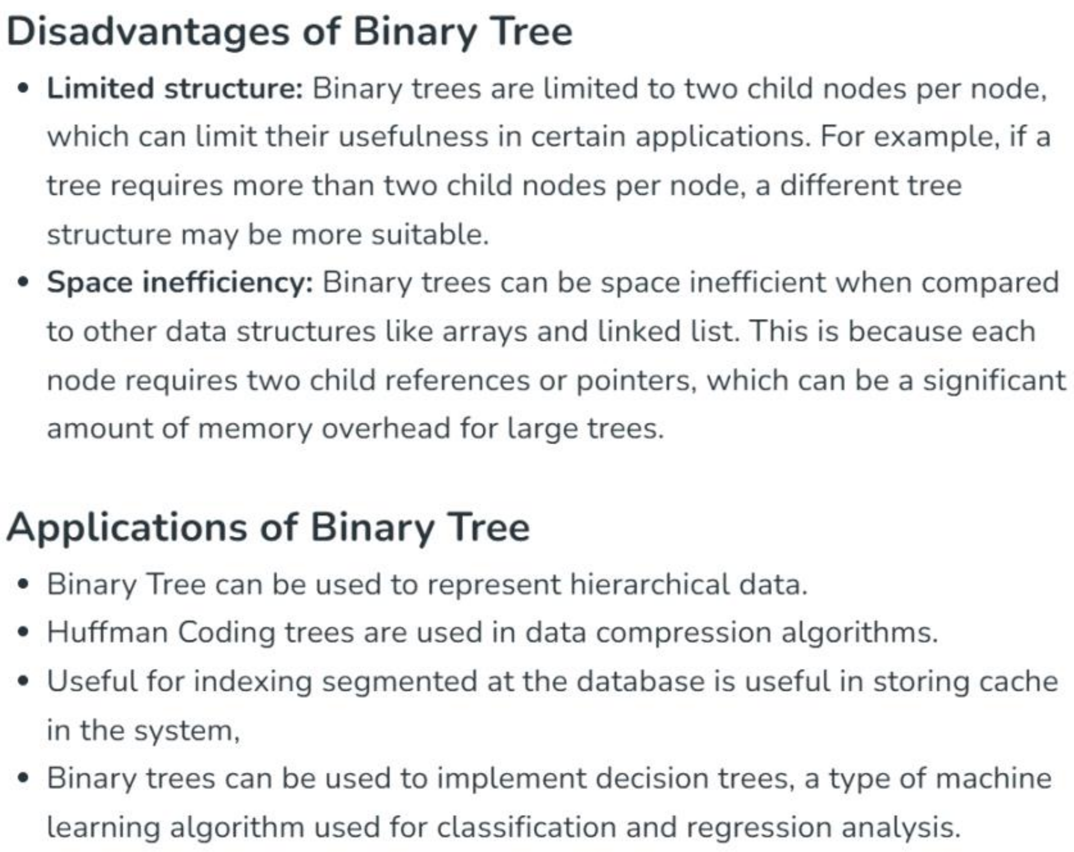
Operations On Binary Tree

Following is a list of common operations that can be performed on a binary tree:

1. Traversal: Depth-First Search (DFS) Traversal and Breadth-First Search (BFS) Traversal
2. Search: Search a node in Binary Tree
3. Insertion and Deletion: Prerequisite: Level Order Traversal, Insert in a Binary Tree and Delete from a Binary Tree

Advantages of Binary Tree

- **Efficient Search:** Binary Search Trees (a variation of Binary Tree) are efficient when searching for a specific element, as each node has at most two child nodes when compared to linked list and arrays
- **Memory Efficient:** Binary trees require lesser memory as compared to other tree data structures, therefore memory-efficient.
- Binary trees are relatively easy to implement and understand as each node has at most two children, left child and right child.



