



Parshvanath Charitable Trust's  
**A. P. SHAH INSTITUTE OF TECHNOLOGY**  
(Approved by AICTE New Delhi & Govt. of Maharashtra, Affiliated to University of Mumbai)  
(Religious Jain Minority)

---

## Department of Information Technology

# Chapter-6 Trees and Graph

Introduction to Trees, Definitions & Tree terminologies, Binary tree representation, Operations on binary tree, Traversal of binary trees, Binary search tree, Threaded Binary tree, Expression tree, Application of Trees

Introduction to Graph, Introduction Graph Terminologies, Graph Representation, Type of graphs, Graph traversal: Depth first search (DFS) & Breadth First search (BFS), Minimum Spanning Tree : Prim's & Kruskal's Shortest Path Algorithm – Dijkstra's Algorithm. Applications of graph

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

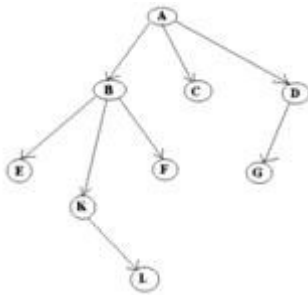
## Unit 6

### Trees and Graphs

#### Trees

A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a *nonlinear* data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more subtrees. A tree has following general properties:

- One node is distinguished as a root;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node; A direction is: *parent* -> *children*



A is a parent of B, C, D,

B is called a child of A.

on the other hand, B is a parent of E, F, K

In the above picture, the root has 3 subtrees.

#### Definitions and Tree Terminologies

Following are the important terms with respect to tree.

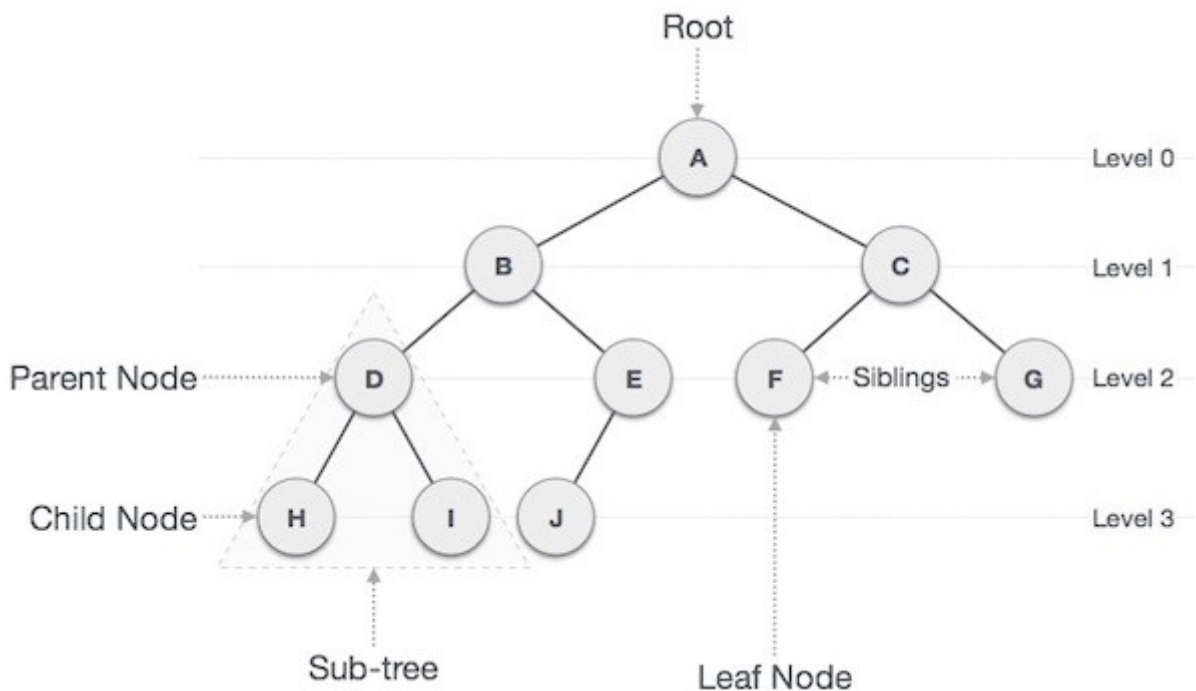
- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – Any node except the root node has one edge upward to a node called parent.
- **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.

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

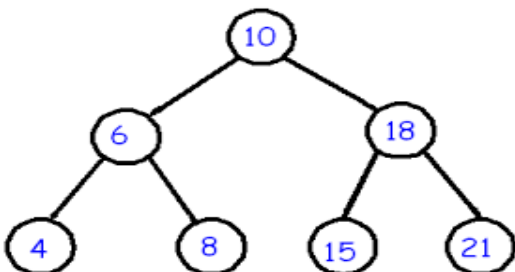
Academic Year: \_\_\_\_\_

- **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.



## Types of Trees

### 1.Binary Trees:



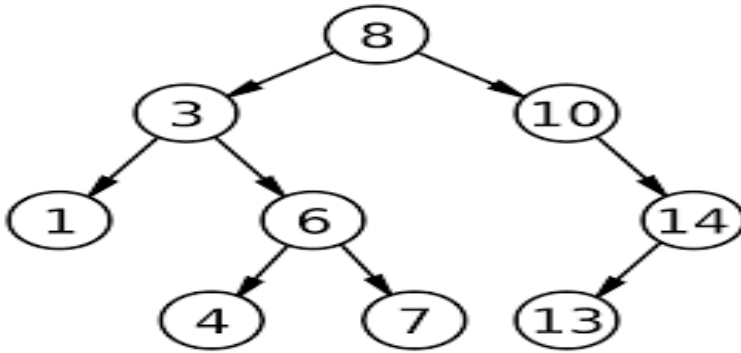
A **binary tree** is a **tree data structure** in which each node has at most two children, which are referred to as the left child and the right child.

