UNIT 5:

Trees and Graphs:

Tree traversal (Inorder, Preorder an Postorder)

In this article, we will discuss the tree traversal in the data structure. The term 'tree traversal' means traversing or visiting each node of a tree. There is a single way to traverse the linear data structure such as linked list, queue, and stack. Whereas, there are multiple ways to traverse a tree that are listed as follows -

- Preorder traversal
- Inorder traversal
- Postorder traversal

So, in this article, we will discuss the above-listed techniques of traversing a tree. Now, let's start discussing the ways of tree traversal.

Preorder traversal

This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

So, in a preorder traversal, each node is visited before both of its subtrees.

The applications of preorder traversal include -

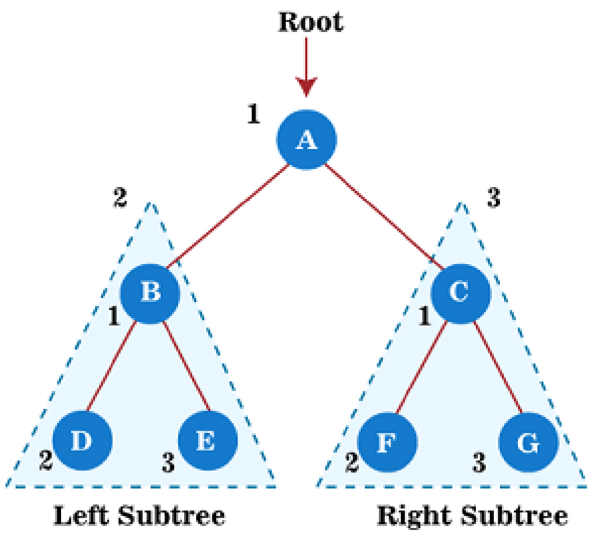
- It is used to create a copy of the tree.
- It can also be used to get the prefix expression of an expression tree.

Algorithm

1. Until all nodes of the tree are not visited
2.
3. Step 1 - Visit the root node
4. Step 2 - Traverse the left subtree recursively.
5. Step 3 - Traverse the right subtree recursively.

Example

Now, let's see the example of the preorder traversal technique.



Now, start applying the preorder traversal on the above tree. First, we traverse the root node **A;** after that, move to its left subtree **B**, which will also be traversed in preorder.

So, for left subtree B, first, the root node **B** is traversed itself; after that, its left subtree **D** is traversed. Since node **D** does not have any children, move to right subtree **E**. As node E also does not have any children, the traversal of the left subtree of root node A is completed.

Now, move towards the right subtree of root node A that is C. So, for right subtree C, first the root node **C** has traversed itself; after that, its left subtree **F** is traversed. Since node **F** does not have any children, move to the right subtree **G**. As node G also does not have any children, traversal of the right subtree of root node A is completed.

Therefore, all the nodes of the tree are traversed. So, the output of the preorder traversal of the above tree is -

A → B → D → E → C → F → G

To know more about the preorder traversal in the data structure, you can follow the link Preorder traversal.

Postorder traversal

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

So, in a postorder traversal, each node is visited after both of its subtrees.

The applications of postorder traversal include -

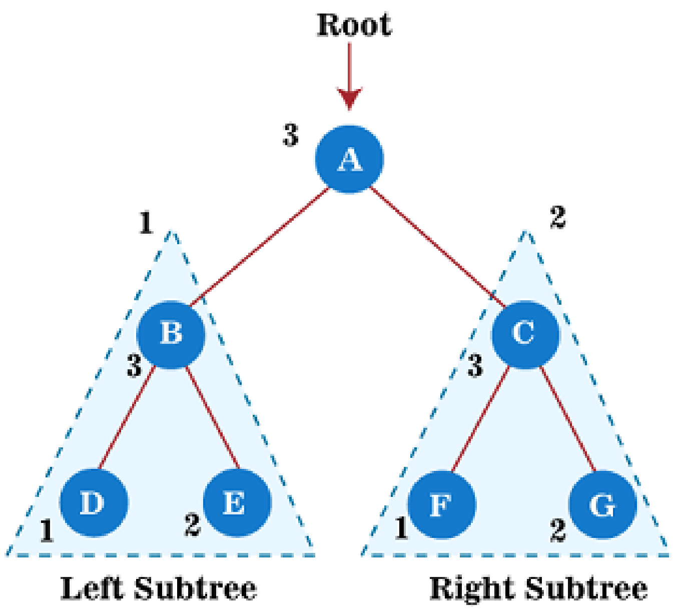
- It is used to delete the tree.
- It can also be used to get the postfix expression of an expression tree.

Algorithm

```
1. Until all nodes of the tree are not visited
2.
3. Step 1 - Traverse the left subtree recursively.
4. Step 2 - Traverse the right subtree recursively.
5. Step 3 - Visit the root node.
```

Example

Now, let's see the example of the postorder traversal technique.



Now, start applying the postorder traversal on the above tree. First, we traverse the left subtree B that will be traversed in postorder. After that, we will traverse the right subtree **C** in postorder. And finally, the root node of the above tree, i.e., **A**, is traversed.

So, for left subtree B, first, its left subtree **D** is traversed. Since node **D** does not have any children, traverse the right subtree **E**. As node E also does not have any children, move to the root node **B.** After traversing node **B,** the traversal of the left subtree of root node A is completed.