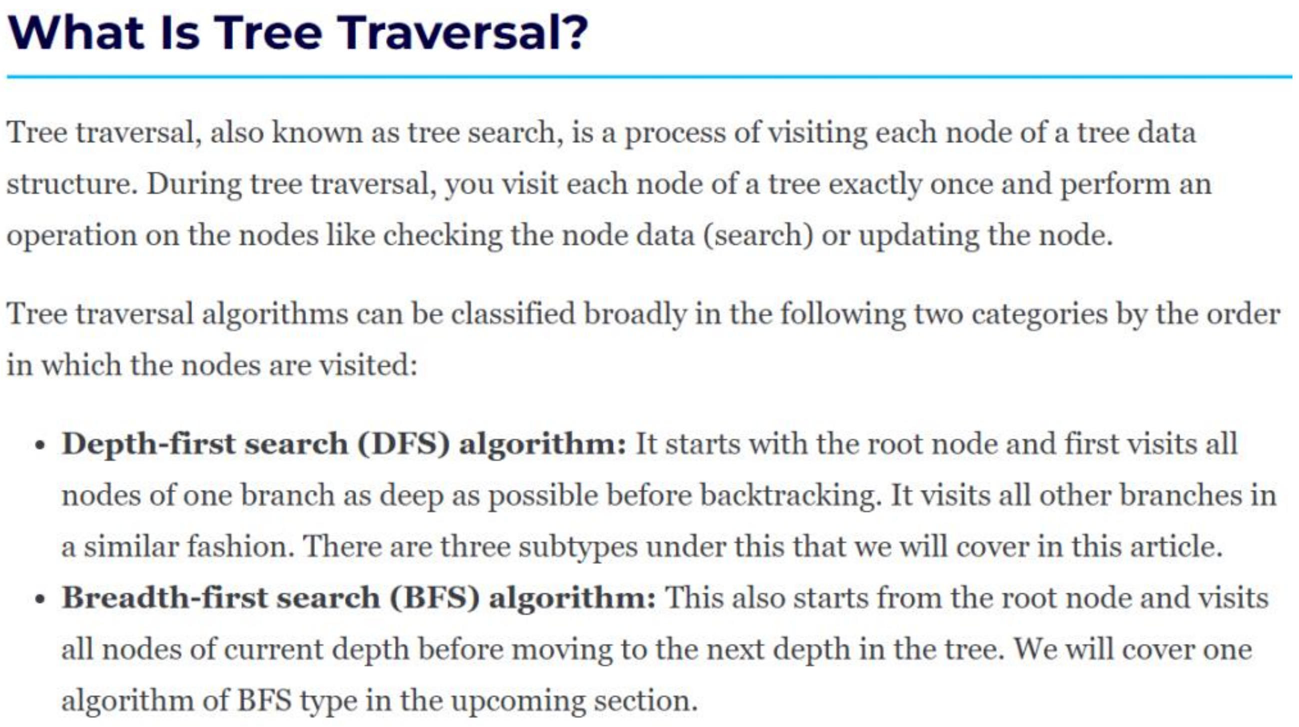
Disadvantages of Binary Tree

- **Limited structure:** Binary trees are limited to two child nodes per node, which can limit their usefulness in certain applications. For example, if a tree requires more than two child nodes per node, a different tree structure may be more suitable.
- **Space inefficiency:** Binary trees can be space inefficient when compared to other data structures like arrays and linked list. This is because each node requires two child references or pointers, which can be a significant amount of memory overhead for large trees.

Applications of Binary Tree

- Binary Tree can be used to represent hierarchical data.
- Huffman Coding trees are used in data compression algorithms.
- Useful for indexing segmented at the database is useful in storing cache in the system,
- Binary trees can be used to implement decision trees, a type of machine learning algorithm used for classification and regression analysis.

https://builtin.com/software-engineering-perspectives/tree-traversal

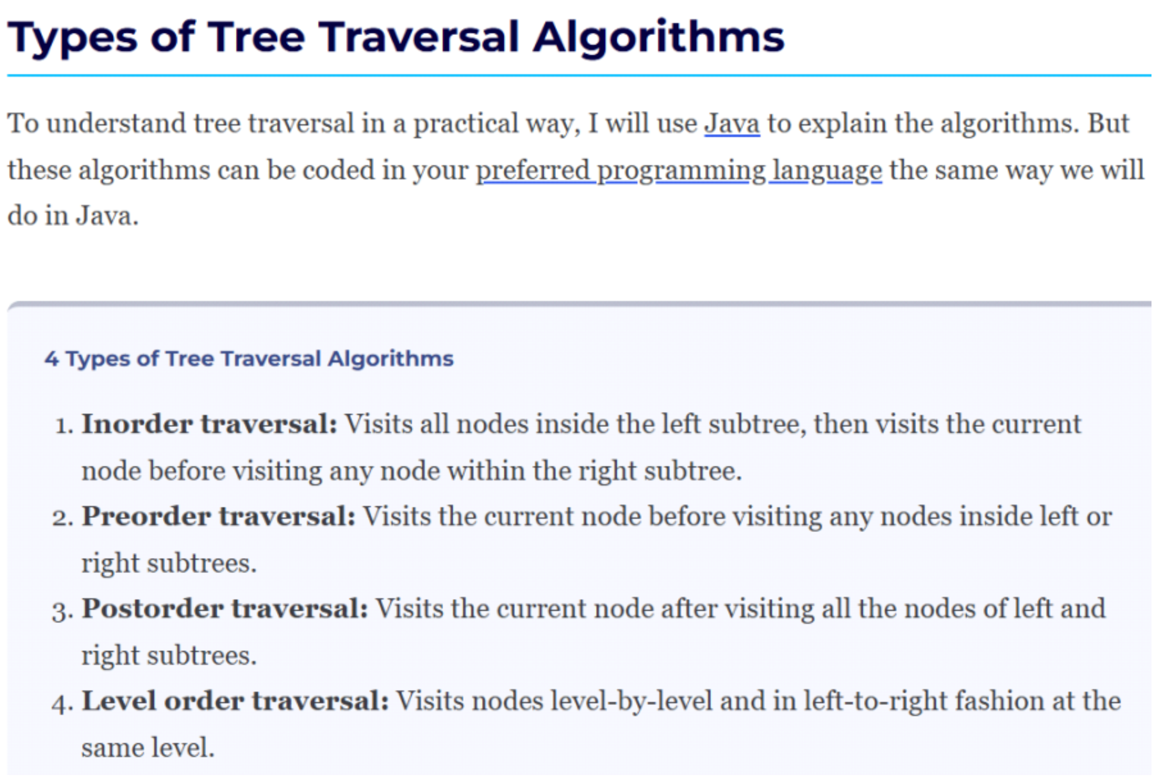


What Is Tree Traversal?

Tree traversal, also known as tree search, is a process of visiting each node of a tree data structure. During tree traversal, you visit each node of a tree exactly once and perform an operation on the nodes like checking the node data (search) or updating the node.

Tree traversal algorithms can be classified broadly in the following two categories by the order in which the nodes are visited:

- **Depth-first search (DFS) algorithm:** It starts with the root node and first visits all nodes of one branch as deep as possible before backtracking. It visits all other branches in a similar fashion. There are three subtypes under this that we will cover in this article.
- **Breadth-first search (BFS) algorithm:** This also starts from the root node and visits all nodes of current depth before moving to the next depth in the tree. We will cover one algorithm of BFS type in the upcoming section.