### 2.Binary Search Tree:

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

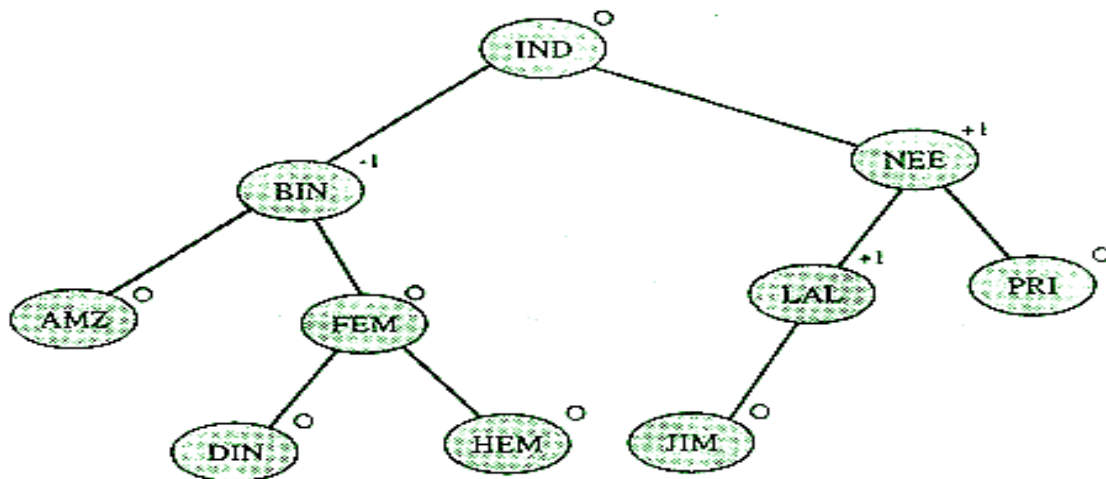


Binary Search Tree follow (Left.Value<Root<Rightchild.value) Rule

\*The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient; they are also easy to code.

\*Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

### 3.AVL TREE:



A **AVL tree** is a self-balancing binary search **tree**. In an **AVL tree**, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.

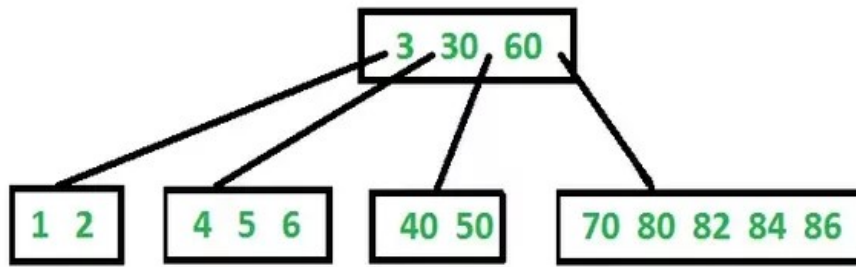
Operations Like Insertion and deletion have low complexity.

### 4.B-Tree:

Semester: \_\_\_\_\_

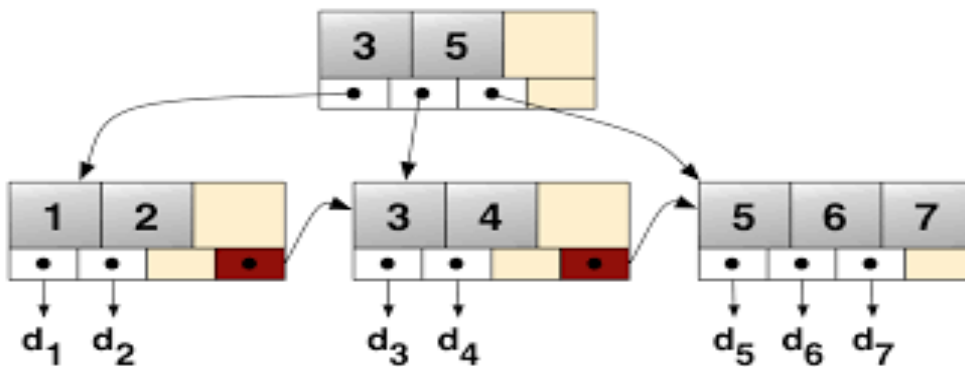
Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_



A **B-tree** is a self-balancing **tree data structure** that keeps **data** sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The **B-tree** is a generalization of a binary search **tree** in that a node can have more than two children.

5.B+ Tree:



A **B+ tree** is an n-array tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.

A B+ tree can be viewed as a [B-tree](#) in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

### Binary Tree Representation

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

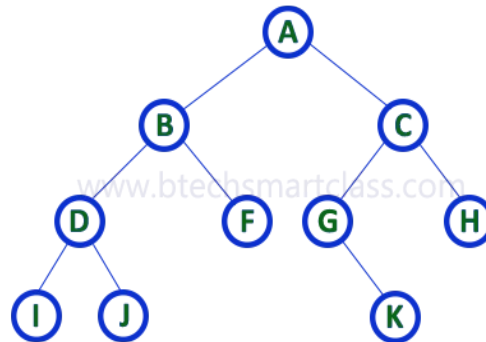


Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

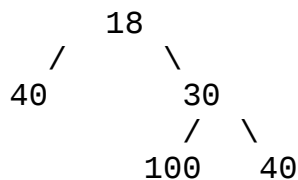
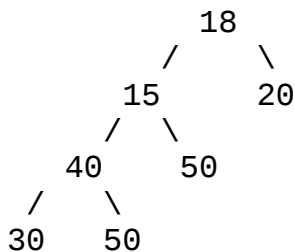
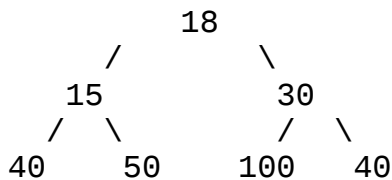
Academic Year: \_\_\_\_\_

Example



There are different types of binary trees and they are...

**Full Binary Tree** A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree.



***In a Full Binary, number of leaf nodes is number of internal nodes plus 1***

$$L = I + 1$$

Where L = Number of leaf nodes, I = Number of internal nodes

See [Handshaking Lemma and Tree](#) for proof.

