Data Structures Lab 12

Course: Data Structures (CL2001) **Semester:** Fall 2024

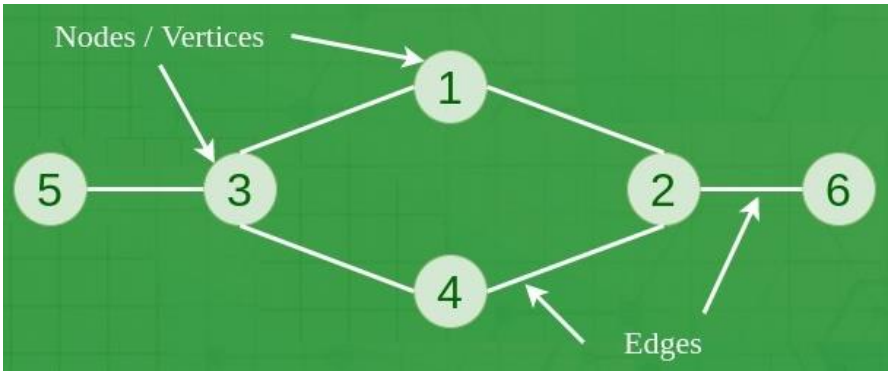
Instructor: Muhammed Nouman Hanif

Note:

- Lab manual cover following topics
  {Graph & Matching Patterns -> Brute Force, Rabin-Karp, Boyer Moore, Knuth Morris Pratt (KMP)}
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Graph Data Structure

A **Graph** is a **non-linear data structure** consisting of **vertices and edges**. The vertices are sometimes also referred to as **nodes** and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by **G (E, V)**.



Components of a Graph:

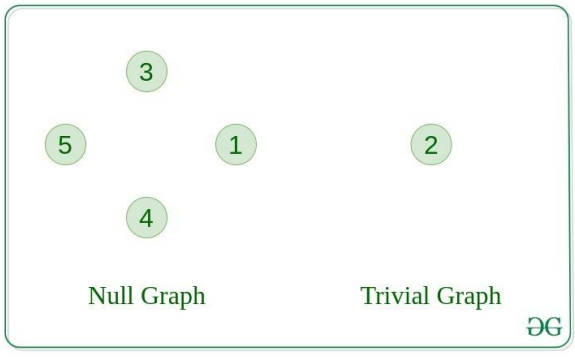
Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as **vertex or nodes**. Every node/vertex can be **labelled or unlabelled**.

Edges: Edges are drawn or used to **connect two nodes of the graph**. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be **labelled/unlabelled**.

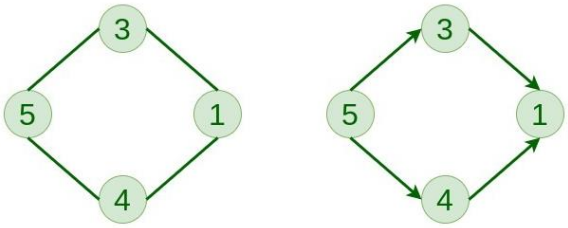
Types of Graphs:

1. Null Graph: A graph is known as a null graph if there are **no edges in the graph**.

2. Trivial Graph: Graph having only a **single vertex**, it is also the smallest graph possible.

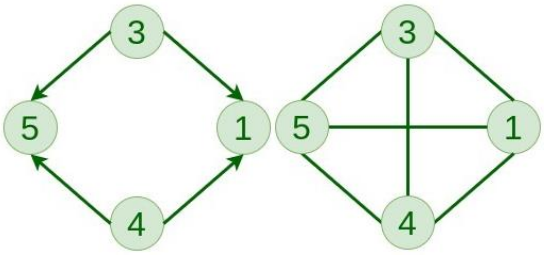


3. Undirected Graph: A graph in which **edges do not have any direction**. That is the nodes are unordered pairs in the definition of every edge.
4. Directed Graph: A graph in which **edge has direction**. That is the nodes are ordered pairs in the definition of every edge.

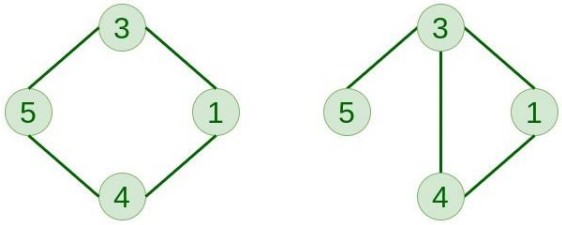


Undirected Graph Directed Graph

6. Directed Acyclic Graph: A Directed Graph that **does not contain any cycle**.
5. Complete Graph: The graph in which from each node there is an **edge to each other node**.

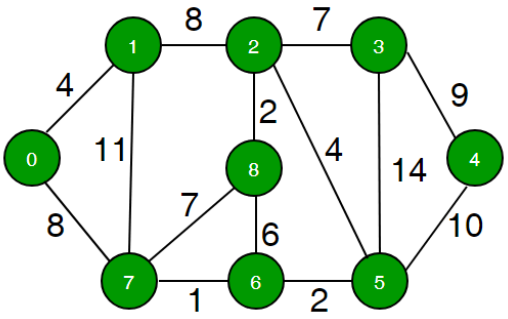


7. Cycle Graph: The graph in which the graph is a **cycle in itself**, the degree of each vertex is 2.
8. Cyclic Graph: A graph containing **at least one cycle** is known as a Cyclic graph.