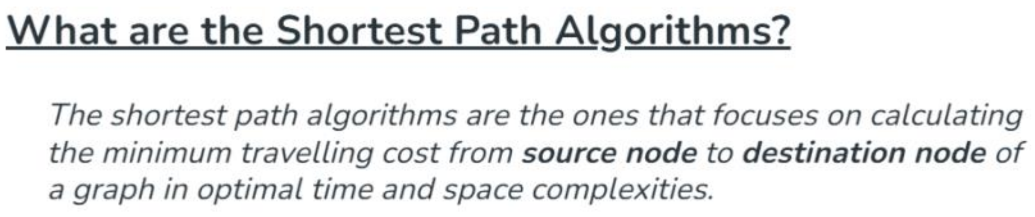
Types of Tree Traversal Algorithms

To understand tree traversal in a practical way, I will use Java to explain the algorithms. But these algorithms can be coded in your preferred programming language the same way we will do in Java.

4 Types of Tree Traversal Algorithms

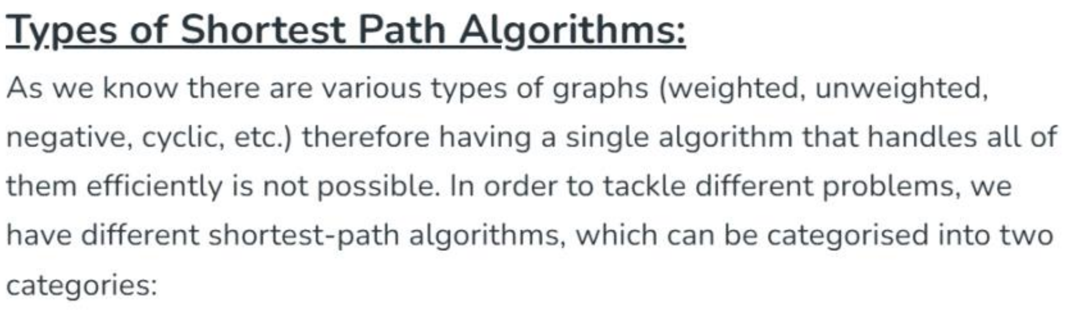
1. **Inorder traversal:** Visits all nodes inside the left subtree, then visits the current node before visiting any node within the right subtree.
2. **Preorder traversal:** Visits the current node before visiting any nodes inside left or right subtrees.
3. **Postorder traversal:** Visits the current node after visiting all the nodes of left and right subtrees.
4. **Level order traversal:** Visits nodes level-by-level and in left-to-right fashion at the same level.

https://www.geeksforgeeks.org/dsa/shortest-path-algorithms-a-complete-guide/



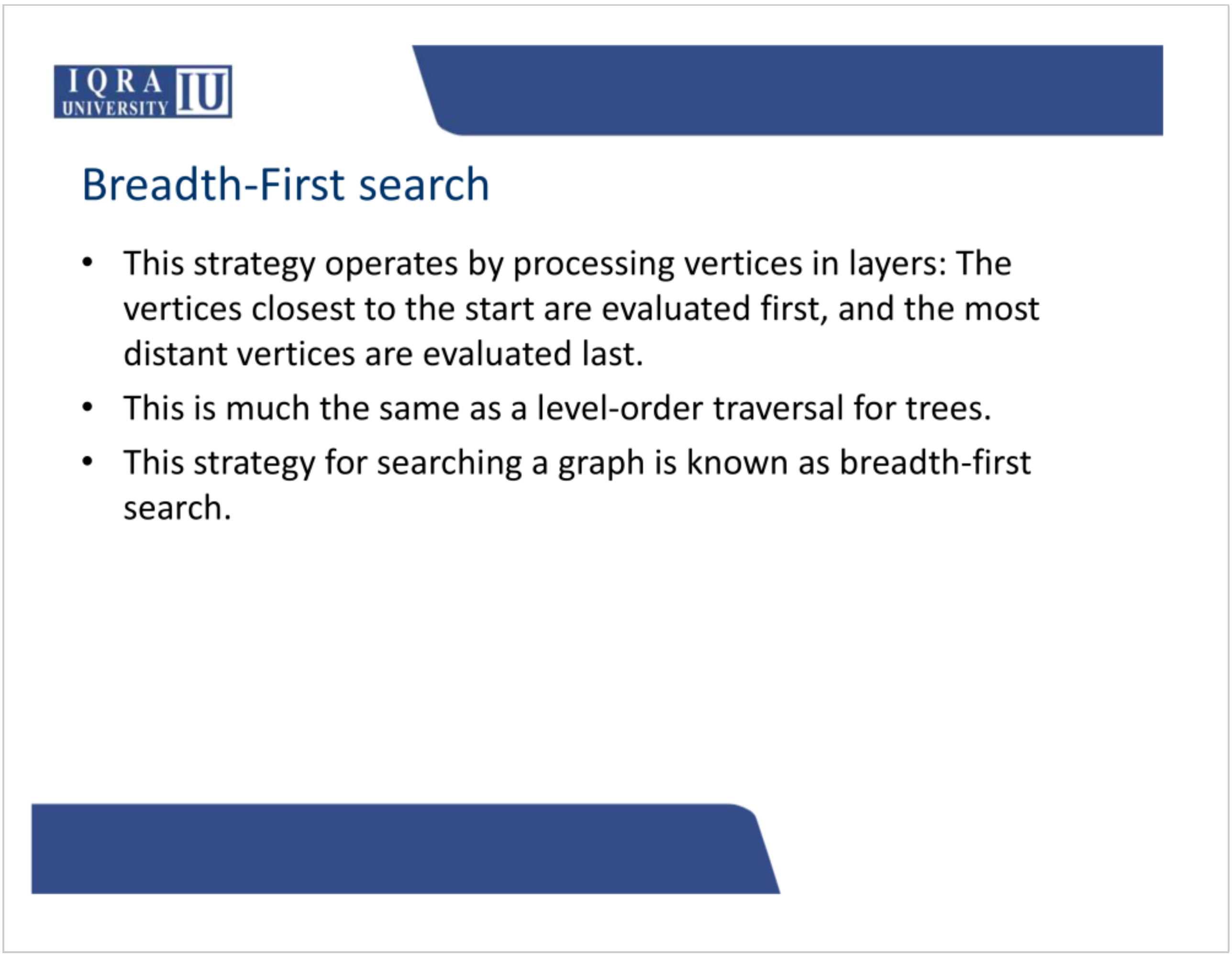
What are the Shortest Path Algorithms?