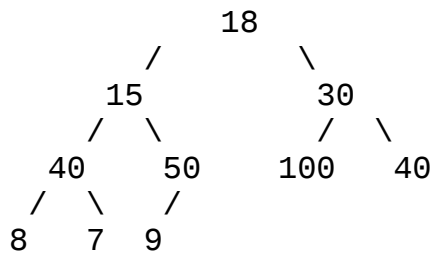
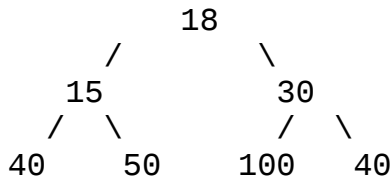
Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

**Complete Binary Tree:** A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

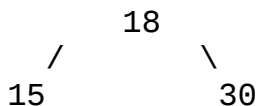
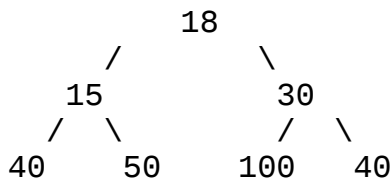
Following are examples of Complete Binary Trees



Practical example of Complete Binary Tree is [Binary Heap](#).

**Perfect Binary Tree** A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

Following are examples of Perfect Binaryr Trees.



A Perfect Binary Tree of height  $h$  (where height is number of nodes on path from root to leaf) has  $2^h - 1$  node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

### Balanced Binary Tree

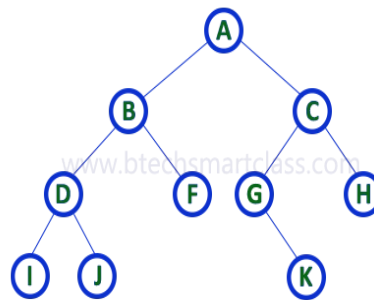
A binary tree is balanced if height of the tree is  $O(\log n)$  where  $n$  is number of nodes. For Example, AVL tree maintain  $O(\log n)$  height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain  $O(\log n)$  height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide  $O(\log n)$  time for search, insert and delete.

### Binary Tree Representation

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation
2. Linked List Representation

Consider the following binary tree...



#### 1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

Consider the above example of binary tree and it is represented as follows...



To represent a binary tree of depth ' $n$ ' using array representation, we need one dimensional array with a maximum size of  $2^{n+1} - 1$ .



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

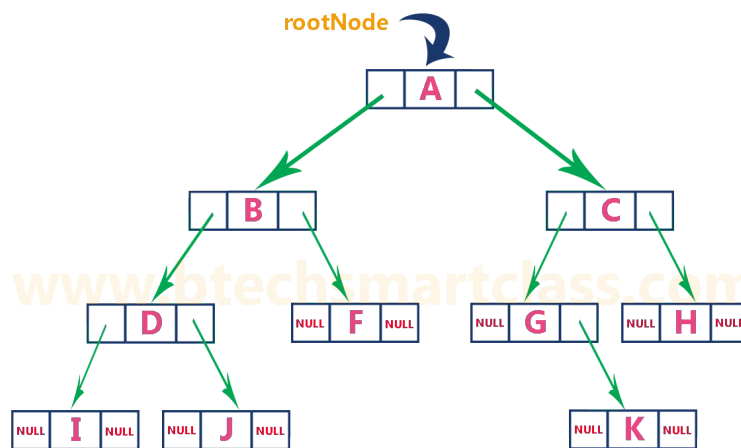
## 2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...



The above example of binary tree represented using Linked list representation is shown as follows...



### Binary Tree Operations

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Binary Tree Representation in C: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.

A Tree node contains following parts.

1. Data
2. Pointer to left child
3. Pointer to right child

In C, we can represent a tree node using structures. Below is an example of a tree node with an integer data.

```
struct node
{
```



Semester: \_\_\_\_\_

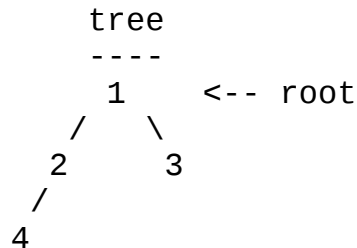
Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

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

First Simple Tree in C

Let us create a simple tree with 4 nodes in C. The created tree would be as following.



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

```
/* newNode() allocates a new node with the given data and NULL  
left and  
right pointers. */
```

```
struct node* newNode(int data)  
{  
    // Allocate memory for new node  
    struct node* node = (struct node*)malloc(sizeof(struct node));  
  
    // Assign data to this node  
    node->data = data;  
  
    // Initialize left and right children as NULL  
    node->left = NULL;  
    node->right = NULL;  
    return(node);  
}
```

```
int main()  
{  
    /*create root*/  
    struct node *root = newNode(1);
```



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

/\* following is the tree after above statement

```
      1
     / \
    NULL NULL
*/
```

```
root->left      = newNode(2);
root->right      = newNode(3);
/* 2 and 3 become left and right children of 1
```

```
      1
     / \
    2   3
   / \ / \
  NULL NULL NULL NULL
*/
```

```
root->left->left = newNode(4);
/* 4 becomes left child of 2
```

```
      1
     / \
    2   3
   / \ / \
  4  NULL NULL NULL
 / \
NULL NULL
*/
```

```
getchar();
return 0;
}
```

## Binary Tree Traversal

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

### 3. Post - Order Traversal

#### 1. In - Order Traversal ( leftChild - root - rightChild )

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

#### 2. Pre - Order Traversal ( root - leftChild - rightChild )

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

## 2. Post - Order Traversal ( leftChild - rightChild - root )

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Inorder: visit left subtree, visit root, visit right subtree	Preorder: visit root, visit left, visit right	Postorder: visit left, visit right, visit root
<pre>private void Inorder(BSTNode root) {     if(root != null) {         Inorder(root.left);         Process(root.value);         Inorder(root.right);     } }  public void Inorder(){     Inorder(root); }</pre>	<pre>private void Preorder(BSTNode root) {     if(root != null) {         Process(root.value);         Preorder(root.left);         Preorder(root.right);     } }  public void Preorder(){     Preorder(root); }</pre>	<pre>private void Postorder(BSTNode root) {     if(root != null) {         Postorder(root.left);         Postorder(root.right);         Process(root.value);     } }  public void Postorder(){     Postorder(root); }</pre>