Now, move towards the right subtree of root node A that is C. So, for right subtree C, first its left subtree **F** is traversed. Since node **F** does not have any children, traverse the right subtree **G**. As node G also does not have any children, therefore, finally, the root node of the right subtree, i.e., **C,** is traversed. The traversal of the right subtree of root node A is completed.

At last, traverse the root node of a given tree, i.e., **A**. After traversing the root node, the postorder traversal of the given tree is completed.

Therefore, all the nodes of the tree are traversed. So, the output of the postorder traversal of the above tree is -

D → E → B → F → G → C → A

To know more about the postorder traversal in the data structure, you can follow the link Postorder traversal.

Inorder traversal

This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

So, in the inorder traversal, each node is visited in between of its subtrees.

The applications of Inorder traversal includes -

- It is used to get the BST nodes in increasing order.
- It can also be used to get the prefix expression of an expression tree.

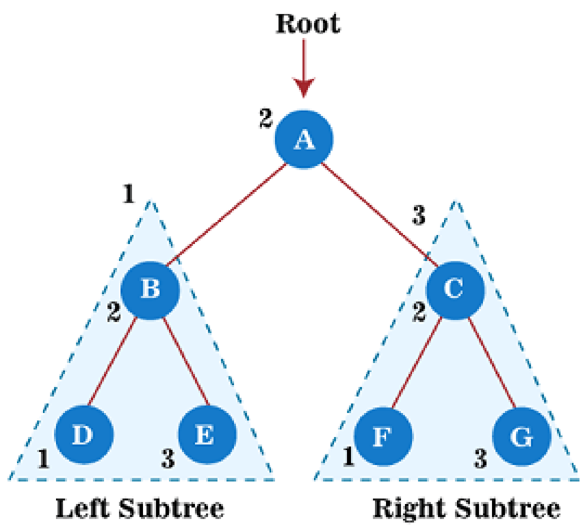
Algorithm

```
1. Until all nodes of the tree are not visited
2.
3. Step 1 - Traverse the left subtree recursively.
```

4. Step 2 - Visit the root node.
5. Step 3 - Traverse the right subtree recursively.

Example

Now, let's see the example of the Inorder traversal technique.



Now, start applying the inorder traversal on the above tree. First, we traverse the left subtree **B** that will be traversed in inorder. After that, we will traverse the root node **A**. And finally, the right subtree **C** is traversed in inorder.

So, for left subtree **B**, first, its left subtree **D** is traversed. Since node **D** does not have any children, so after traversing it, node **B** will be traversed, and at last, right subtree of node B, that is **E**, is traversed. Node E also does not have any children; therefore, the traversal of the left subtree of root node A is completed.

After that, traverse the root node of a given tree, i.e., **A**.

At last, move towards the right subtree of root node A that is C. So, for right subtree C; first, its left subtree **F** is traversed. Since node **F** does not have any children, node **C** will be traversed, and at last, a right subtree of node C, that is, **G**, is traversed. Node G also does not have any children; therefore, the traversal of the right subtree of root node A is completed.

As all the nodes of the tree are traversed, the inorder traversal of the given tree is completed. The output of the inorder traversal of the above tree is -

D → B → E → A → F → C → G

To know more about the inorder traversal in data structure, you can follow the link Inorder Traversal.

Complexity of Tree traversal techniques

The time complexity of tree traversal techniques discussed above is **O(n)**, where **'n'** is the size of binary tree.

Whereas the space complexity of tree traversal techniques discussed above is **O(1)** if we do not consider the stack size for function calls. Otherwise, the space complexity of these techniques is **O(h)**, where **'h'** is the tree's height.

A recursive algorithm

A recursive algorithm simplifies a problem by breaking it down into sub-problems of the same type. The output of one recursion becomes the input for another recursion.

More about Recursive Algorithm

Recursive algorithms get the result for the current input by executing simple operations on the reciprocal value for the simpler or smaller input. They do this by using smaller input values to call themselves.

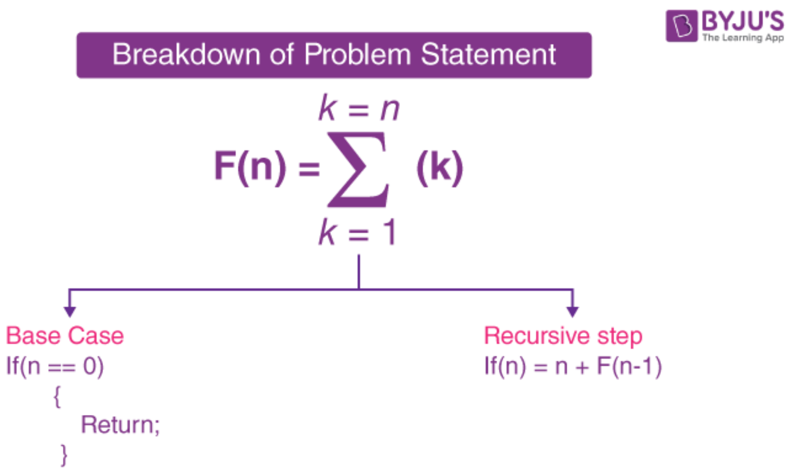
When it comes to develop a recursive algorithm, we need to break the provided problem statement into two parts. The base case is the primary, and the recursive step is the second.

- **Base Case:** The simplest case of a problem is considered as the base case, which consists of a state and condition that ends the recursive function. When a certain condition is met, this base case assesses the result.
- **Recursive Step:** It computes the output by reaching the same function repeatedly but with smaller or more complex inputs.