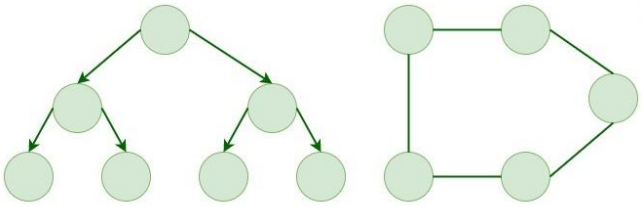
Cycle Graph Cyclic Graph

9. Weighted Graph: A graph in which the edges are already specified with **suitable weight** is known as a weighted graph. Weighted graphs can be further classified as **directed weighted graphs and undirected weighted graphs**.



Tree v/s Graph

Trees are the **restricted types of graphs**, just with **some more rules**. **Every tree will always be a graph** but **not all graphs will be trees**. Linked List, Trees, and Heaps all are **special cases of graphs**.



Tree Graph

Representation of Graphs:

There are two ways to store a graph: -

1. Adjacency Matrix (**Dense Graph**)
2. Adjacency List (**Sparse Graph**)

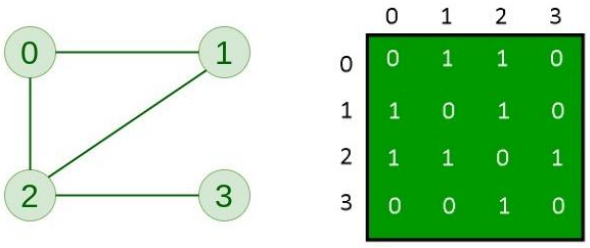
1. Adjacency Matrix:

In this method, the graph is stored in the form of the **2D matrix** where rows and columns denote vertices. Each entry in the matrix represents the **weight of the edge** between those vertices.

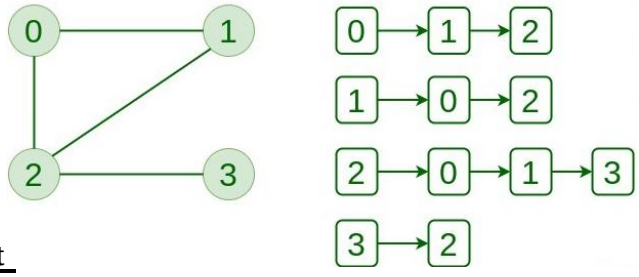
2. Adjacency List:

This graph is represented as a collection of linked lists. There is an **array of pointer** which points to the edges connected to that vertex.

Adjacency Matrix of Graph



Adjacency List of Graph



Comparison between Adjacency Matrix and Adjacency List

When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty. An algorithm such as Prim's and Dijkstra adjacency matrix is used to have less complexity.

Action	Adjacency Matrix	Adjacency List
Adding Edge	O(1)	O(1)
Removing an edge	O(1)	O(N)
Initializing	O(N*N)	O(N)

Basic Operations on Graphs:

Below are the basic operations on the graph:

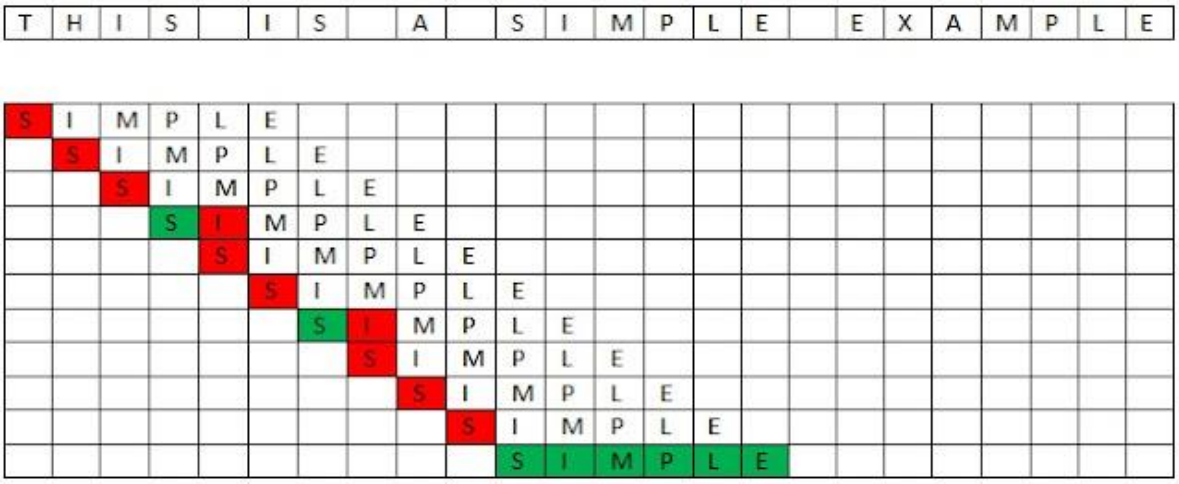
- Insertion of Nodes/Edges in the graph – Insert a node into the graph.
- Deletion of Nodes/Edges in the graph – Delete a node from the graph.
- Searching on Graphs – Search an entity in the graph.
- Traversal of Graphs – Traversing all the nodes in the graph

Matching Pattern (Pattern Searching) Algorithm

Pattern Matching is widely used in computer science and many other fields. Pattern Matching algorithms are used to **search for patterns within a larger text or data set**. In C++, strings are sequences of characters stored in a char array. Matching a pattern in a string involves searching for a specific sequence of characters (the pattern) within a given string. Efficient string search algorithms reduce the computational overhead of naive matching methods.