## Binary Search Tree

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.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

left\_subtree (keys) ≤ node (key) ≤ right\_subtree (keys)

Semester: \_\_\_\_\_

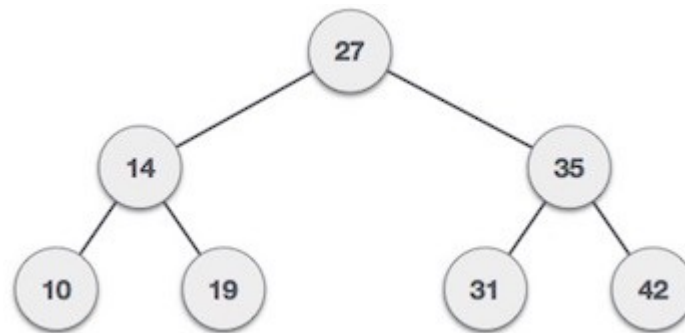
Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

### 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.

### Node

Define a node having some data, references to its left and right child nodes.

```
struct node {  
    int data;  
    struct node *leftChild;
```



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

```
    struct node *rightChild;  
};
```

### 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. Follow the same algorithm for each node.

### Algorithm

```
struct node* search(int data){  
    struct node *current = root;  
    printf("Visiting elements: ");  
  
    while(current->data != data){  
  
        if(current != NULL) {  
            printf("%d ", current->data);  
  
            //go to left tree  
            if(current->data > data){  
                current = current->leftChild;  
            }//else go to right tree  
            else {  
                current = current->rightChild;  
            }  
  
            //not found  
            if(current == NULL){  
                return NULL;  
            }  
        }  
    }  
    return current;  
}
```

### 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.

### Algorithm

```
void insert(int data) {  
    struct node *tempNode = (struct node*) malloc(sizeof(struct  
node));
```



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

```
struct node *current;
struct node *parent;

tempNode->data = data;
tempNode->leftChild = NULL;
tempNode->rightChild = NULL;

//if tree is empty
if(root == NULL) {
    root = tempNode;
} else {
    current = root;
    parent = NULL;

    while(1) {
        parent = current;

        //go to left of the tree
        if(data < parent->data) {
            current = current->leftChild;
            //insert to the left

            if(current == NULL) {
                parent->leftChild = tempNode;
                return;
            }
        } //go to right of the tree
        else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
                parent->rightChild = tempNode;
                return;
            }
        }
    }
}
```

### Threaded Binary Tree

Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

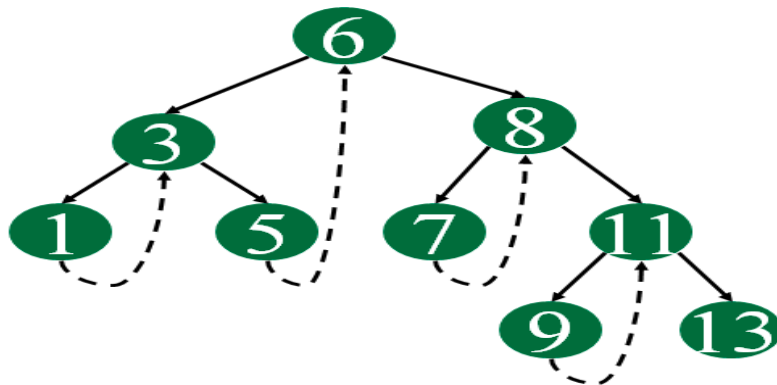
There are two types of threaded binary trees.

**Single Threaded:** Where a NULL right pointers is made to point to the inorder successor (if successor exists)

**Double Threaded:** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



### C representation of a Threaded Node

Following is C representation of a single threaded node.

```
struct Node
{
    int data;
    Node *left, *right;
    bool rightThread;
}
```

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

### Inorder Taversal using Threads



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

Following is C code for inorder traversal in a threaded binary tree.

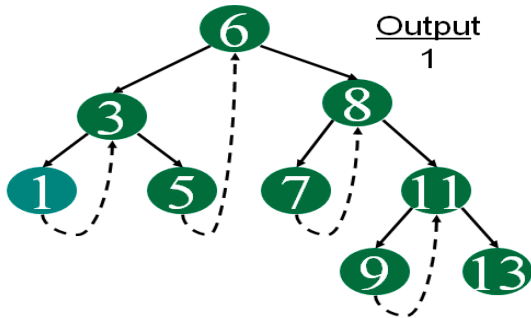
```
// Utility function to find leftmost node in a tree rooted with n
struct Node* leftMost(struct Node *n)
{
```

Following diagram demonstrates inorder order traversal using threads.

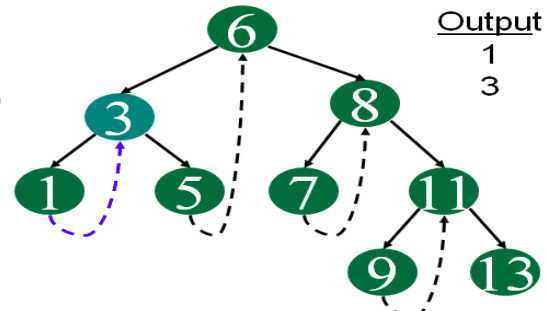
Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

