DATA STRUCTURE AND ALGORITHMS

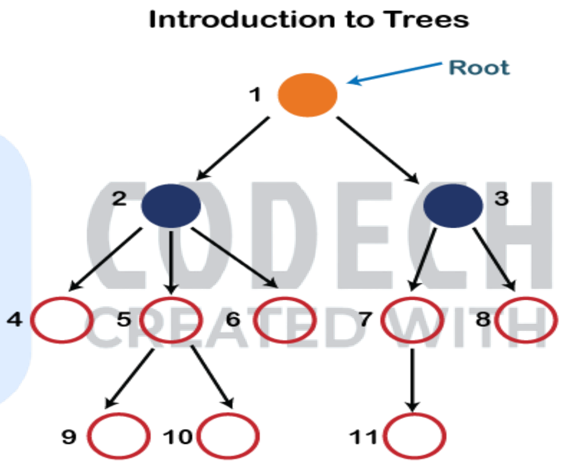
UNIT-4

Trees:

A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially.

Some basic terms used in Tree data structure.

Let's consider the tree structure, which is shown below:



In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.

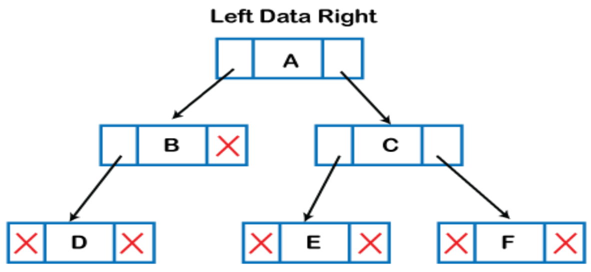
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has at least one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

Implementation of Tree:

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:025

```
struct node

{

  int data;

struct node *left;

struct node *right;

}
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications of trees:

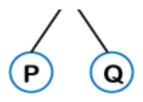
The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure:

1. General tree:

The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.

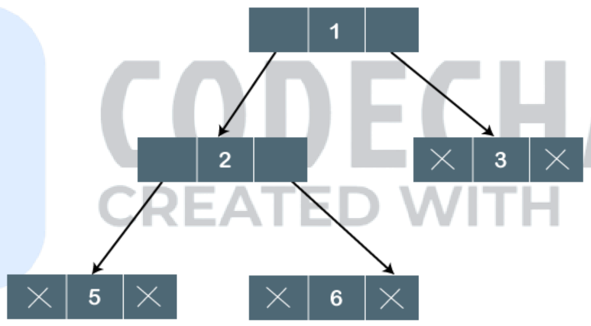


- There can be n number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.
- Every non-empty tree has a downward edge, and these edges are connected to the nodes known as **child nodes**. The root node is labeled with level 0. The nodes that have the same parent are known as **siblings**.

2. Binary tree:

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree:

- At each level of i, the maximum number of nodes is 2i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is (20 + 21 + 22+….2h) = 2h+1 -1.
- The minimum number of nodes possible at height h is equal to h+1.

- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

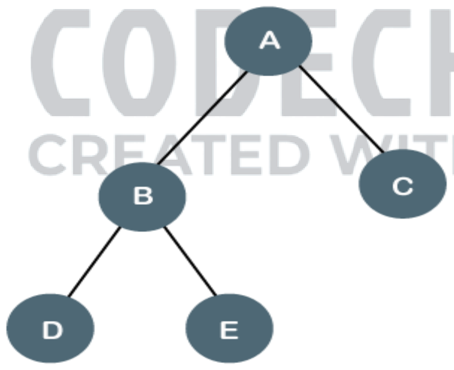
Types of Binary Tree:

1. Full/ proper/ strict Binary tree
2. Complete Binary tree
3. Perfect Binary tree
4. Degenerate Binary tree
5. Balanced Binary tree

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.

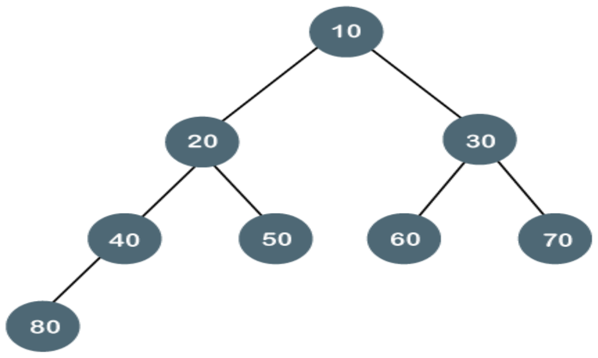


In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

2. Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



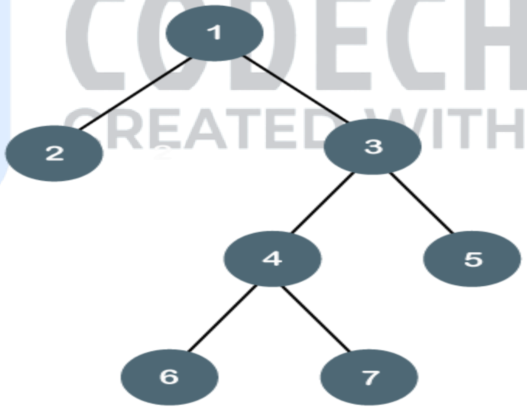
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

3. Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.

Let's look at a simple example of a perfect binary tree.

The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.

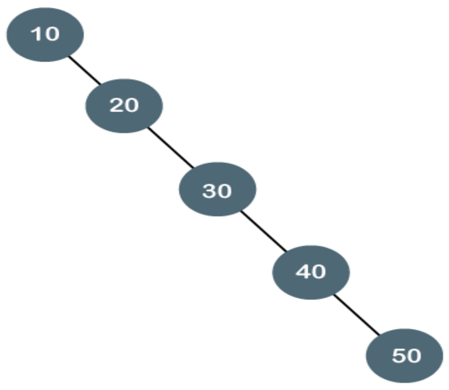


Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

4. Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one child.

Let's understand the Degenerate binary tree through examples.

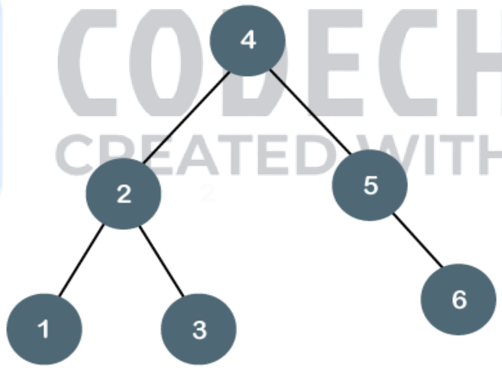


The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

5. Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by at most 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.

Binary Tree Implementation:

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```
struct node

{

  int data,
  struct node *left, *right;
}
```