*The shortest path algorithms are the ones that focuses on calculating the minimum travelling cost from **source node** to **destination node** of a graph in optimal time and space complexities.*



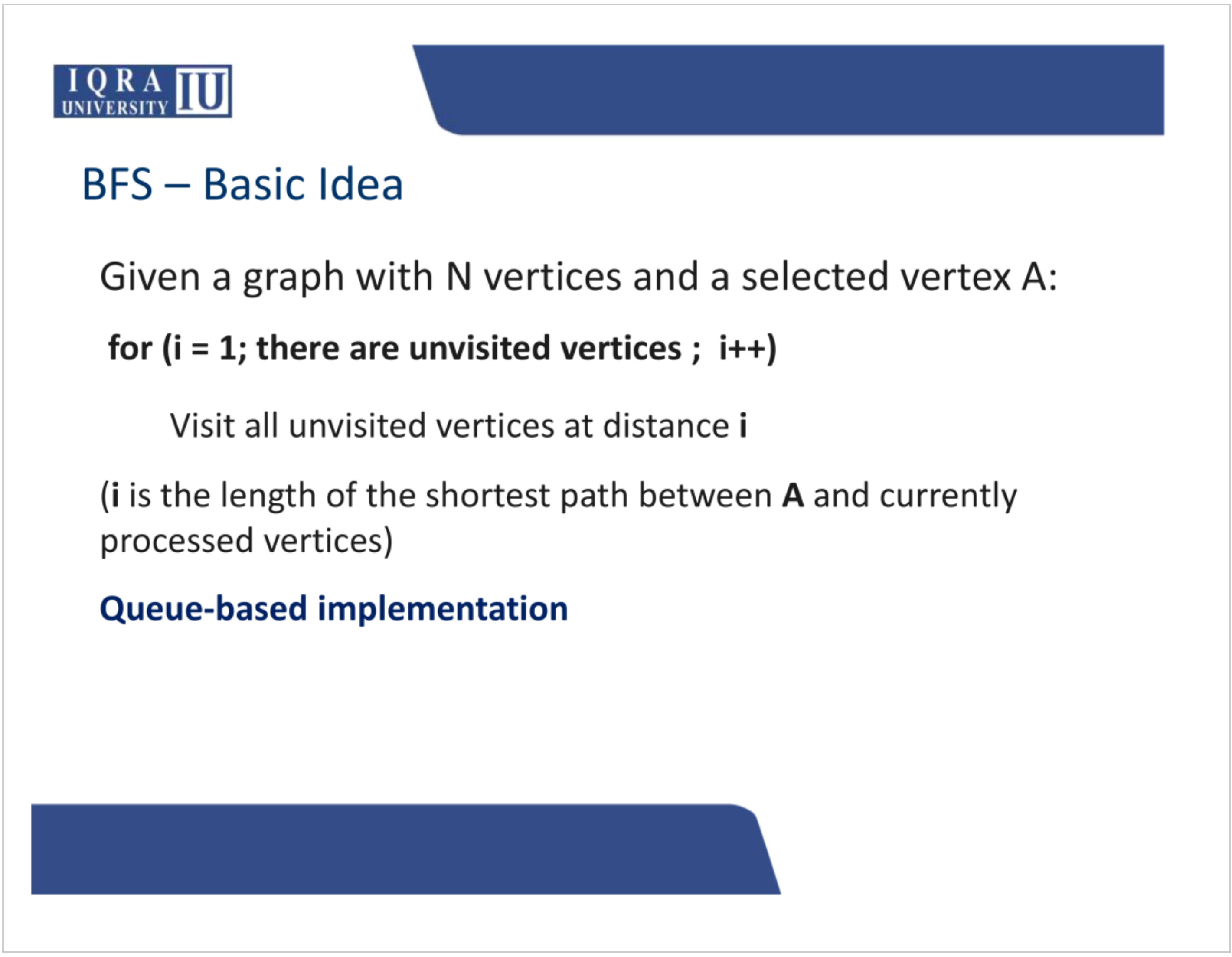
Types of Shortest Path Algorithms:

As we know there are various types of graphs (weighted, unweighted, negative, cyclic, etc.) therefore having a single algorithm that handles all of them efficiently is not possible. In order to tackle different problems, we have different shortest-path algorithms, which can be categorised into two categories:



Breadth-First search

- This strategy operates by processing vertices in layers: The vertices closest to the start are evaluated first, and the most distant vertices are evaluated last.
- This is much the same as a level-order traversal for trees.
- This strategy for searching a graph is known as breadth-first search.