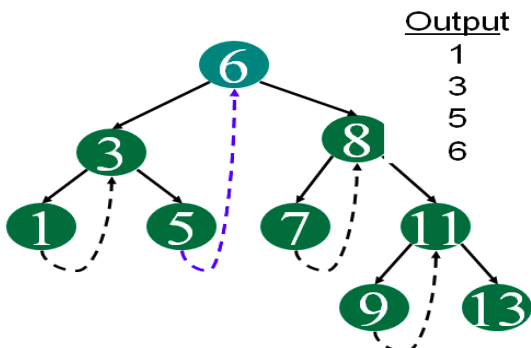
Academic Year: \_\_\_\_\_



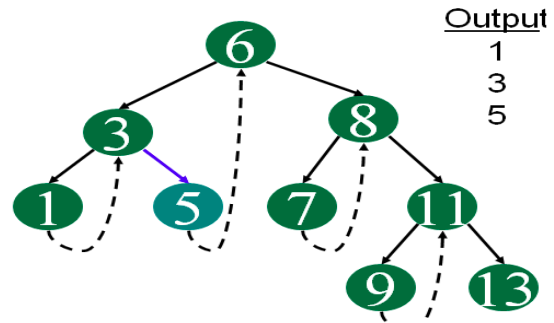
Start at leftmost node, print it



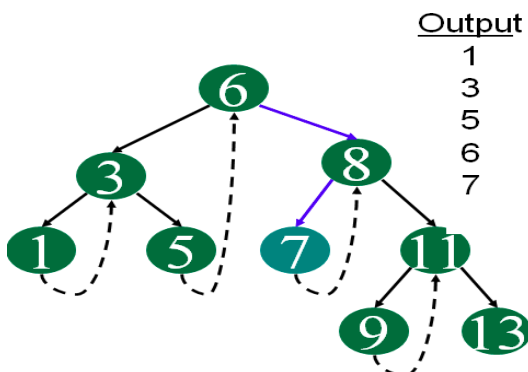
Follow thread to right, print node



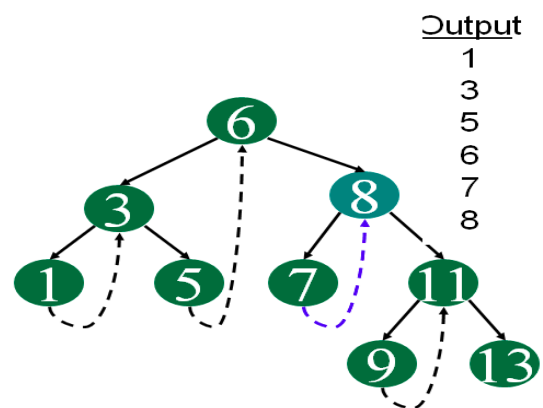
Follow thread to right, print node



Follow link to right, go to leftmost node and print



Follow link to right, go to leftmost node and print



Follow thread to right, print node

**continue same way for remaining node.....**

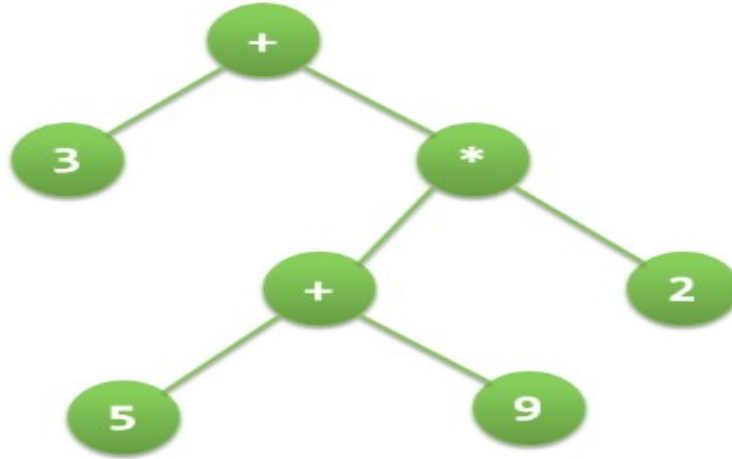
Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

### Expression Tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for  $3 + ((5+9)*2)$  would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

### **Evaluating the expression represented by expression tree:**

```
Let t be the expression tree
If t is not null then
    If t.value is operand then
        Return t.value
    A = solve(t.left)
    B = solve(t.right)

    // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

### **Construction of Expression Tree:**

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

- 1) If character is operand push that into stack
- 2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

### //Implementation of Expression Tree

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#include<conio.h>
```

```
struct tree
```

```
{
```

```
char data;
```

```
struct tree *left,*right;
```

```
};
```

```
int top=-1;
```

```
struct tree *stack[20];
```

```
struct tree *node;
```

```
void push(struct tree *node)// to push operator and operands in tree
```

```
{
```

```
++top;
```

```
stack[top]=node;
```

```
}
```

```
struct tree *pop();//to pop elements for evaluation
```

```
{
```

```
return(stack[top--]);
```

```
}
```

```
//check() function checks if the element is operator or operand
```

```
int check(char ch)
```

```
{
```



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

```
if(ch=='+' || ch=='-' || ch=='*' || ch=='/')
```

```
    return 2;//for operator return 2
```

```
else
```

```
    return 1;//for operand return 1
```

```
}
```

```
int cal(struct tree *node)
```

```
{
```

```
    int ch;
```

```
    ch=check(node->data);//check if the node is operator or operand
```

```
    if(ch==1)//if element is operand
```

```
        return node->data-48;
```

/\*convert numerbers which we have stored as string into int...refer ASCII Table e.g. ASCII value of 1 is 49 so node->data-48 is 49-48=1, 50-48=2 where 50 is ASCII value of 2\*/

```
    else if(ch==2)//if element is operator
```

```
    {
```

```
        if(node->data=='+')
```

```
            return cal(node->left)+cal(node->right);
```

```
        if(node->data=='-')
```

```
            return cal(node->left)-cal(node->right);
```

```
        if(node->data=='*')
```

```
            return cal(node->left)*cal(node->right);
```

```
        if(node->data=='/')
```

```
            return cal(node->left)/cal(node->right);
```

```
    }
```

```
}
```

```
//inorder traversal
```



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

```
void inorder(struct tree *node)
```

```
{
```

```
if(node!=NULL)
```

```
{
```

```
inorder(node->left);
```

```
printf("%c ",node->data);
```

```
inorder(node->right);
```

```
}
```

```
}
```

//if the element is operand then a node is created having left and right ptr set to null and it is pushed on stack

```
void operand(char b)
```

```
{
```

```
node=(struct tree*)malloc(sizeof(struct tree));
```

```
node->data=b;
```

```
node->left=NULL;
```

```
node->right=NULL;
```

```
push(node);
```

```
}
```