Brute Force Pattern Matching Algorithm:

Brute Force Pattern Matching is the simplest Pattern Matching Algorithm. It involves comparing the characters of the pattern with the **characters of the text one by one**. If all the characters match, the algorithm returns the **starting position of the pattern in the text**. If not, the algorithm moves to the next position in the text and repeats the comparison until a match is found or the end of the text is reached. The Time Complexity of the Brute Force Algorithm is **O(M x N)**, where **M** denotes the length of the text and **N** denotes the length of the pattern.



Concept: It is the simplest pattern-matching algorithm. It compares the pattern with every possible substring in the text.

Steps:

- Start from the first character of the text.
- Compare each character of the pattern with the corresponding characters in the text.
- If a mismatch occurs, shift the pattern one position forward and repeat.

Algorithm:

1. **Loop through each character in the text**.
2. For each position in the text, **compare the substring of the same length as the pattern**.
3. If the pattern matches, return the starting index.

Rabin-Karp Algorithm:

Rabin-Karp algorithm check every substring and matches the **hash value** of the **pattern** with the **hash value** of the current substring of **text**, and if the **hash values** match then only it starts matching individual characters. So, Rabin Karp algorithm needs to calculate hash values for the following strings.

- **Pattern** itself
- All the substrings of the **text** of length **m** which is the size of pattern.

The average and best-case running time of the Rabin-Karp algorithm is **O(n+m)**, but its worst-case time is **O(nm)**.

$$\textbf{hash} = (\textbf{hash} - (\textbf{text[i - pattern\_length]} * (b^{\textbf{pattern\_length} - 1})) \% p) * b + \textbf{text[i]}$$

where, prime number 'p' as the modulus, base 'b' usually a prime number as well, initial hash value 'hash' to 0

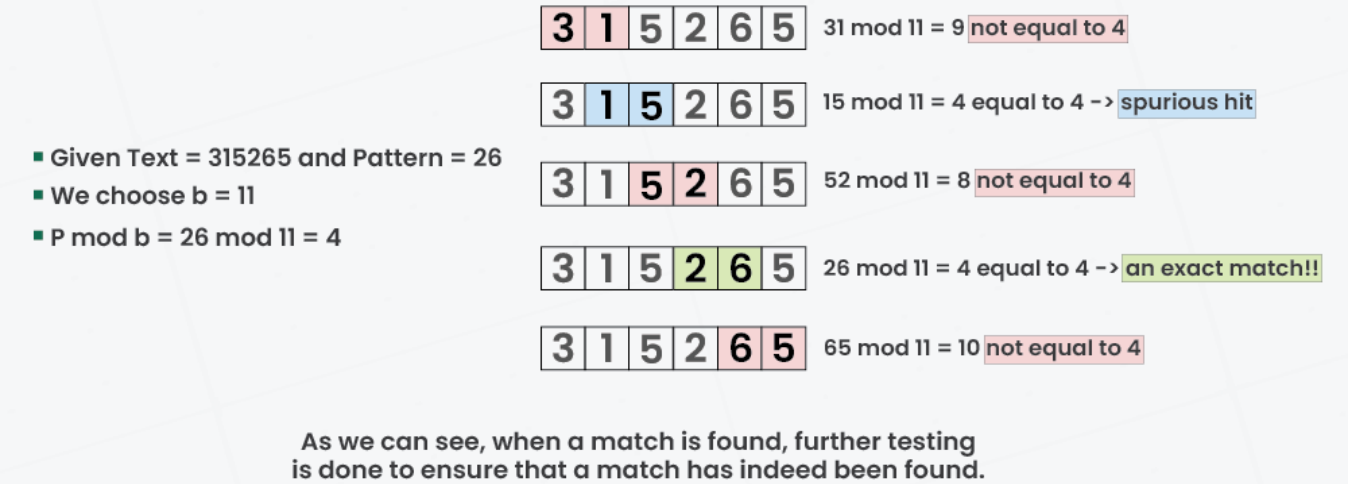
Concept: It uses a hash function to find a pattern in the text, avoiding character-by-character comparisons.

Steps:

- Compute the hash value of the pattern and the initial substring of the text (of the same length as the pattern).
- Compare hash values:
  - If they match, compare characters to confirm.
  - If they do not match, slide the pattern one character forward and update the hash.