BFS – Basic Idea

Given a graph with N vertices and a selected vertex A:

```
for (i = 1; there are unvisited vertices ;  i++)
        Visit all unvisited vertices at distance i
```

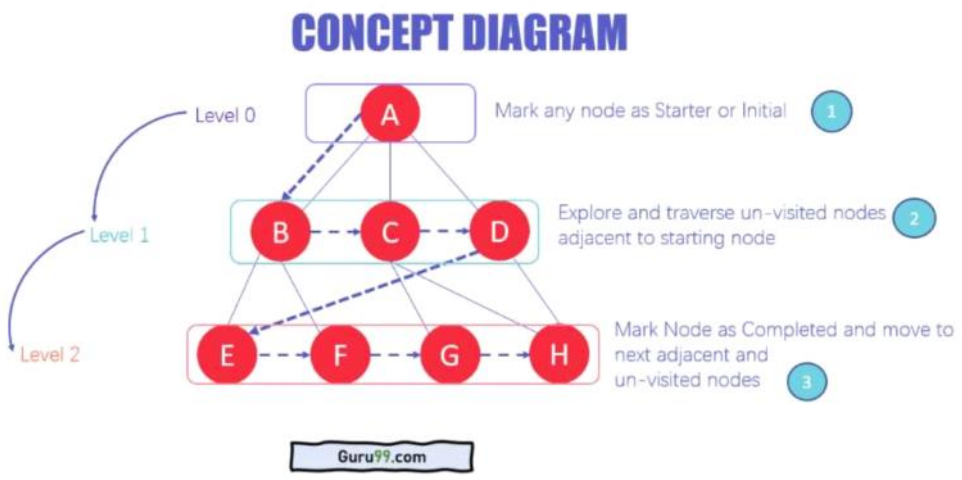
(**i** is the length of the shortest path between **A** and currently processed vertices)

Queue-based implementation

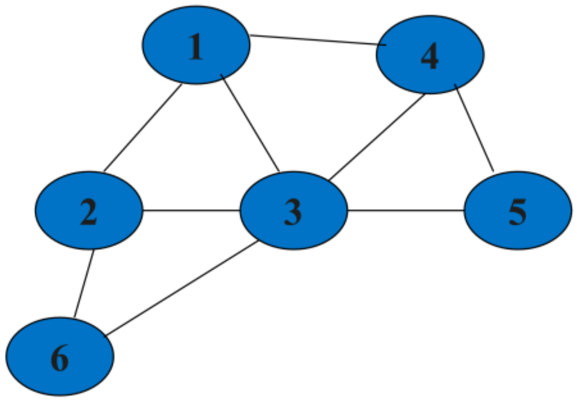
BFS – Algorithm

```
BFS algorithm
1. Store source vertex S in a queue and mark as processed
2. while queue is not empty
        Read vertex v from the queue
     for all neighbors w:
             If w is not processed
                   Mark as processed
                   Append in the queue
```

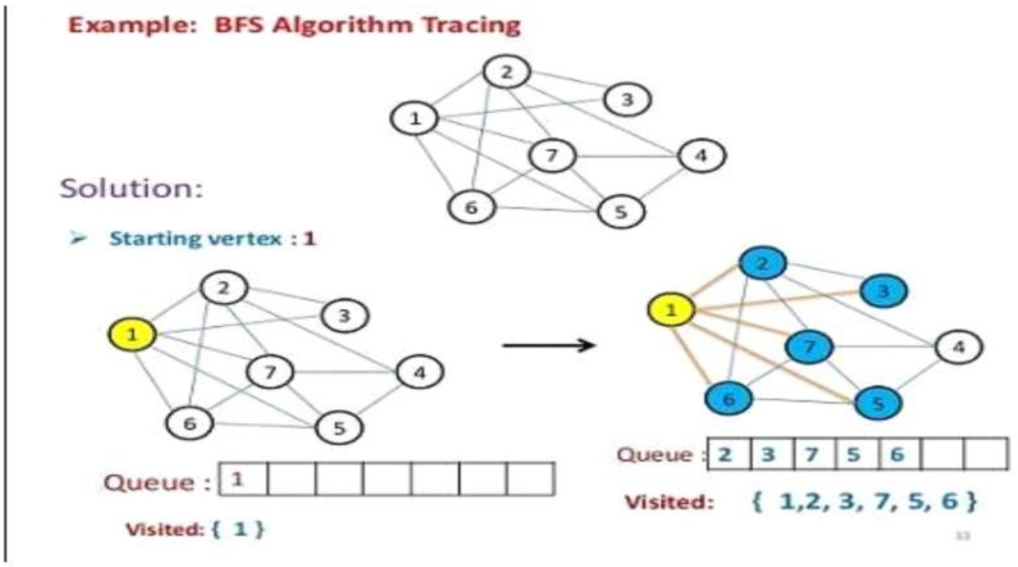
BFS


CONCEPT DIAGRAM
Level 0
Level 1
Level 2
A
B
C
D
E
F
G
H
Mark any node as Starter or Initial
1
Explore and traverse un-visited nodes adjacent to starting node
2
Mark Node as Completed and move to next adjacent and un-visited nodes
3
Guru99.com

BFS – Example



BFS



Example: BFS Algorithm Tracing

Solution:

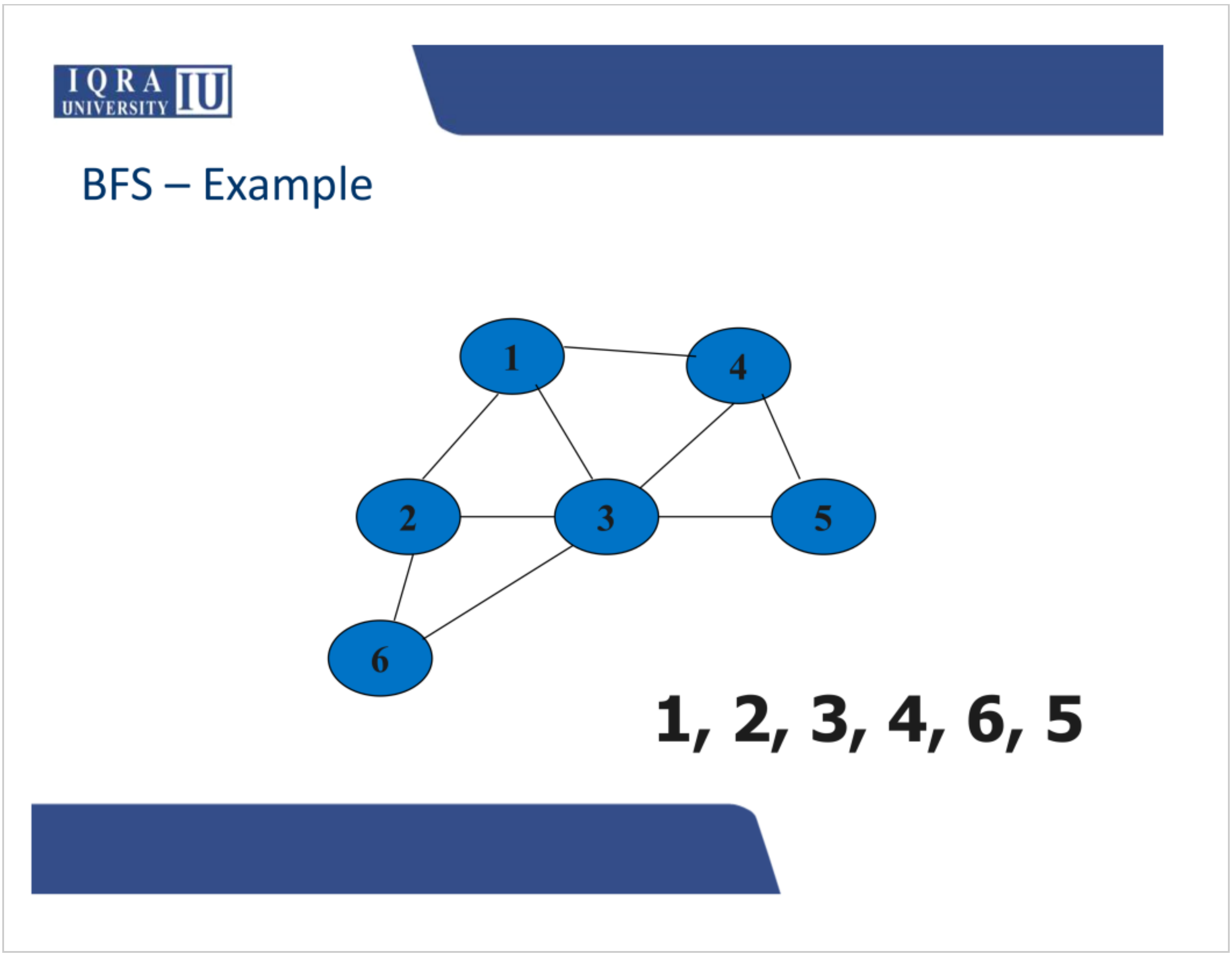
➢ Starting vertex : 1

Queue : | 1 | | | | | | |

Visited: { 1 }

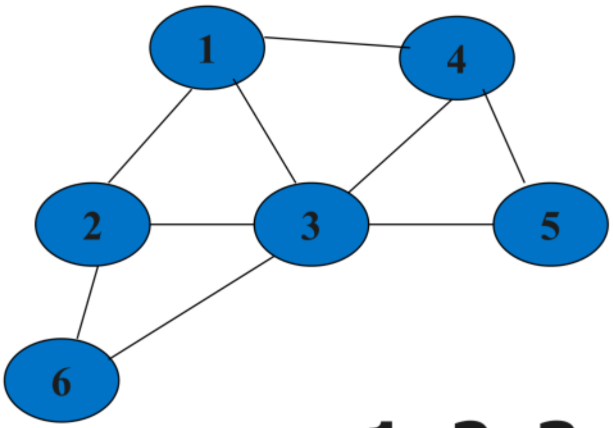
Queue : | 2 | 3 | 7 | 5 | 6 | | |

Visited: { 1,2, 3, 7, 5, 6 }



IQRA UNIVERSITY IU

BFS – Example



1, 2, 3, 4, 6, 5

IQRA UNIVERSITY IU

Example

Breadth-first traversal: 1, 2, 3, 4, 6, 5

1: starting node

2, 3, 4 : adjacent to 1

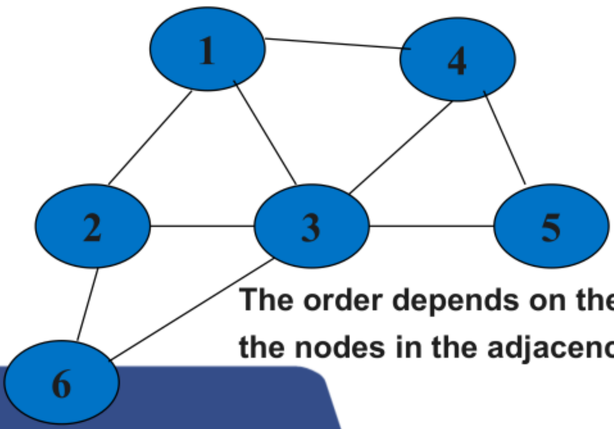
(at distance 1 from node 1)

6 : unvisited adjacent to node 2.