//if the element is operator then a node is created and we will pop two node elements i.e operands from stack and assign it to that operator. then we will push the node on the tree.

```
void operator(char a)
```

```
{
```

```
node=(struct tree*)malloc(sizeof(struct tree));
```

```
node->data=a;
```

```
node->right=pop();
```

```
node->left=pop();
```



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

```
push(node);  
}
```

```
void main()  
{  
    int i,p,k,ans;  
    char s[20];  
    printf("\nEnter a postfix Expression : ");  
    scanf("%s",s);  
    for(i=0;s[i]!='\0';i++)//expression tree is created using this loop  
    {  
        p=check(s[i]);//check if the element is operator or operand  
  
        if(p==1)  
            operand(s[i]);  
        else if(p==2)  
            operater(s[i]);  
        else  
            exit(0);  
    }  
    ans=cal(stack[top]);  
    printf("\nValue of the postfix Expression you entered is %d.",ans);  
    printf("\nThe inorder traversal of the tree is:-\n");  
    inorder(stack[top]);  
    printf("\n");  
}
```





Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

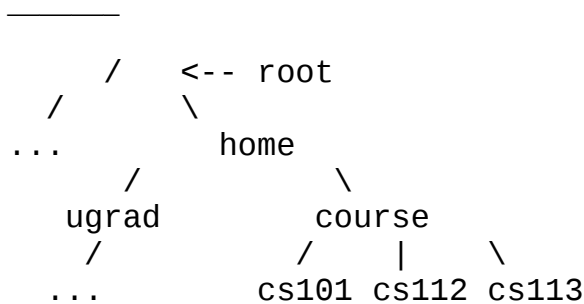
Academic Year: \_\_\_\_\_

### Applications of tree data structure

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system



2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of  $O(\log n)$  for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of  $O(\log n)$  for insertion/deletion.

4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Following are the common uses of tree.

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

## Unit 6

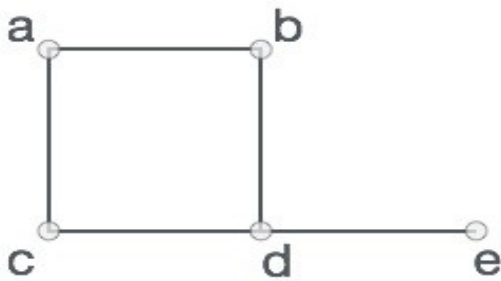
### Trees and Graphs

#### Graphs

#### Introduction to Graph

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,

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

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

#### Introduction Graph Terminologies

The following is a graph with 5 vertices and 6 edges.

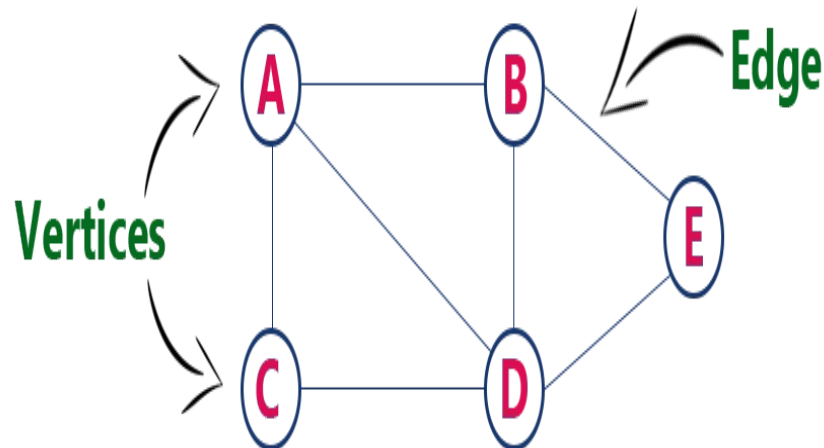
This graph  $G$  can be defined as  $G = (V, E)$

Where  $V = \{A, B, C, D, E\}$  and  $E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}$ .

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_



## Graph Terminology

We use the following terms in graph data structure...

### Vertex

A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

### Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge** - An undirected edge is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge** - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge** - A weighted edge is an edge with cost on it.

### 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.



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

### **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.

### **Incident**

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

### **Outgoing Edge**

A directed edge is said to be outgoing edge on its origin 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.

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

### 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.

### Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

### 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.

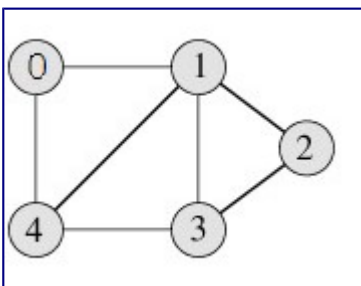
## Graph Representation

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form  $(u, v)$  called as edge. The pair is ordered because  $(u, v)$  is not same as  $(v, u)$  in case of directed graph(di-graph). The pair of form  $(u, v)$  indicates that there is an edge from vertex  $u$  to vertex  $v$ . The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. See [this](#) for more applications of graph.

Following is an example undirected graph with 5 vertices.



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

Following two are the most commonly used representations of graph.

### 1. Adjacency Matrix

### 2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

### Adjacency Matrix:

Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a graph. Let the 2D array be  $adj[i][j]$ , 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  $w$ .

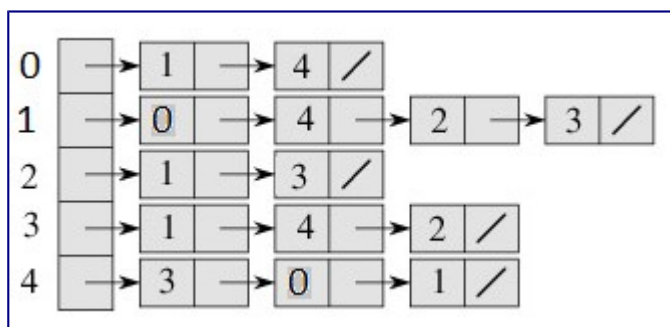
The adjacency matrix for the above example graph is:

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

Adjacency Matrix Representation of the above graph

### Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be  $array[]$ . An entry  $array[i]$  represents the linked list of vertices adjacent to the  $i$ th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

## Type of graphs

Various types of graphs have the following specializations and particulars about how they are usually drawn.