Algorithm:

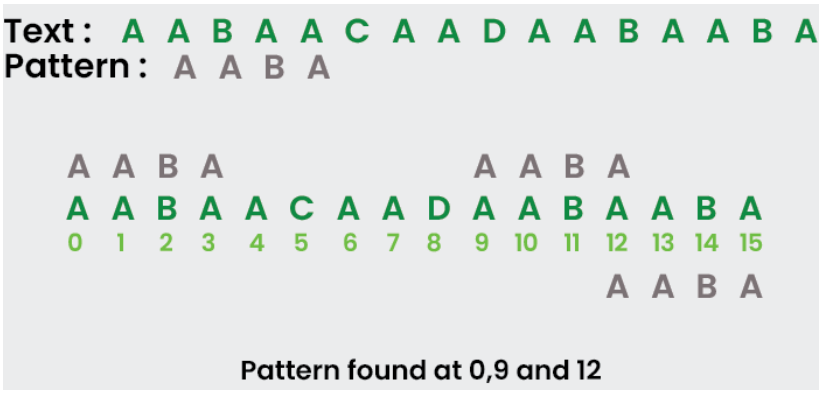
1. Compute the hash of the pattern.
2. Compute the hash for each substring of the text of the same length.
3. Compare the hash values to find a match.
4. If a match is found, compare the actual characters for verification.



Boyer-Moore Algorithm:

One of the most popular Pattern Matching algorithms is the **Boyer-Moore** algorithm. This algorithm was first published in 1977 by Robert S. Boyer and J Strother Moore. The **Boyer-Moore** algorithm compares a pattem with a **larger set of data or text from right to left** instead of left to right, as with most other pattern matching algorithms.

The **Boyer-Moore** algorithm has two main components: the **bad character rule** and the **good suffix rule**. The bad character rule works by **comparing the character in the pattern with the corresponding character in the data or text**. If the characters do not match, the algorithm moves the pattern to the right until it finds a character that matches. The good suffix rule compares the **suffix of the pattern with the corresponding suffix of the data or text**. If the suffixes do not match, the algorithm moves the pattern to the right until it finds a matching suffix. The **Boyer-Moore** algorithm is known for its efficiency and is widely used in many applications. It is considered one of the fastest patterns matching algorithms available.



Concept: It improves efficiency by skipping unnecessary comparisons using two heuristics: Bad Character and Good Suffix.

Steps:

- Start comparing the pattern from the end with the text.
- If a mismatch occurs:
  - Use the *Bad Character Rule* to skip characters based on the mismatched character.
  - Use the *Good Suffix Rule* to shift based on matched suffixes.

Algorithm:

- Create a **bad-character table for quick mismatches**.
- Loop through the text and try to match the pattern.
- If a mismatch occurs, use the bad-character table to shift the pattern by some positions.

Knuth-Morris-Pratt Algorithm:

The **Knuth-Morris-Pratt (KMP)** algorithm is a more advanced Pattern Matching algorithm. It is based on the **observation that when a mismatch occurs**, some information about the text and the pattern can be used to **avoid unnecessary comparisons**. The algorithm **precomputes a table** (**Longest Proper Prefix** which is also **Suffix**) that contains information about the pattern. The table determines how many characters of the pattern can be **skipped when a mismatch occurs**. The Time Complexity of the **KMP** algorithm is **O(M+N)**.

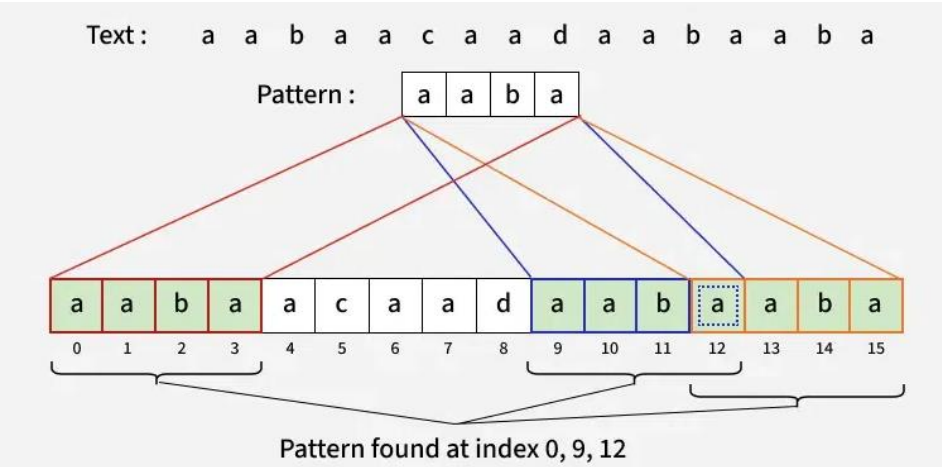
Concept: It preprocesses the pattern to avoid redundant comparisons by using a Partial Match Table (LPS array).

Steps:

- Preprocess the pattern to create the LPS array, which indicates the longest prefix which is also a suffix.
- Compare the pattern with the text:
  - If characters match, continue.
  - If a mismatch occurs, use the LPS array to skip unnecessary comparisons.

Algorithm:

1. Build the **prefix table by checking the longest prefix of the pattern** that matches a suffix.
2. Loop through the text to find a match.
3. Use the **prefix table for efficient jumps when mismatches occur**.