In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.

Binary tree Traversals:

When all the nodes are created, then it forms a binary tree structure. The process of visiting the nodes is known as tree traversal. There are three types traversals used to visit a node:

- In order traversal
- Preorder traversal
- Post order traversal

Advantages of Binary Tree:

- The searching operation in a binary tree is very fast.
- simple to implement
- easy to understand.
- a hierarchical structure.
- reflect structural relationships that are present in the data set
- easy to insert data than in another data store.
- easy to store data in memory management.
- user can use many nodes
- executions are fast.
- store an arbitrary number of data values.

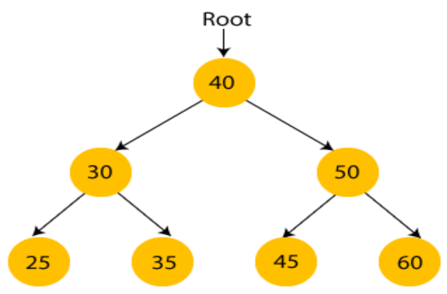
Disadvantages of Binary Tree:

- In binary tree traversals, there are many pointers that are null and hence useless.
- The access operation in a Binary Search Tree (BST) is slower than in an array.
- A basic option is dependent on the height of the tree.
- Deletion node not easy.
- A basic option is based on the height of tree.

3. Binary Search tree

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Searching in Binary search tree:

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

heap sort:

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Heapsort is the in-place sorting algorithm.

Algorithm:

HeapSort(arr)

BuildMaxHeap(arr)

```
for i = length(arr) to 2
    swap arr[1] with arr[i]
        heap_size[arr] = heap_size[arr] ? 1
        MaxHeapify(arr,1)
End
```

Heap sort complexity:

Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1. Time Complexity

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

2. Space Complexity

- The space complexity of Heap sort is O(1).