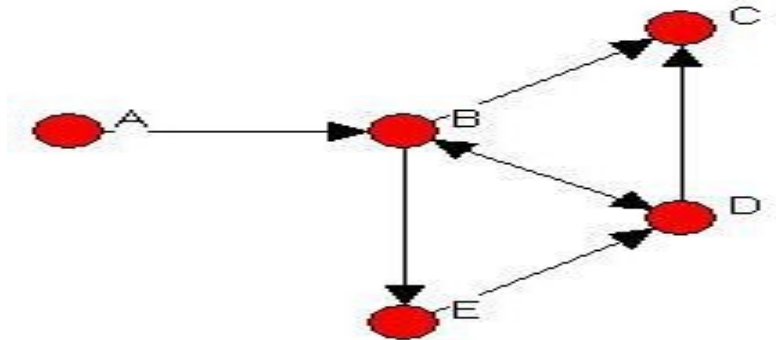
### Undirected Graphs.

In an undirected graph, the order of the vertices in the pairs in the Edge set doesn't matter. Thus, if we view the sample graph above we could have written the Edge set as  $\{(4,6),(4,5), (3,4),(3,2),(2,5)),(1,2)),(1,5)\}$ . Undirected graphs usually are drawn with straight lines between the vertices.

The adjacency relation is symmetric in an undirected graph, so if  $u \sim v$  then it is also the case that  $v \sim u$ .

### Directed Graphs.

In a directed graph the order of the vertices in the pairs in the edge set matters. Thus  $u$  is adjacent to  $v$  only if the pair  $(u,v)$  is in the Edge set. For directed graphs we usually use arrows for the arcs between vertices. An arrow from  $u$  to  $v$  is drawn only if  $(u,v)$  is in the Edge set. The directed graph below



Has the following parts.

- The underlying set for the Vertices set is capital letters.
- The Vertices set =  $\{A,B,C,D,E\}$
- The Edge set =  $\{(A,B),(B,C),(D,C),(B,D),(D,B),(E,D),(B,E)\}$

Note that both  $(B,D)$  and  $(D,B)$  are in the Edge set, so the arc between  $B$  and  $D$  is an arrow in both directions.

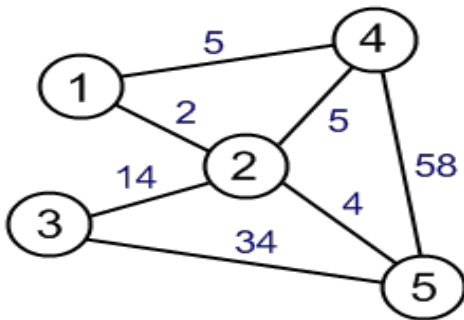
Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

### • Weighted Graphs.

A weighted graph is an edge labeled graph where the labels can be operated on by the usual arithmetic operators, including comparisons like using less than and greater than. In Haskell we'd say the edge labels are in the Num class. Usually they are integers or floats. The idea is that some edges may be more (or less) expensive, and this cost is represented by the edge labels or weight. In the graph below, which is an undirected graph, the weights are drawn adjacent to the edges and appear in dark purple.



Here we have the following parts.

- The underlying set for the Vertices set is Integer.
- The underlying set for the weights is Integer.
- The Vertices set = {1,2,3,4,5}
- The Edge set = {(1,4,5), (4,5,58), (3,5,34), (2,4,5), (2,5,4), (3,2,14), (1,2,2)}

### • Vertex labeled Graphs.

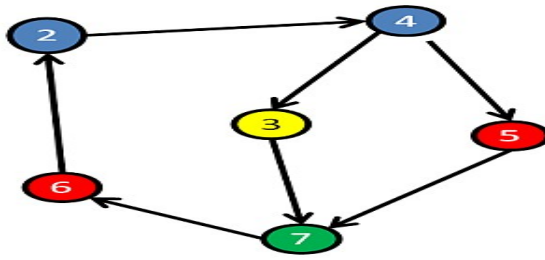
In a labeled graph, each vertex is labeled with some data in addition to the data that identifies the vertex. Only the identifying data is present in the pair in the Edge set. This is similar to the (key,satellite) data distinction for sorting.



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_



Here we have the following parts.

- The underlying set for the keys of the Vertices set is the integers.
- The underlying set for the satellite data is Color.
- The Vertices set =  $\{(2, \text{Blue}), (4, \text{Blue}), (5, \text{Red}), (7, \text{Green}), (6, \text{Red}), (3, \text{Yellow})\}$
- The Edge set =  $\{(2, 4), (4, 5), (5, 7), (7, 6), (6, 2), (4, 3), (3, 7)\}$

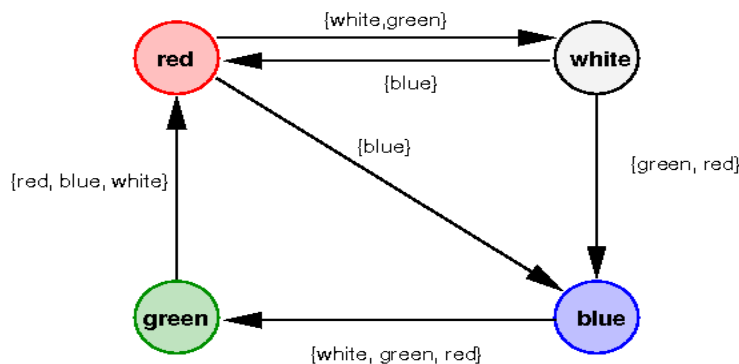
### Cyclic Graphs.

A cyclic graph is a directed graph with at least one cycle. A cycle is a path along the directed edges from a vertex to itself. The vertex labeled graph above has several cycles. One of them is  $2 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 6 \rightarrow 2$

### Edge labeled Graphs.

An Edge labeled graph is a graph where the edges are associated with labels. One can indicate this by making the Edge set be a set of triples. Thus if  $(u, v, X)$  is in the edge set, then there is an edge from  $u$  to  $v$  with label  $X$

Edge labeled graphs are usually drawn with the labels drawn adjacent to the arcs specifying the edges.



Here we have the following parts.