Reference Lab

Breadth First Search

The **Breadth First Search (BFS)** algorithm is used **to search a graph** data structure for a node that meets a set of criteria. It starts at **the root of the graph** and visits all nodes at the current level before moving on to the nodes at the next level.

Relation between BFS for Graph and Tree traversal:

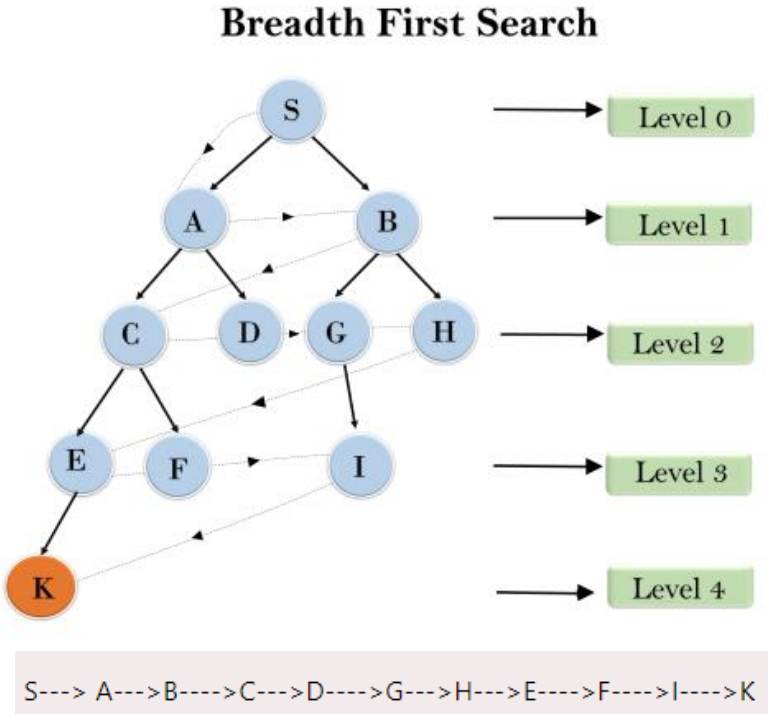
Breadth-First Traversal (level order traversal) for a graph is similar to the Breadth-First Traversal of a tree. The only catch here is, that, unlike trees, **graphs may contain cycles**, so we may come to the same node again. To **avoid processing a node more than once**, we divide the vertices into two categories:

1. **Visited and**
2. **Not visited.**

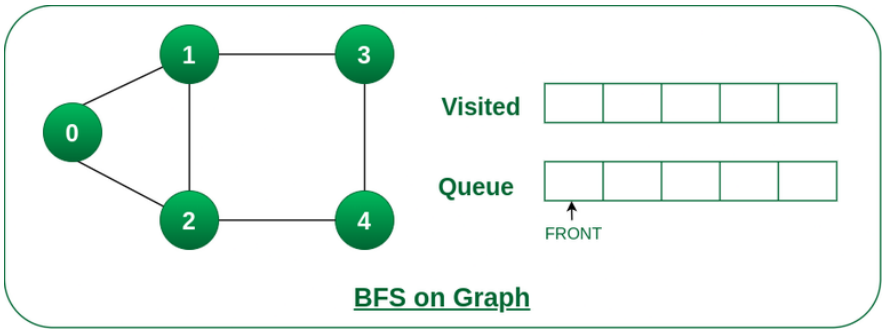
A **Boolean visited array** is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses **a queue data structure** for traversal.

How does BFS work?

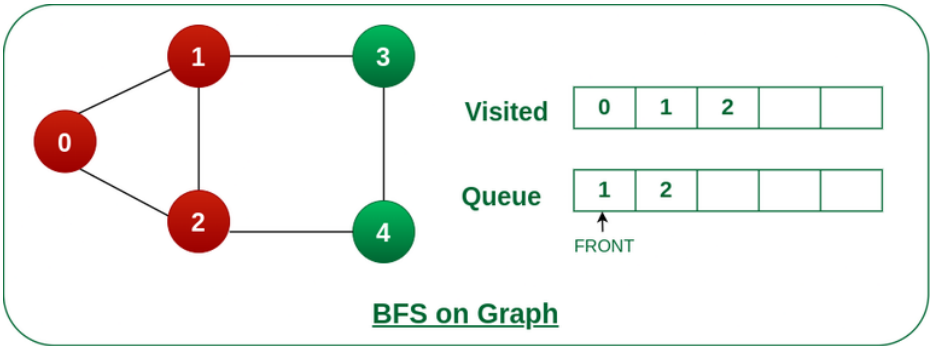
Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited. To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.



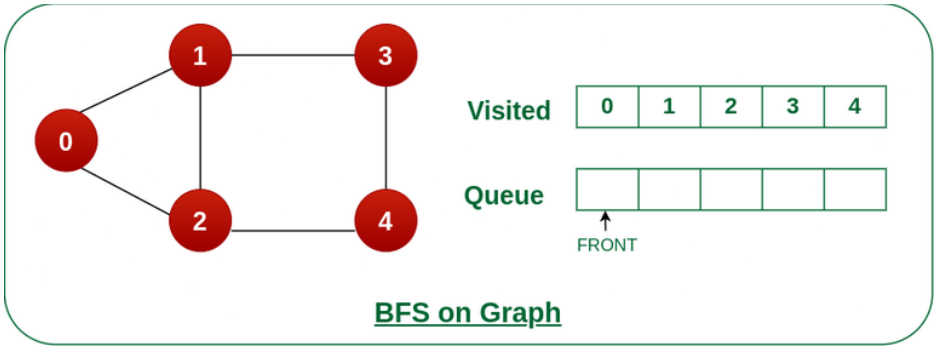
Example:



Step1: Initially queue and visited arrays are empty.
Step2: Push node 0 into queue and mark it visited.
Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.
Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.
Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.
Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.