For example, let’s consider that we have one simplified version of a problem:

$$F(n) = \sum_{k=1}^{k=n} (k)$$

```
Now we will break it down into two parts, as discussed above.
You can see in the given picture the two parts or two steps
representation:
```



```
If(n == 0)
    {
        Return;
    }
```

```
If(n) = n + F(n-1)
```

A non-recursive technique is anything that doesn't use recursion. Insertion sort is a simple example of a non-recursive sorting algorithm. A recursive sorting algorithm calls on itself to sort a smaller part of the array, then combining the partially sorted results. Quick-sort is an example.

A tree is a hierarchical data structure where each node has a maximum of two child nodes (sometimes referred to as successors). The topmost node in the tree is called the root node, and the bottommost nodes are called leaves. leaves. Between the root and leaves are node branches, a tree can be empty, with just a root node, or it can have many levels of nodes. The root node is unique, but other nodes can have multiple parent nodes.

For example, in a family tree, each person has only one biological mother and father, but they may have multiple grandparents, aunts, and uncles, etc. In computer science and software programming, trees are often used to represent the structure of HTML doc, trees are often used to represent the structure of HTML documents or file systems. They can also be used to store data such as

DNA sequences or mathematical expressions. trees are often implemented using pointers in programming languages such as C++.

Types of Trees

After introducing trees in the data structure, we know they are used for different purposes. Here is an overview of some of the most popular types of trees in the data structure.

1. General Tree

A general tree is the most basic type of tree. It is made up of nodes that can have any number of child nodes. There is no specific relationship between the nodes; they can be traversed in any order. General trees are used when the relationship between the nodes is not important.

2. Binary tree

A binary tree is a special type of tree where each and every node can have no more than two child nodes. The left child node is always less than the parent node, and the right child node is always greater than or equal to the parent node. Binary trees are used when the nodes' relationship is important and needs to be kept in order.

3. Binary Search Tree

A binary search tree (BST) is a binary tree where every node has a value greater than all the values in its left subtree and less than all the values in its right subtree. BSTs are used when quickly searching for a value in a large dataset is important.

4. AVL Tree

An AVL tree (named after its inventors Adelson-Velsky and Landis) is a type of BST where each node has a value that is greater than all the values in its left subtree and less than all the values in its right subtree. In addition, an AVL tree must also be balanced, meaning that the difference between the heights of the left subtree and right subtree must be no more than 1. AVL trees are used when quickly searching for a value in a large dataset is important and when maintaining balance is also important.

5. Red-Black Tree

A red-black tree (RBT) is another type of self-balancing BST where each node has an extra bit associated with it that denotes its color (red or black). In addition, certain constraints must be met for an RBT to be valid: 1) every leaf (NULL) node is black, 2) if a node is red, then both its children must be black, 3) every simple path from any given node to any of its descendant leaves contains the same number of black nodes. RBTs are used when quickly searching for a value in a large dataset while also maintaining balance and ensuring constraint satisfaction is important.

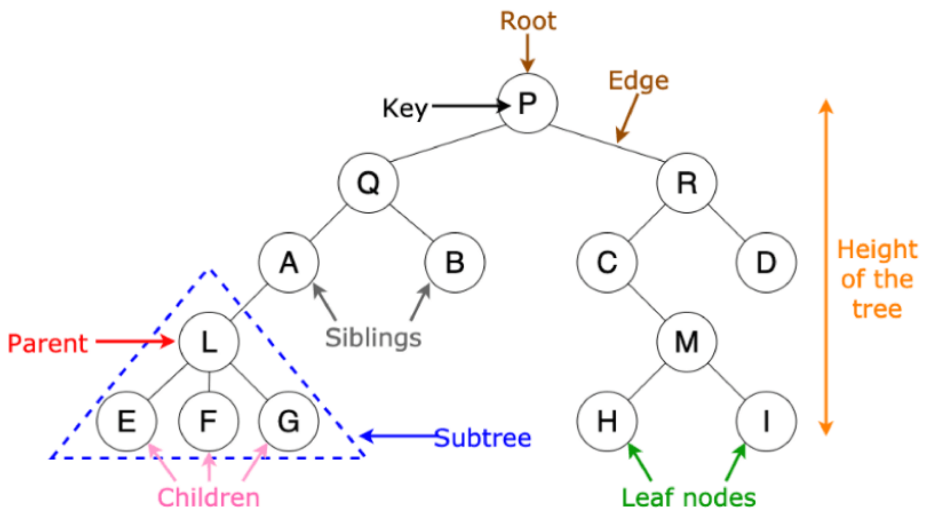
6. N-ary Tree

An n-ary tree (or k-ary tree) generalizes BSTs and RBTs by allowing each node to have no more than k children instead of just 2 children as in BSTs/RBTs. N-ary trees are used when quick search timeshare is important and when the data does not fit well into a traditional binary tree structure(i.e., when k > 2).

Basic Terminologies Used in Tree Data Structure

Understanding basic tree data structure terminologies is crucial for anyone who wants to work with this type of data. Here, we will review some of the most important terms that you need to know.

- **Root:** The root is the topmost node in a tree. It does not have a parent and typically has zero or more child nodes.
- **Child Node**: A child node is any node that has a parent. Child nodes can have their own children (sub-nodes), which makes them parent nodes as well.
- **Parent**: A parent is a node that has at least one child node. Parent nodes can also have their own parents (super-nodes), making them child nodes.