Graph:

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Important Terms:

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

Graph representation:

In this article, we will discuss the ways to represent the graph. By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

A graph is a data structure that consist a set of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

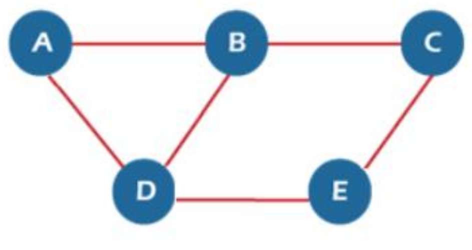
1. Sequential representation (or, Adjacency matrix representation)
2. Linked list representation (or, Adjacency list representation)

1. Sequential representation:

In sequential representation, there is a use of an adjacency matrix to represent the mapping between vertices and edges of the graph. We can use an adjacency matrix to represent the undirected graph, directed graph, weighted directed graph, and weighted undirected graph.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

Now, let's see the adjacency matrix representation of an undirected graph.



Undirected Graph

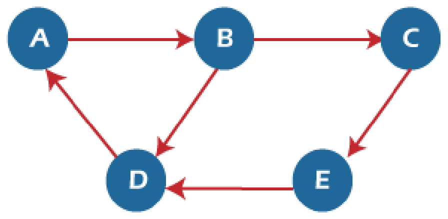
$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

Adjacency Matrix

Adjacency matrix for a directed graph:

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

Consider the below-directed graph and try to construct the adjacency matrix of it.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

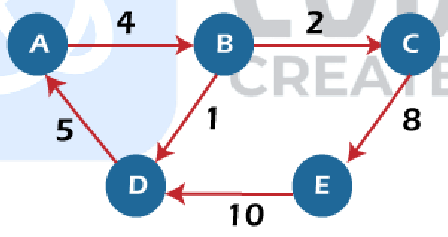
Directed Graph

Adjacency Matrix

In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

Adjacency matrix for a weighted directed graph:

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

weighted Directed Graph

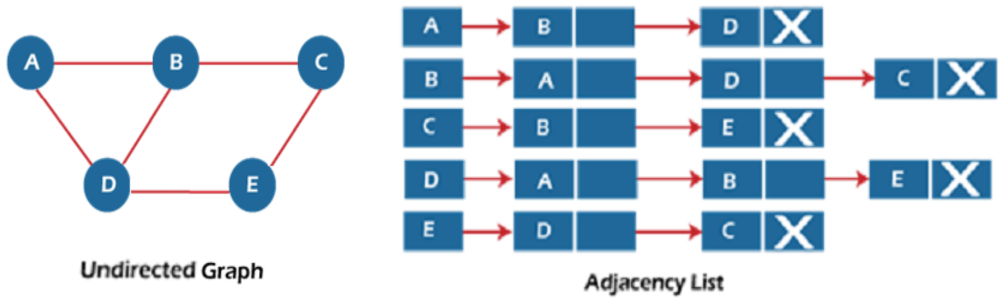
Adjacency Matrix

In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

2. Linked list representation:

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

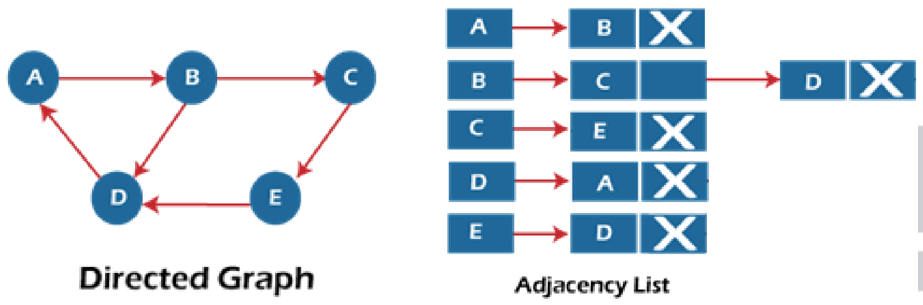
Let's see the adjacency list representation of an undirected graph.



In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

Adjacency list representation of direct graph:

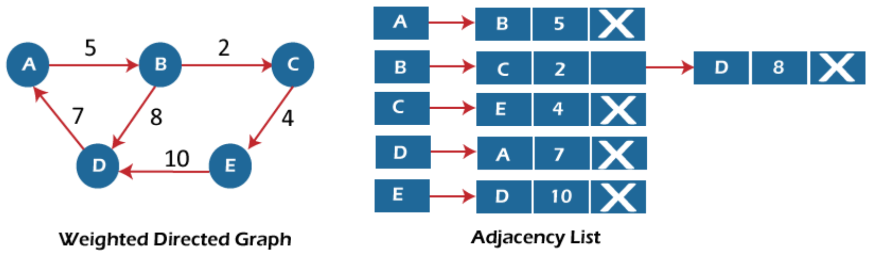
Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Adjacency list representation of weighted directed graph:-

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.

