**Q1. Define Trees. What is the basic trees terminology?**

**Ans.** Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically. Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



**Important Terms**

Following are the important terms with respect to tree.

* **Path** − Path refers to the sequence of nodes along the edges of a tree.
* **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
* **Parent** − Any node except the root node has one edge upward to a node called parent.
* **Child** − The node below a given node connected by its edge downward is called its child node.
* **Leaf** − The node which does not have any child node is called the leaf node.
* **Subtree** − Subtree represents the descendants of a node.
* **Visiting** − Visiting refers to checking the value of a node when control is on the node.
* **Traversing** − Traversing means passing through nodes in a specific order.
* **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
* **Keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

## BST Basic Operations

The basic operations that can be performed on a binary search tree data structure are the following −

* **Insert** − Inserts an element in a tree/create a tree.
* **Search** − Searches an element in a tree.
* **Preorder Traversal** − Traverses a tree in a pre-order manner.
* **Inorder Traversal** − Traverses a tree in an in-order manner.
* **Postorder Traversal** − Traverses a tree in a post-order manner.

**Q2. Explain Preorder, Inorder and Postorder traversal with example.**

**Q3. Write a short note on Tree Traversal.**

## Ans. Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

* In-order Traversal
* Pre-order Traversal
* Post-order Traversal

**In-order Traversal:**

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains. In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself. If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

***D → B → E → A → F → C → G***

### Algorithm

Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Visit root node.

**Step 3** − Recursively traverse right subtree.

## Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

***A → B → D → E → C → F → G***

### Algorithm

Until all nodes are traversed −

**Step 1** − Visit root node.

**Step 2** − Recursively traverse left subtree.

**Step 3** − Recursively traverse right subtree.

## Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

***D → E → B → F → G → C → A***

### Algorithm

Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Recursively traverse right subtree.

**Step 3** − Visit root node.

**Q4. Explain Binary Search Tree. Design a java program for Binary tree.**

**Ans.** A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

* The left sub-tree of a node has a key less than or equal to its parent node's key.
* The right sub-tree of a node has a key greater than to its parent node's key.

**Representation**

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

**Basic Operations**

Following are the basic operations of a tree −

* **Search** − Searches an element in a tree.
* **Insert** − Inserts an element in a tree.
* **Pre-order Traversal** − Traverses a tree in a pre-order manner.
* **In-order Traversal** − Traverses a tree in an in-order manner.
* **Post-order Traversal** − Traverses a tree in a post-order manner.

## Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree.

## Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

**Q5. Explain and give an example for Depth First Search algorithm.**

**Ans.** Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs the following rules.

* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
* **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
* **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1. | Depth First Search Step One | Initialize the stack. |
| 2. | Depth First Search Step Two | Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. |
| 3. | Depth First Search Step Three | Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only. |
| 4. | Depth First Search Step Four | Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order. |
| 5. | Depth First Search Step Five | We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack. |
| 6. | Depth First Search Step Six | We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack. |
| 7. | Depth First Search Step Seven | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack. |

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

**Q6. Explain and give an example for Depth First Search algorithm.**

**Ans.** Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
* **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
* **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1. | Breadth First Search Step One | Initialize the queue. |
| 2. | Breadth First Search Step Two | We start from visiting **S** (starting node), and mark it as visited. |
| 3. | Breadth First Search Step Three | We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it. |
| 4. | Breadth First Search Step Four | Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it. |
| 5. | Breadth First Search Step Five | Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it. |
| 6. | Breadth First Search Step Six | Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**. |
| 7. | Breadth First Search Step Seven | From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it. |

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

**Q7. Explain graph with its types.**

**Ans.** A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

## Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some 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.



## Basic Operations

Following are basic primary operations of a Graph −

* **Add Vertex** − Adds a vertex to the graph.
* **Add Edge** − Adds an edge between the two vertices of the graph.
* **Display Vertex** − Displays a vertex of the graph.