Source

- **Sibling**: Siblings are nodes that share the same parent node. They can be thought of as "brothers and sisters" within the tree structure.
- **Leaf Node**: A leaf node is any node with no child nodes. Leaf nodes are typically the "end" of a tree branch.
- **Internal Nodes**: An internal node is a node that has at least one child node. Internal nodes are typically found "in-between" other nodes in a tree structure.
- **Ancestor Node:** An ancestor node is any node that is on the path from the root to the current node. Ancestor nodes can be thought of as "parents, grandparents, etc."
- **Descendant:** A descendant is a node that is a child, grandchild, great-grandchild, etc., of the current node. In other words, a descendant is any node that is "below" the current node in the tree structure.
- **Height of a Node**: The height of a node is the no. of edges from the node to the deepest leaf descendant. To put it another way, it is the "distance" from the node to the bottom of the tree.
- **Depth of a Node**: The depth of a node is the no. of edges from the root to the node. Therefore, it is the "distance" from the root to the node.
- **Height of a Tree:** The height of a tree is the height of its root node.

To get an insider's view into the advanced terminologies of trees in the data structure, you can look for Python Programming for beginners and experts in online courses. You can get the best out of your knowledge with the most reliable resources around.

Properties of Tree Data Structure

In computer science, a tree is a widely used data structure that simulates a hierarchical tree structure with a set of linked nodes. A tree data structure has the following properties:

1. Recursive data structure
2. Number of edges
3. Depth of node x
4. Height of node x

Read on to learn more about each of these properties of tree data structure in detail!

1. **Recursive Data Structure:** A tree is a recursive data structure because it has a root node, and every node has zero or more child nodes. The root node is the topmost node in the tree, and the child nodes are located below the root node. If each node in the tree has zero or more child nodes, then the tree is said to be an n-ary tree.
2. **Number of Edges:** The number of edges in a tree is always one less than the number of nodes. This is because there is always one fewer edge than there are nodes in any given path from the root to any leaf node.
3. **Depth of Node x:** The depth of a node is defined as the length of the shortest path from the root to that node. In other words, it is simply the number of edges on the path from the root to that particular node.
4. **Height of Node x:** The height of a node is expressed as the length of the longest path from the node to any leaf node. In other words, it is simply the number of edges on the path from that particular node to the deepest leafnode.

By understanding these four properties, you will have a strong foundation on which to build more complex applications using trees!

Applications

In computer science, the tree data structure can be used to store hierarchical data. A tree traversal is a process of visiting each node in a tree. This can be done in different ways like pre-order, post-order, or in-order. Trees are also used to store data that naturally have hierarchical relationships, like the file system on our computers. Besides that, trees are also used in several applications like heaps, tries, and suffix trees. Let's take a look at some of these applications:

1) Storing Naturally Hierarchical Data

One of the main applications of tree data structure is to store hierarchical data. A lot of real-world data falls into this category. For instance, think about the file system on your computer. The files and folders are stored in a hierarchical fashion with a root folder (usually denoted by /). Each subfolder can further have more subfolders and so on. So when you want to store such data, a tree data structure is the most intuitive way to do it.

2) Organize Data

Trees can also be used as an organizational tool. For instance, a family tree is one such example where family relationships are represented using a tree-like structure. Similarly, trees can also be used to represent geographical features like states and cities in the USA or Countries and Continents in the world etc.

3) Trie

Trie is an efficient information retrieval data structure that is based on the key concept of words being prefixes of other words. It's also known as a radix or prefix tree. A Trie has three main properties – keys have consistent length, keys are in lexicographical order, and no key is a prefix of another key at the same level.

With these three conditions met, Trie provides an efficient way to retrieve strings from a dataset with a time complexity of O(M), where M is the length of the string retrieved. This makes it suitable for dictionary operations like autocomplete or spell check etc., which have become very popular these days with internet users all over the world.

4) Heap

A heap is a special type of binary tree where every parent node has either two child nodes or no child nodes, and every node satisfies one heap property – min heap or max heap property software programming. The Min heap property states that every parent node must have a value less than or equal to its child node, while the max heap property specifies that the parent node's value must be greater than or equal to the value of its child nodes (in the case of two children).

So depending upon which property we need to enforce upon our nodes, we call it min heap or max heap accordingly. Since heaps are complete binary trees, they provide a good performance guarantee for insertion and deletion operation with the time complexity of O(logN), where N number of elements currently present inside our heap data structure. Heaps also play an important role behind recommendation algorithms like the "People Also Watched" section on Netflix or Amazon Prime.

5) B-Tree and T-Tree

B-trees and T-trees are two types of trees in the data structure that are used to efficiently store large amounts of data. These trees are often used in databases because they allow for quick insertion and deletion of records while still maintaining fast access times.

6) Routing Table

A routing table maintains information about routes to particular network destinations, possibly via multiple network links/routers, thereby allowing efficient route discovery based on partial match instead of comparison against all known routes leading up to the destination, reducing routing overhead considerably, especially in high volume traffic networks.

Difference Between the Binary Tree and the Binary Search Tree

There are two main types of binary trees: the binary tree and the binary search tree. Both types of trees in data structure have their own unique characteristics and drawbacks.

The biggest difference between the two types of trees in the data structure is in how they are structured. A binary tree comprises two nodes, each of which can have zero, one, or two child nodes. On the other hand, a binary search tree is made up of nodes that each have two child nodes. This difference in structure means that binary trees are more efficient when searching for data, while binary search trees are more efficient when it comes to insertions and deletions.