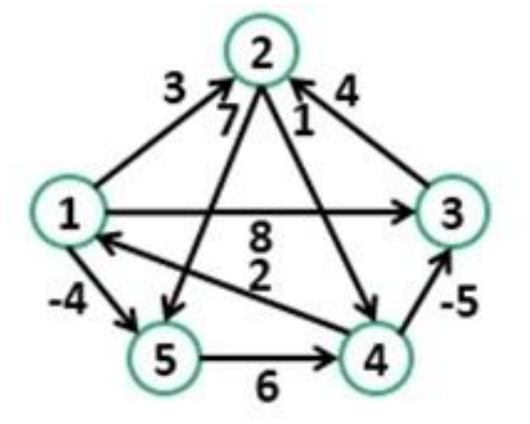


In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Warshall's Algorithm:

Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
- For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
- k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
- k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j]

Graph Traversal:

The graph has two types of traversal algorithms. These are called the Breadth First Search and Depth First Search.

1. Breadth First Search (BFS)

The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

Algorithm:

```
1.  Begin
2.  define an empty queue que at first mark all nodes status as unvisited
3.  add the start vertex into the que while que is not empty, do
4.  delete item from que and set to u
5.  display the vertex u
6.  for all vertices 1 adjacent with u, do
7.  if vertices[i] is unvisited, then  mark vertices[i] as temporarily visited
8.  add v into the queue
9.  mark u as completely visited
10.     Done
11.     End
```

2. Depth First Search (DFS)

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

Algorithm:

```
1.  Begin
2.  initially make the state to unvisited for all nodes
3.  push start into the stack while stack is not empty, do
4.  pop element from stack and set to u
5.  display the node u
6.  if u is not visited, then mark u as visited
7.  for all nodes i connected to u, do
8.  if ith vertex is unvisited, then
9.  push ith vertex into the stack
10. mark ith vertex as visited
11. Done
12. End
```

Operations on Graph:

1. Sub graph:

Getting a sub graph out of a graph is an interesting operation. A sub graph of a graph G (V, E) can be obtained by the following means:

- Removing one or more vertices from the vertex set.
- Removing one or more edges from the edge family.
- Removing either vertices or edges from the graph.

We can extract sub graphs for simple graphs, directed graphs, multi edge graphs and all types of graphs.
The Null Graph is always a sub graph of all the graphs. There can be many sub graphs for a graph.

2. Neighbourhood graph

The neighbourhood graph of a graph G(V,E) only makes sense when we mention it with respect to a given vertex set. For e.g. if V = {1,2,3,4,5} then we can find out the Neighbourhood graph of G(V,E) for vertex set {1}.

So, the neighbourhood graphs contains the vertices 1 and all the edges incident on them and the vertices connected to these edges.

3. Spanning Tree

A spanning tree of a connected graph G(V,E) is a sub graph that is also a tree and connects all vertices in V. For a disconnected graph the spanning tree would be the spanning tree of each component respectively.

There is an interesting set of problems related to finding the minimum spanning tree (which we will be discussing in upcoming posts). There are many algorithms available to solve this problem, for e.g.: Kruskal's, Prim's etc. Note that the concept of minimum spanning tree mostly makes sense in case of weighted graphs. If the graph is not weighted, we consider all the weights to be one and any spanning tree becomes the minimum spanning tree.

4. Reversing a graph

Reversing a graph is an operation meant for directed graphs. To reverse a graph, we reverse the direction of the edges. The reverse graph has the same vertex set as the original graph. As the edges are reversed, the adjacency matrix for this graph is the Transpose of the adjacency matrix for the original graph (Left to the user to draw and check if this holds true).

5. Delete Vertex

We cannot successfully remove a vertex if we do not remove all the edges incident on the vertex. Once we remove the vertex then the adjacency matrix will not contain the row and column for the corresponding vertex.
This operation changes the vertex set and the edge family of the graph.

6. Contract Vertex

One of the important operations on a graph is contraction. It can be done by contracting two vertices into one. Also, it can be done by contracting an edge, which we will see later in this article.

We cannot contract one vertex, for contraction we need a set of vertices, minimum two. Contraction can only be done when there is an edge between the two vertices. The operation basically removes all the edges between the two vertices.

7. Add & Delete Edge

Adding an edge to a graph can be done by connecting two given vertices. Deleting an edge can be done by removing the connection between the given vertices. I am not putting any image as it is very trivial operation.

8. Contracting an Edge

Edge contraction is exactly similar to vertex contraction. It removes the edge from the graph and merges the vertices on which the edge is incident.