**Types of Graph:**

**Undirected Graph**

A graph with only undirected edges is said to be undirected graph.

**Directed Graph**

A graph with only directed edges is said to be directed graph.

**Mixed Graph**

A graph with undirected and directed edges is said to be mixed graph.

**End vertices or Endpoints**

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

**Origin**

If an edge is directed, its first endpoint is said to be origin of it.

**Destination**

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

**Adjacent**

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

**Outgoing Edge**

A directed edge is said to be outgoing edge on its orign vertex.

**Incoming Edge**

A directed edge is said to be incoming edge on its destination vertex.

**Degree**

Total number of edges connected to a vertex is said to be degree of that vertex.

**Indegree**

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

**Outdegree**

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

**Parallel edges or Multiple edges**

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

**Simple Graph**

A graph is said to be simple if there are no parallel and self-loop edges.

**Path**

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

**Q8. What is Warshalls algorithm? Explain with example.**

Floyd-Warshall algorithm is a procedure, which is used to find the shorthest (longest) paths among all pairs of nodes in a graph, which does not contain any cycles of negative lenght. The main advantage of Floyd-Warshall algorithm is its simplicity.

The **shortest path** between two nodes of a graph is a sequence of connected nodes so that the sum of the edges that inter-connect them is minimal.

Take this graph,



There are several paths between *A* and *E*:

***Path 1****: A -> B -> E 20****Path 2****: A -> D -> E 25****Path 3****: A -> B -> D -> E 35****Path 4****: A -> D -> B -> E 20*

There are several things to notice here:

1. There can be more then one route between two nodes
2. The number of nodes in the route isn’t important (**Path 4** has *4* nodes but is shorter than **Path 2**, which has *3* nodes)
3. There can be more than one path of minimal length

Something else that should be obvious from the graph is that any path worth considering is [simple](http://en.wikipedia.org/wiki/Path_%28graph_theory%29). That is, you only go through each node **once**.

Unfortunately, this is not always the case. The problem appears when you allow negative weight edges. This isn’t by itself bad. But if **a loop of negative weight appears**, then **there is no shortest path**. Look at this example:



Look at the path *B -> E -> D -> B*. This is a loop, because the starting node is the also the end. What’s the cost? It’s *10 – 20 + 5 = -5*. This means that adding this loop to a path once lowers the cost of the path by *5*. Adding it twice would lower the cost by *2 \* 5 = 10*. So, whatever shortest path you may have come up with, you can make it smaller by going through the loop one more time. BTW there’s no problem with a negative cost path.

**Adjacency Matrices**

There are several different ways to represent a graph in a computer. Although graphs are usually shown diagrammatically, this is only possible when the number of vertices and edges is reasonably small.

Graphs can also be represented in the form of matrices. The major advantage of matrix representation is that the calculation of paths and cycles can easily be performed using well known operations of matrices. However, the disadvantage is that this form of representation takes away from the visual aspect of graphs. It would be difficult to illustrate in a matrix, properties that are easily illustrated graphically.

**Example: Matrix representation of a graph**

Consider the following directed graph G (in which the vertices are ordered as v1, v2, v3, v4, and v5), and its equivalent adjacency matrix representation on the right:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| directed graph |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **v1** | **v2** | **v3** | **v4** | **v5** |
| **v1** | 0 | 1 | 0 | 1 | 1 |
| **v2** | 0 | 0 | 0 | 1 | 0 |
| **v3** | 0 | 0 | 0 | 0 | 1 |
| **v4** | 0 | 0 | 0 | 0 | 0 |
| **v5** | 0 | 1 | 0 | 0 | 0 |

 |

**Definition of an Adjacency Matrix**

An **adjacency matrix** is defined as follows: Let G be a graph with "n" vertices that are assumed to be ordered from v1 to vn.
The n x n matrix A, in which

aij= 1 if there exists a path from vito vj
aij= 0 otherwise

is called an adjacency matrix.