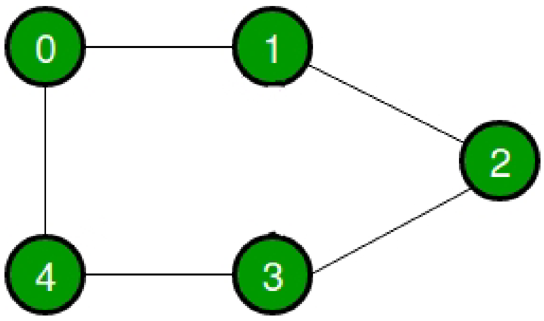
Another difference between the two types of trees in the data structure is how they are traversed. Binary trees can be traversed in either a breadth-first or a depth-first manner, while binary search trees can only be traversed in a depth-first manner. This difference can be significant in performance; the breadth-first traversal of a binary tree is typically faster than the depth-first traversal of a binary search tree.

Overall, the choice of which tree to use depends on the application's specific needs. Both kinds of trees have their own advantages and disadvantages, so it is important to choose the type that best suits the application's needs.

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. In this article, we will understand the difference between the ways of representation of the graph.
A graph can be represented in mainly two ways. They are:

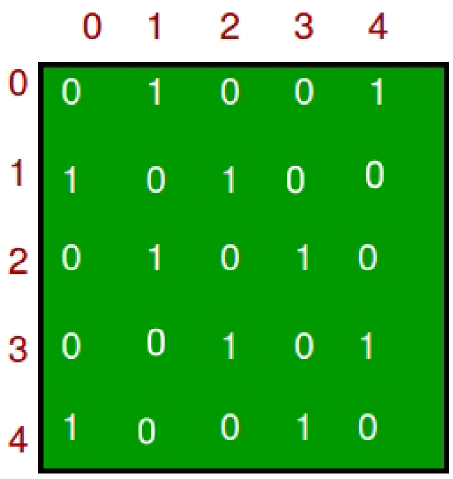
1. **Adjacency List:** An Adjacency list is an array consisting of the address of all the linked lists. The first node of the linked list represents the vertex and the remaining lists connected to this node represents the vertices to which this node is connected. This representation can also be used to represent a weighted graph. The linked list can slightly be changed to even store the weight of the edge.
2. **Adjacency Matrix:** Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Let us consider a graph to understand the adjacency list and adjacency matrix representation. Let the undirected graph be:



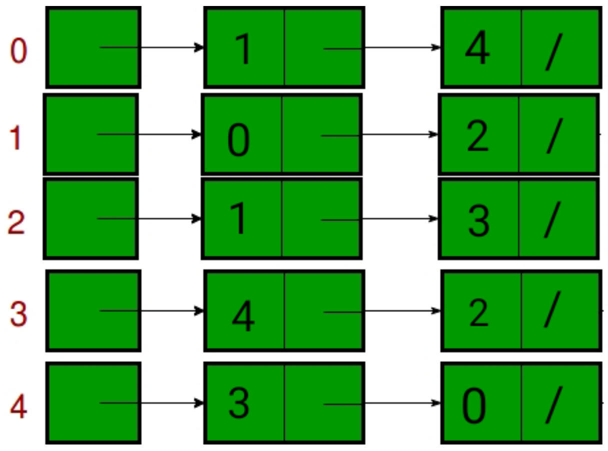
The following graph is represented in the above representations as:

1. **Adjacency Matrix:** In the adjacency matrix representation, a graph is represented in the form of a two-dimensional array. The size of the array is **V x V**, where V is the set of vertices. The following image represents the adjacency matrix representation:



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	0
3	0	0	1	0	1
4	1	0	0	1	0

1. **Adjacency List:** In the adjacency list representation, a graph is represented as an array of linked list. The index of the array represents a vertex and each element in its linked list represents the vertices that form an edge with the vertex. The following image represents the adjacency list representation:



The following table describes the difference between the adjacency matrix and the adjacency list:

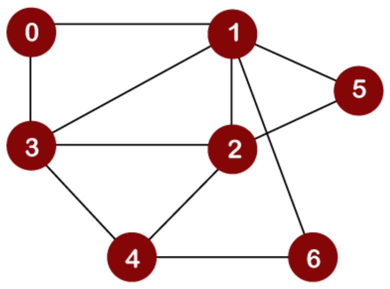
Operations	**Adjacency Matrix**	**Adjacency List**
Storage Space	This representation makes use of VxV matrix, so space required in worst case is **O(|V|^2)**.	In this representation, for every vertex we store its neighbours. In the worst case, if a graph is connected O(V) is required for a vertex and O(E) is required for storing neighbours corresponding to every vertex .Thus, overall space complexity is O(|V|+|E|).
Adding a vertex	In order to add a new vertex to VxV matrix the storage must be increases to (|V|+1)^2. To achieve this we need to copy the whole matrix. Therefore the complexity is **O(|V|^2)**.	There are two pointers in adjacency list first points to the front node and the other one points to the rear node.Thus insertion of a vertex can be done directly in **O(1) time.**
Adding an edge	To add an edge say from i to j, matrix[i][j] = 1 which requires **O(1)** time.	Similar to insertion of vertex here also two pointers are used pointing to the rear and front of the list. Thus, an edge can be inserted in **O(1)**time.
Removing a vertex	In order to remove a vertex from V*V matrix the storage must be decreased to |V|^2 from (|V|+1)^2. To achieve this we need to copy the whole matrix. Therefore the complexity is **O(|V|^2)**.	In order to remove a vertex, we need to search for the vertex which will require O(|V|) time in worst case, after this we need to traverse the edges and in worst case it will require O(|E|) time.Hence, total time complexity is **O(|V|+|E|)**.
Removing an edge	To remove an edge say from i to j, matrix[i][j] = 0 which requires **O(1)** time.	To remove an edge traversing through the edges is required and in worst case we need to traverse through all the edges.Thus, the time complexity is **O(|E|)**.
Querying	In order to find for an existing edge the content of matrix needs to be checked. Given	In an adjacency list every vertex is associated with a list of adjacent vertices. For a given graph, in order