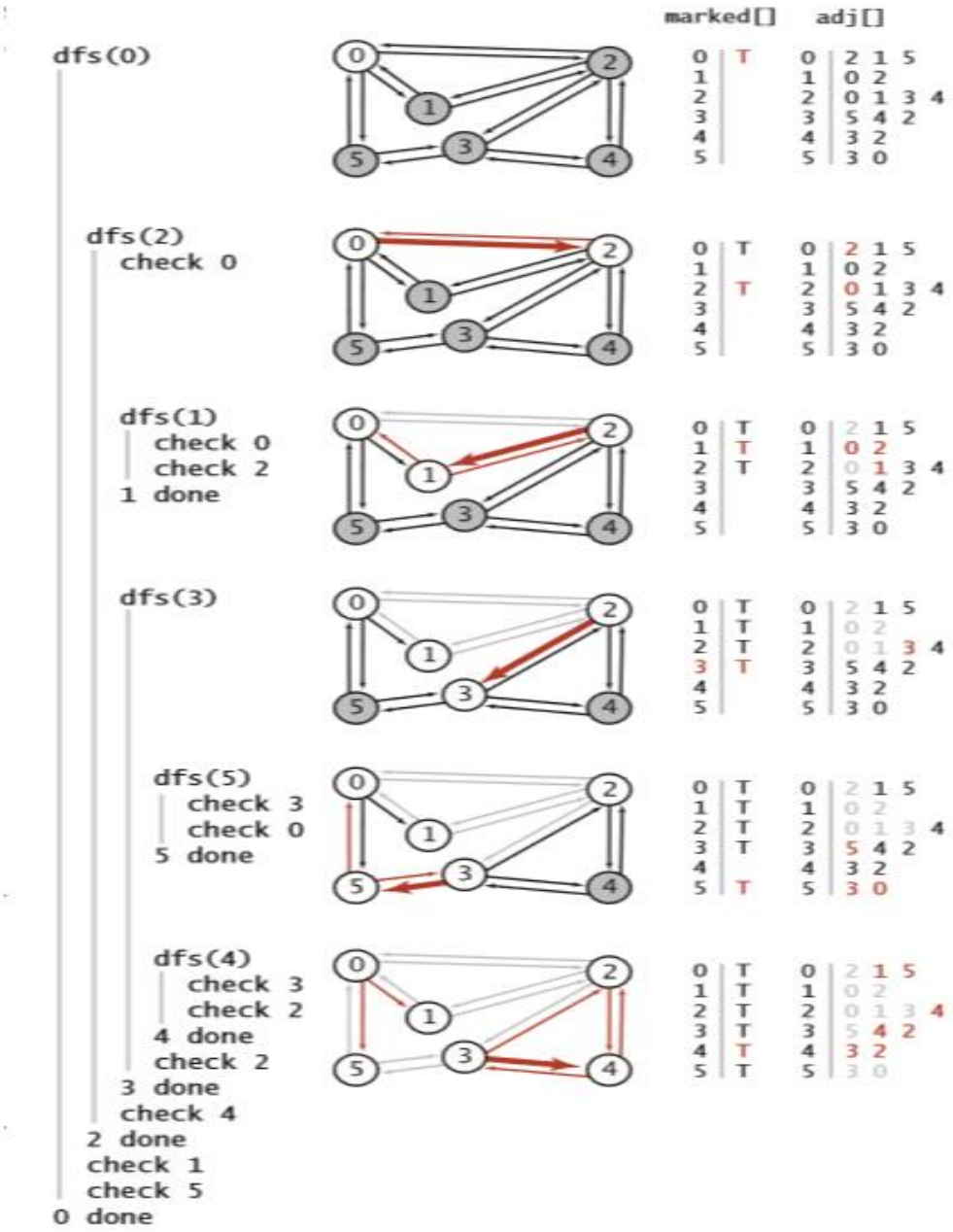
Now, Queue becomes empty, So, terminate these process of iteration.

Depth First Search

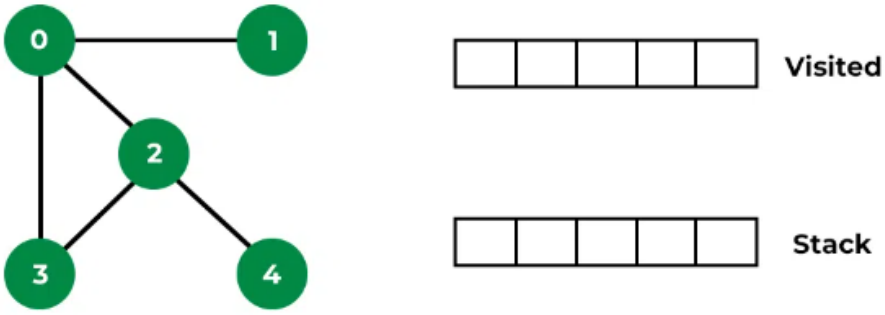
Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a **node may be visited twice**). To avoid processing a node more than once, use a Boolean visited array. A graph can have more than one DFS traversal.