5 : unvisited, adjacent to node 3

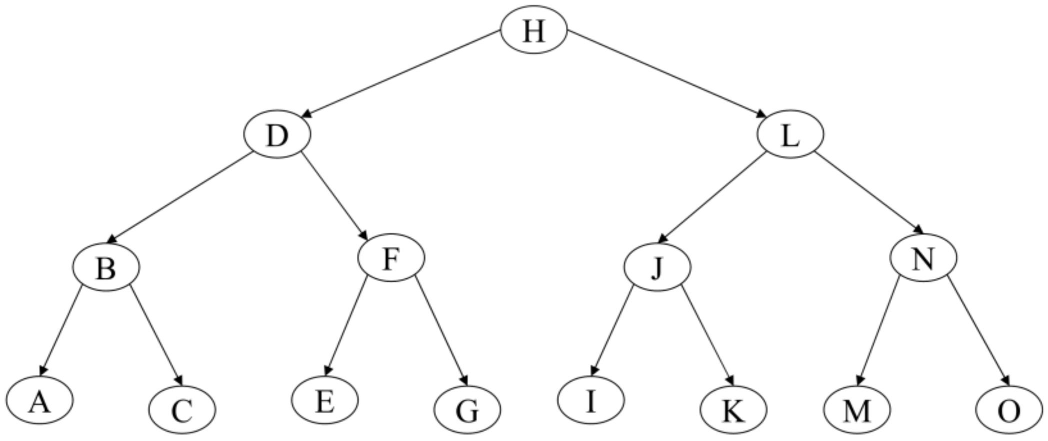
Adjacency lists

1: 2, 3, 4
2: 1, 3, 6
3: 1, 2, 4, 5, 6
4: 1, 3, 5
5: 3, 4
6: 2, 3



The order depends on the order of the nodes in the adjacency lists

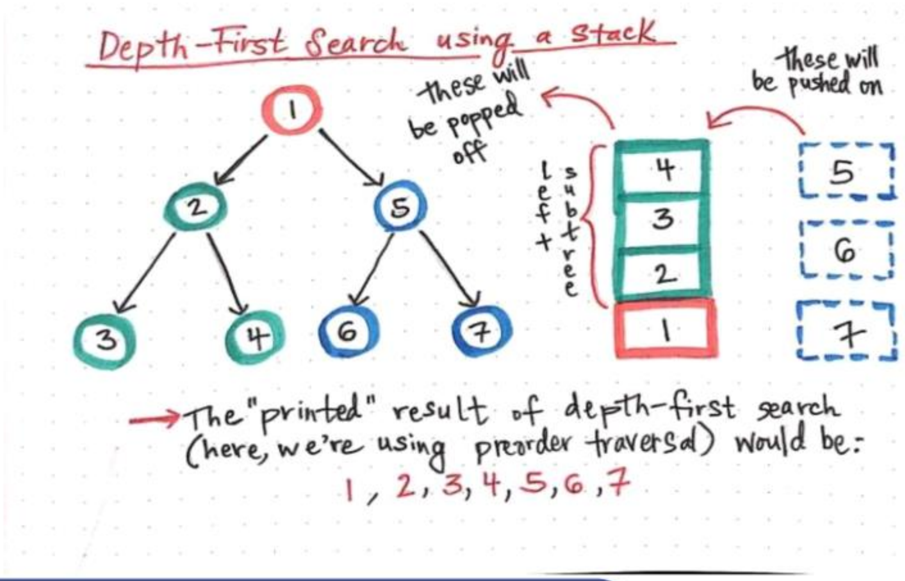
BFS in Tree – Example



H	D	L	B	F	J	N	A	C	E	G	I	K	M	O

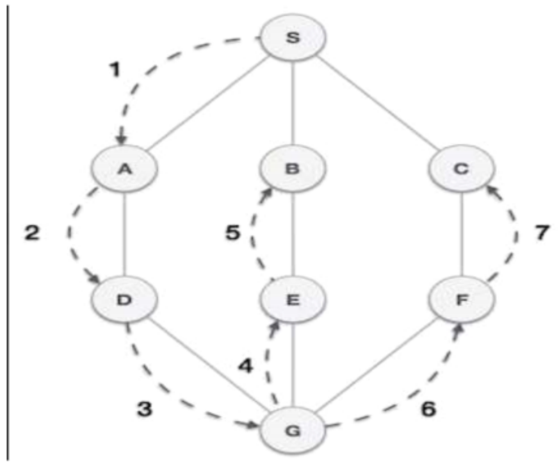
Depth-First Search

- Starting at some vertex, v, we process v and then recursively traverse all vertices adjacent to v.
- Depth-first search is a generalization of preorder traversal.
- Recursive in nature.



Depth-First Search using a Stack
these will be popped off
these will be pushed on
left subtree
1
2
3
4
5
6
7
4
3
2
1
5
6
7
The "printed" result of depth-first search (here, we're using preorder traversal) would be:
1, 2, 3, 4, 5, 6, 7

DFS



DFS – Algorithm

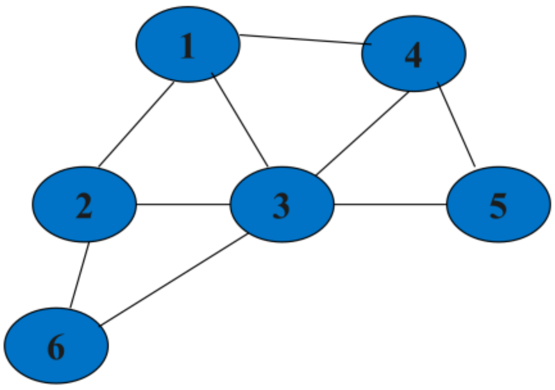
mark all vertices in the graph as not reached
invoke **dfs** on source node s

```
Procedure dfs (s)
          mark and visit s
          for each neighbor w of s
                    if the neighbor is not reached
                              invoke dfs(w)
```