Operations	Adjacency Matrix	Adjacency List
	two vertices say i and j matrix[i][j] can be checked in **O(1)** time.	to check for an edge we need to check for vertices adjacent to given vertex. A vertex can have at most O(|V|) neighbours and in worst can we would have to check for every adjacent vertex. Therefore, time complexity is **O(|V|)** .

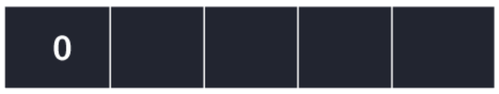
Breadth First Search.

BFS stands for ***Breadth First Search***. It is also known as **level order traversal**. The Queue data structure is used for the Breadth First Search traversal. When we use the BFS algorithm for the traversal in a graph, we can consider any node as a root node.

Let's consider the below graph for the breadth first search traversal.

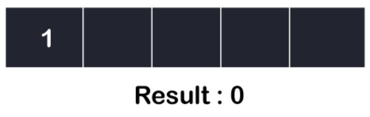


Suppose we consider node 0 as a root node. Therefore, the traversing would be started from node 0.



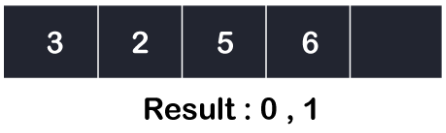
Once node 0 is removed from the Queue, it gets printed and marked as a ***visited node.***

Once node 0 gets removed from the Queue, then the adjacent nodes of node 0 would be inserted in a Queue as shown below:

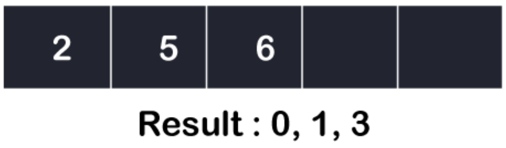


Now the node 1 will be removed from the Queue; it gets printed and marked as a visited node

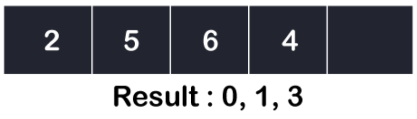
Once node 1 gets removed from the Queue, then all the adjacent nodes of a node 1 will be added in a Queue. The adjacent nodes of node 1 are 0, 3, 2, 6, and 5. But we have to insert only unvisited nodes in a Queue. Since nodes 3, 2, 6, and 5 are unvisited; therefore, these nodes will be added in a Queue as shown below:



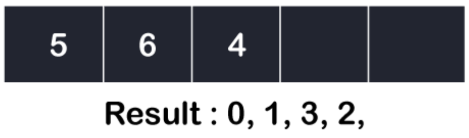
The next node is 3 in a Queue. So, node 3 will be removed from the Queue, it gets printed and marked as visited as shown below:



Once node 3 gets removed from the Queue, then all the adjacent nodes of node 3 except the visited nodes will be added in a Queue. The adjacent nodes of node 3 are 0, 1, 2, and 4. Since nodes 0, 1 are already visited, and node 2 is present in a Queue; therefore, we need to insert only node 4 in a Queue.

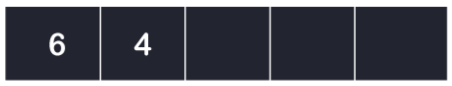


Now, the next node in the Queue is 2. So, 2 would be deleted from the Queue. It gets printed and marked as visited as shown below:



Once node 2 gets removed from the Queue, then all the adjacent nodes of node 2 except the visited nodes will be added in a Queue. The adjacent nodes of node 2 are 1, 3, 5, 6, and 4. Since the nodes 1 and 3 have already been visited, and 4, 5, 6 are already added in the Queue; therefore, we do not need to insert any node in the Queue.

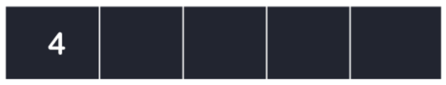
The next element is 5. So, 5 would be deleted from the Queue. It gets printed and marked as visited as shown below:



Result : 0, 1, 3, 2, 5

Once node 5 gets removed from the Queue, then all the adjacent nodes of node 5 except the visited nodes will be added in the Queue. The adjacent nodes of node 5 are 1 and 2. Since both the nodes have already been visited; therefore, there is no vertex to be inserted in a Queue.

The next node is 6. So, 6 would be deleted from the Queue. It gets printed and marked as visited as shown below:



Result : 0, 1, 3, 2, 5, 6

Once the node 6 gets removed from the Queue, then all the adjacent nodes of node 6 except the visited nodes will be added in the Queue. The adjacent nodes of node 6 are 1 and 4. Since the node 1 has already been visited and node 4 is already added in the Queue; therefore, there is not vertex to be inserted in the Queue.