How does DFS work?

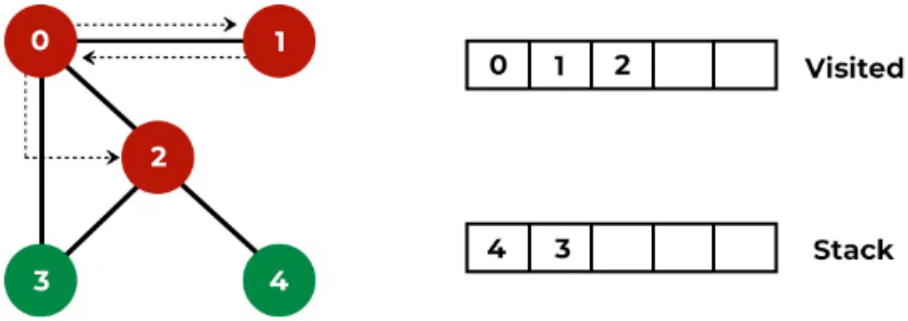
Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at **the root node** (selecting some arbitrary node as the root node in the case of a graph) and **explores as far as possible along each branch before backtracking**.



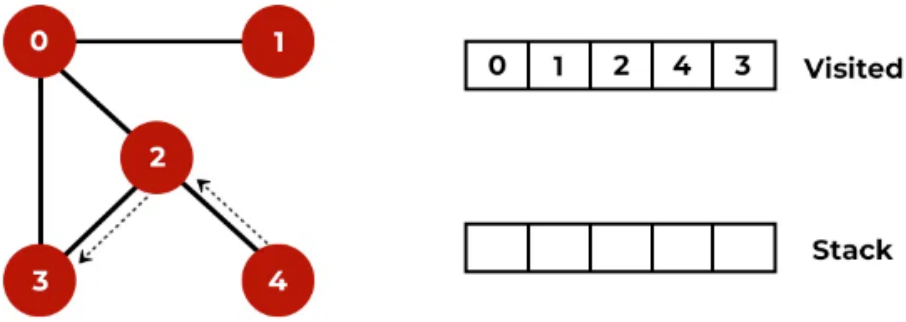
Example:



Step1: Initially stack and visited arrays are empty.
Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.
Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.
Step 4: Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.



Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.
Step 6: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

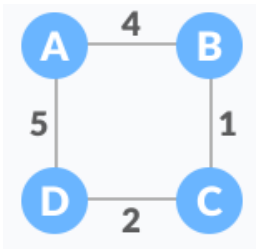


Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

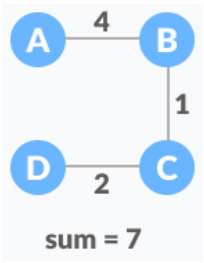
More Algorithms

Minimum Spanning Tree (Prim's Algorithm)

A minimum spanning tree is a spanning tree in which the sum of the **weight of the edges is as minimum as possible**.