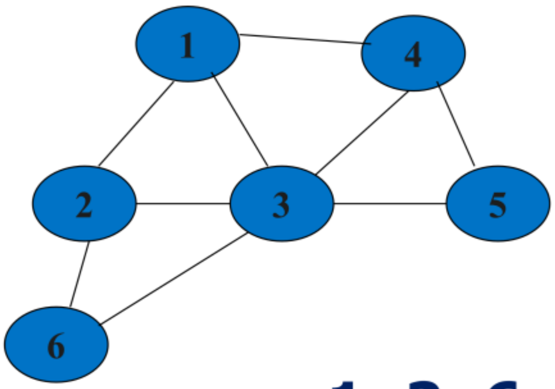
DFS – Algorithm

```
void dfs( Vertex v )
{
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}
```

DFS – Example



DFS – Example



1, 2, 6, 3, 5, 4

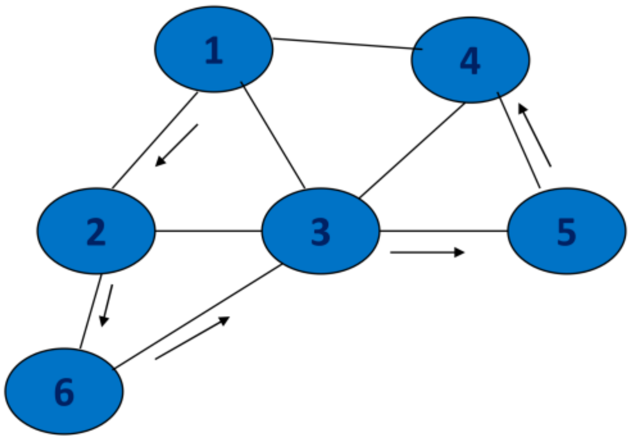
Example

Depth first traversal: 1, 2, 6, 3, 5, 4

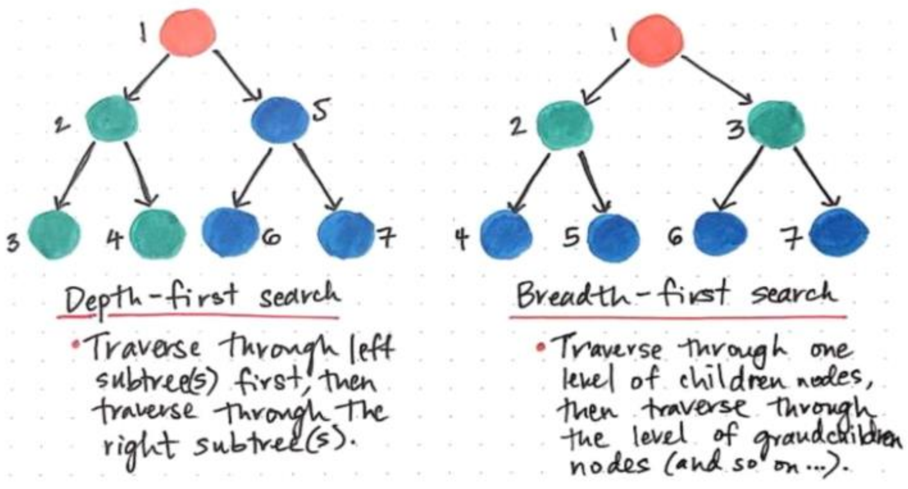
the particular order is dependent on the order of nodes in the adjacency lists

Adjacency lists

1: 2, 3, 4
2: 6, 3, 1
3: 1, 2, 6, 5, 4
4: 1, 3, 5
5: 3, 4
6: 2, 3



DFS & BFS


1
2
5
3
4
6
7
Depth-first search
•Traverse through left subtree(s) first, then traverse through the right subtree(s).
1
2
3
4
5
6
7
Breadth-first search
•Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

BFS & DFS Complexity

- The Time complexity of both BFS and DFS will be

  $O(V + E)$

where V is the number of vertices, and E is the number of Edges

- Memory requirement for BFS is higher than DFS

Applications of DFS

- Trees: preorder traversal
- Graphs:
  - Connectivity
  - Biconnectivity – articulation points
  - Euler graphs

Connectivity

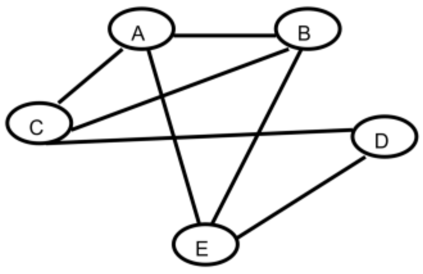
Definition:

An undirected graph is said to be **connected**

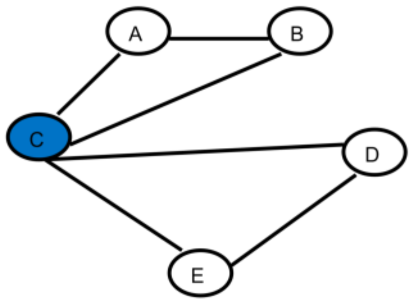
- if for any pair of nodes of the graph, the two nodes are reachable from one another (i.e. there is a path between them).
- If starting from any vertex we can visit all other vertices, then the graph is connected

Biconnectivity

- A graph is **biconnected**, if there are no vertices whose removal will disconnect the graph.



biconnected



not biconnected