The next element in the Queue is 4. So, 4 would be deleted from the Queue. It gets printed and marked as visited.

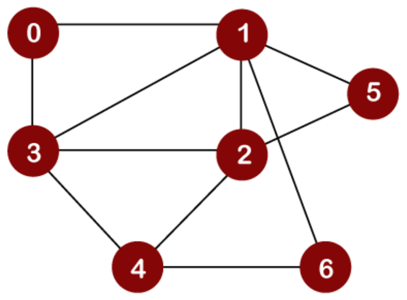
Once the node 4 gets removed from the Queue, then all the adjacent nodes of node 4 except the visited nodes will be added in the Queue. The adjacent nodes of node 4 are 3, 2, and 6. Since all the adjacent nodes have already been visited; so, there is no vertex to be inserted in the Queue.

What is DFS?

DFS stands for Depth First Search. In DFS traversal, the stack data structure is used, which works on the LIFO (Last In First Out) principle. In DFS, traversing can be started from any node, or we can say that any node can be considered as a root node until the root node is not mentioned in the problem.

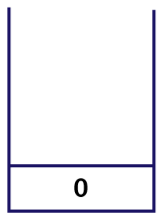
In the case of BFS, the element which is deleted from the Queue, the adjacent nodes of the deleted node are added to the Queue. In contrast, in DFS, the element which is removed from the stack, then only one adjacent node of a deleted node is added in the stack.

Let's consider the below graph for the Depth First Search traversal.

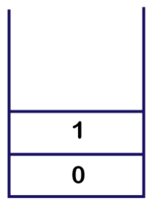


Consider node 0 as a root node.

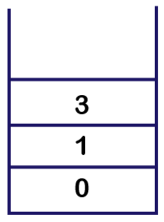
First, we insert the element 0 in the stack as shown below:



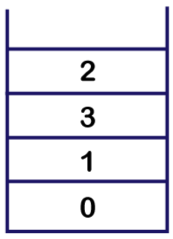
The node 0 has two adjacent nodes, i.e., 1 and 3. Now we can take only one adjacent node, either 1 or 3, for traversing. Suppose we consider node 1; therefore, 1 is inserted in a stack and gets printed as shown below:



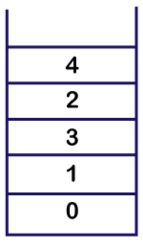
Now we will look at the adjacent vertices of node 1. The unvisited adjacent vertices of node 1 are 3, 2, 5 and 6. We can consider any of these four vertices. Suppose we take node 3 and insert it in the stack as shown below:



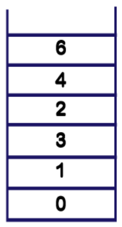
Consider the unvisited adjacent vertices of node 3. The unvisited adjacent vertices of node 3 are 2 and 4. We can take either of the vertices, i.e., 2 or 4. Suppose we take vertex 2 and insert it in the stack as shown below:



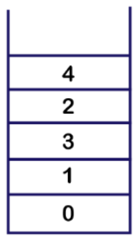
The unvisited adjacent vertices of node 2 are 5 and 4. We can choose either of the vertices, i.e., 5 or 4. Suppose we take vertex 4 and insert in the stack as shown below:

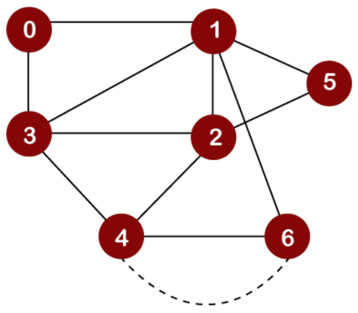


Now we will consider the unvisited adjacent vertices of node 4. The unvisited adjacent vertex of node 4 is node 6. Therefore, element 6 is inserted into the stack as shown below:

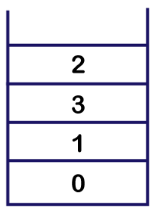


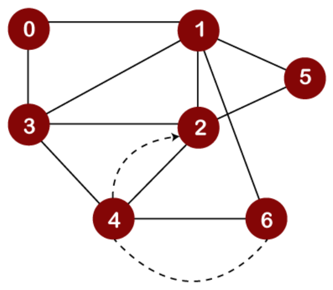
After inserting element 6 in the stack, we will look at the unvisited adjacent vertices of node 6. As there is no unvisited adjacent vertices of node 6, so we cannot move beyond node 6. In this case, we will perform **backtracking**. The topmost element, i.e., 6 would be popped out from the stack as shown below:



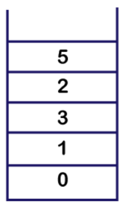


The topmost element in the stack is 4. Since there are no unvisited adjacent vertices left of node 4; therefore, node 4 is popped out from the stack as shown below:

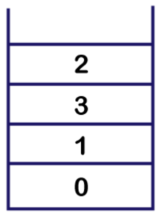




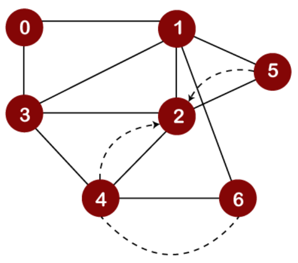
The next topmost element in the stack is 2. Now, we will look at the unvisited adjacent vertices of node 2. Since only one unvisited node, i.e., 5 is left, so node 5 would be pushed into the stack above 2 and gets printed as shown below:



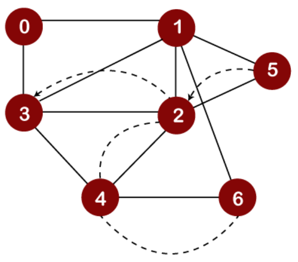
Now we will check the adjacent vertices of node 5, which are still unvisited. Since there is no vertex left to be visited, so we pop the element 5 from the stack as shown below:



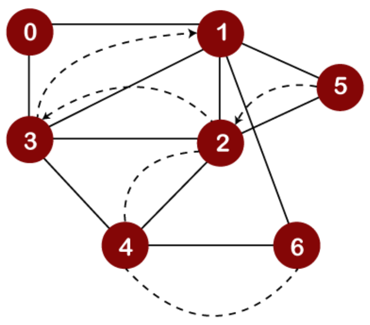
We cannot move further 5, so we need to perform backtracking. In backtracking, the topmost element would be popped out from the stack. The topmost element is 5 that would be popped out from the stack, and we move back to node 2 as shown below:



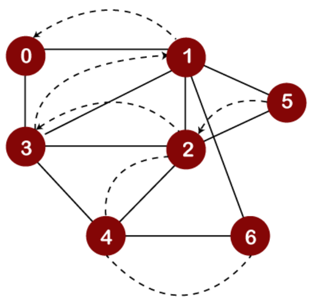
Now we will check the unvisited adjacent vertices of node 2. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 2 would be popped out from the stack, and we move back to the node 3 as shown below:



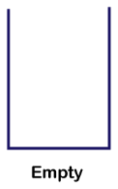
Now we will check the unvisited adjacent vertices of node 3. As there is no adjacent vertex left to be visited, so we perform backtracking. In backtracking, the topmost element, i.e., 3 would be popped out from the stack and we move back to node 1 as shown below:



After popping out element 3, we will check the unvisited adjacent vertices of node 1. Since there is no vertex left to be visited; therefore, the backtracking will be performed. In backtracking, the topmost element, i.e., 1 would be popped out from the stack, and we move back to node 0 as shown below:



We will check the adjacent vertices of node 0, which are still unvisited. As there is no adjacent vertex left to be visited, so we perform backtracking. In this, only one element, i.e., 0 left in the stack, would be popped out from the stack as shown below:



As we can observe in the above figure that the stack is empty. So, we have to stop the DFS traversal here, and the elements which are printed is the result of the DFS traversal.

Differences between BFS and DFS

The following are the differences between the BFS and DFS:

BFS	DFS

Full form	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Technique	It a vertex-based technique to find the shortest path in a graph.	It is an edge-based technique because the vertices along the edge are explored first from the starting to the end node.
Definition	BFS is a traversal technique in which all the nodes of the same level are explored first, and then we move to the next level.	DFS is also a traversal technique in which traversal is started from the root node and explore the nodes as far as possible until we reach the node that has no unvisited adjacent nodes.
Data Structure	Queue data structure is used for the BFS traversal.	Stack data structure is used for the BFS traversal.
Backtracking	BFS does not use the backtracking concept.	DFS uses backtracking to traverse all the unvisited nodes.
Number of edges	BFS finds the shortest path having a minimum number of edges to traverse from the source to the destination vertex.	In DFS, a greater number of edges are required to traverse from the source vertex to the destination vertex.
Optimality	BFS traversal is optimal for those vertices which are to be searched closer to the source vertex.	DFS traversal is optimal for those graphs in which solutions are away from the source vertex.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Suitability for decision tree	It is not suitable for the decision tree because it requires exploring all the neighboring nodes first.	It is suitable for the decision tree. Based on the decision, it explores all the paths. When the goal is found, it stops its traversal.
Memory efficient	It is not memory efficient as it requires more memory than DFS.	It is memory efficient as it requires less memory than BFS.

A spanning tree

A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

A spanning tree consists of (n-1) edges, where 'n' is the number of vertices (or nodes). Edges of the spanning tree may or may not have weights assigned to them. All the possible spanning trees created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.

A complete undirected graph can have $\mathbf{n^{n-2}}$ number of spanning trees where **n** is the number of vertices in the graph. Suppose, if **n = 5**, the number of maximum possible spanning trees would be $\mathbf{5^{5-2} = 125.}$

Applications of the spanning tree

Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows -

- Cluster Analysis
- Civil network planning
- Computer network routing protocol

Kruskal's algorithm for MST

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Below are the steps for finding MST using Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning-tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Prim's algorithm for MST

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Below are the steps for finding MST using Prim's algorithm

1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3. While mstSet doesn't include all vertices
    - Pick a vertex u which is not there in mstSet and has minimum key value.
    - Include u to mstSet.
    - Update the key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u-v is less than the previous key value of v, update the key value as the weight of u-v

Both **Prim's and Kruskal's algorithm** finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few **major differences between them**.

Prim's Algorithm	**Kruskal's Algorithm**
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of O(V^2), V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps.	Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components

Prim’s Algorithm	**Kruskal’s Algorithm**
Prim’s algorithm runs faster in dense graphs.	Kruskal’s algorithm runs faster in sparse graphs.
It generates the minimum spanning tree starting from the root vertex.	It generates the minimum spanning tree starting from the least weighted edge.
Applications of prim’s algorithm are Travelling Salesman Problem, Network for roads and Rail tracks connecting all the cities etc.	Applications of Kruskal algorithm are LAN connection, TV Network etc.
Prim’s algorithm prefer list data structures.	Kruskal’s algorithm prefer heap data structures.