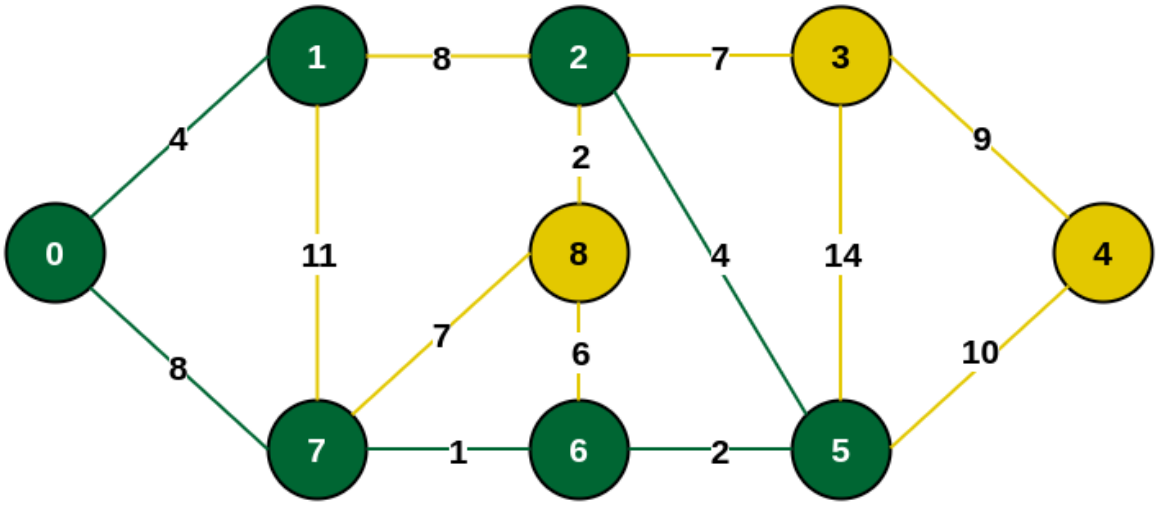


Weighted Graph



A possible spanning tree

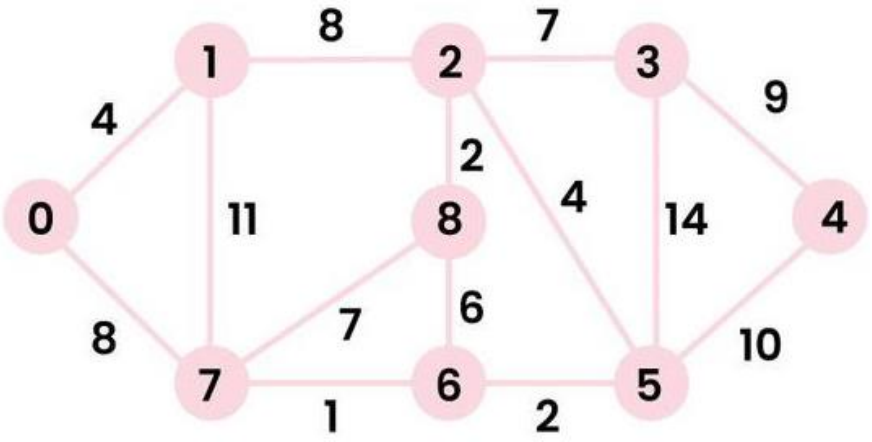
Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex has the minimum sum of weights among all the trees that can be formed from the graph. To apply Prim's algorithm, the given graph must be weighted, connected and undirected.



Minimum weighted edge from MST to other vertices is 5-2 with weight 4

Single source shortest path (Dijkstra's Algorithm)

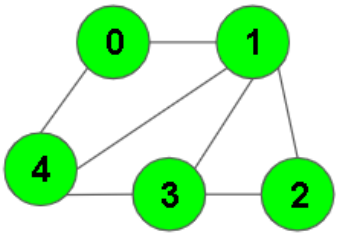
Dijkstra Algorithm is a very famous greedy algorithm which is used for solving the single source shortest path problem. It computes the shortest path from one source node to all other remaining nodes of the graph.



Tasks

1. Given a grid of size n*m (n is the number of rows and m is the number of columns in the grid) consisting of '0's (Water) and '1's(Land). Find the number of islands.
   Note: An island is either surrounded by water or boundary of grid and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.
   Example 1:
   Input:
   grid = {{0,1}, {1,0}, {1,1}, {1,0}}
   Output:
   1
   Explanation:
   The grid is-
   0 1
   1 0
   1 1
   1 0
   All lands are connected.

2. Given an undirected graph with **V** nodes and **E edges**, create and return an adjacency list of the graph. **0-based indexing** is followed everywhere.
   Input: V = 5, E = 7
   edges = {(0,1), (0,4), (4,1), (4,3), (1,3), (1,2), (3,2)}



   Output: {{1,4}, {0,2,3,4}, {1,3}, {1,2,4}, {0,1,3}}
   Explanation:
   Node 0 is connected to 1 and 4.
   Node 1 is connected to 0,2,3 and 4.
   Node 2 is connected to 1 and 3.
   Node 3 is connected to 1,2 and 4.
   Node 4 is connected to 0,1 and 3.

3. Write a program using the **Brute Force Algorithm** to search for overlapping occurrences of a pattern in a string. For example, in the text "ABABABA", the pattern "ABA" overlaps and should be detected at indices 0 and 2.

4. Use both **Boyer-Moore** and **Knuth-Morris-Pratt (KMP)** algorithm to search for the pattern "needle" in the text "findingneedleinneedinahaystack".
   Display:
   - Precomputed tables/rules used by the algorithm.
   - The starting index of each match.

5. Modify the **Knuth-Morris-Pratt Algorithm** to perform a case-insensitive search. For example, in the text "Data Structures", the pattern "data", “DATA”, “Data” all should match.

6. Use the **Rabin-Karp Algorithm** to find all substrings of a given string that are palindromes. For example, in the text "ABCBAB", identify palindromes like "BCB" and "BAB.

7. Given a **connected undirected graph** represented by an adjacency list **adj**, which is a vector of vectors where each adj[i] represents the list of vertices connected to vertex i. Perform a **Depth First Traversal (DFS)** starting from vertex 0, visiting vertices from left to right as per the adjacency list, and return a list containing the DFS traversal of the graph.
   Note: Do traverse in the same order as they are in the adjacency list.
   Examples:
   Input: adj = [[2,3,1], [0], [0,4], [0], [2]]
   Output: [0, 2, 4, 3, 1]
   Explanation: Starting from 0, the DFS traversal as follows:
   Visit 0 → Output: 0
   Visit 2 (the first neighbor of 0) → Output: 0, 2
   Visit 4 (the first neighbor of 2) → Output: 0, 2, 4
   Backtrack to 2, then backtrack to 0, and visit 3 → Output: 0, 2, 4, 3
   Finally, backtrack to 0 and visit 1 → Final Output: 0, 2, 4, 3, 1