- The underlying set for the Vertices set is Color.



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

- The underlying set for the edge labels is sets of Color.
- The Vertices set = {Red,Green,Blue,White}
- The Edge set = {(red,white,{ white,green}) ,(white,red,{ blue}) ,(white,blue,{ green,red}) ,(red,blue,{ blue}) ,(green,red,{ red,blue,white}) ,(blue,green,{ white,green,red})}



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

## **Graph traversal:Depth first search(DFS)&Breadth First search(BFS)**

### **Minimum Spanning Tree :**

#### **What is a Spanning Tree?**

Given an undirected and connected graph  $G=(V,E)$

, a spanning tree of the graph  $G$  is a tree that spans  $G$  (that is, it includes every vertex of  $G$ ) and is a subgraph of  $G$  (every edge in the tree belongs to  $G$ )

)

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

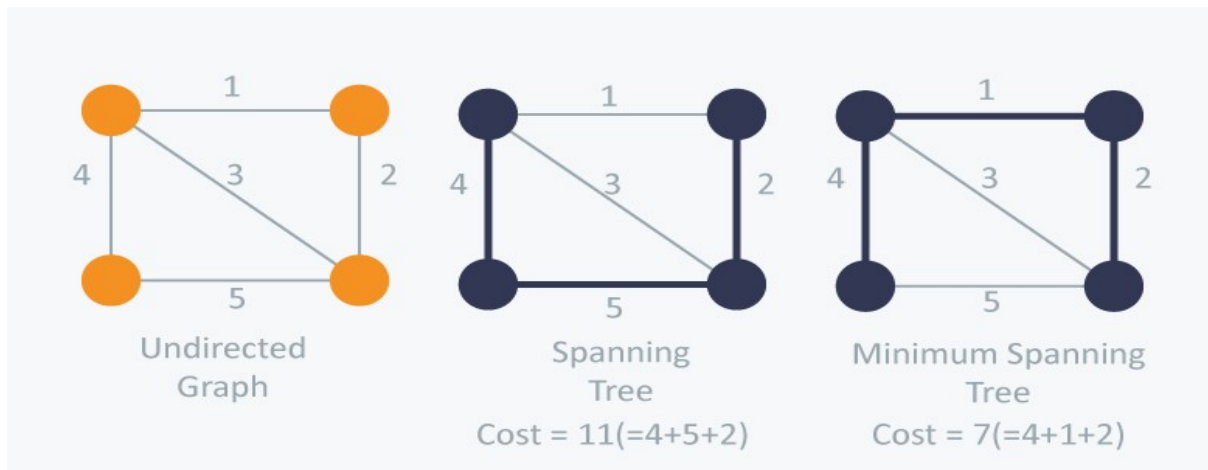
Academic Year: \_\_\_\_\_

# What is a Minimum Spanning Tree?

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

## Prim's Algorithm

Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms.

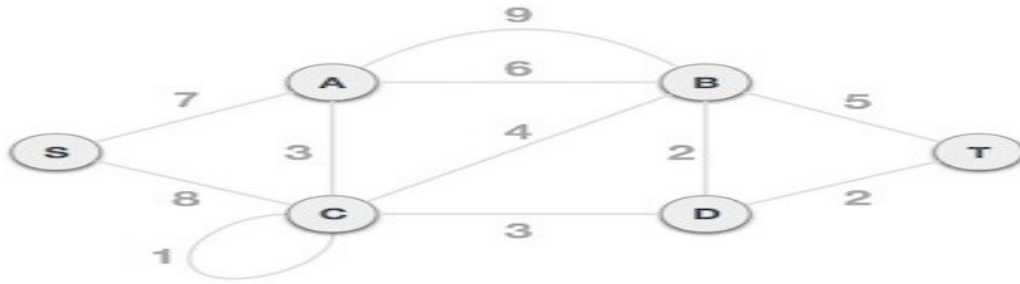
Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –

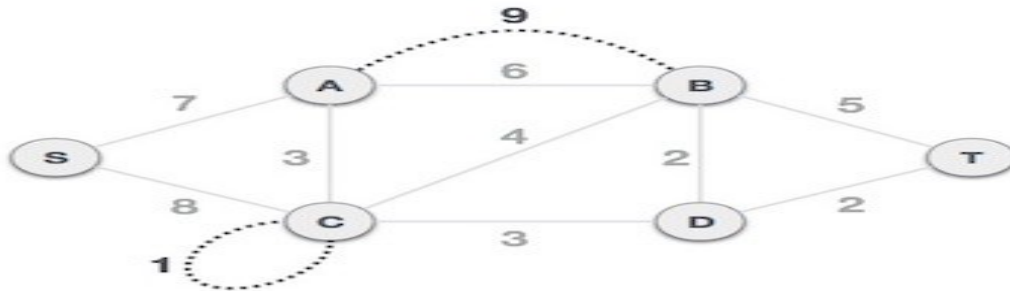
Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

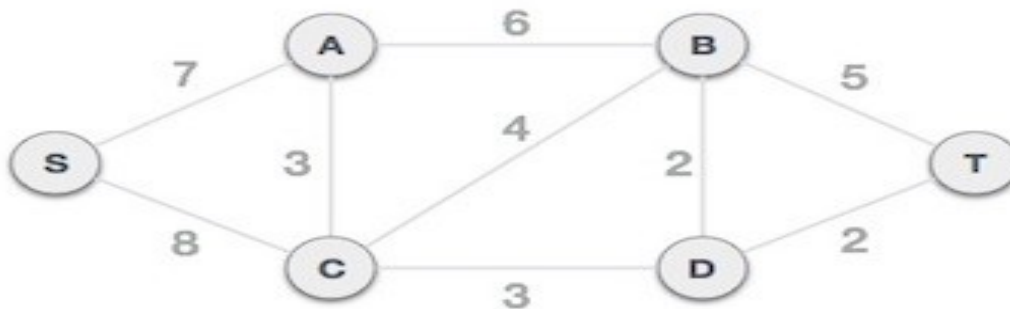
Academic Year: \_\_\_\_\_



### Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.



### Step 2 - Choose any arbitrary node as root node

In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

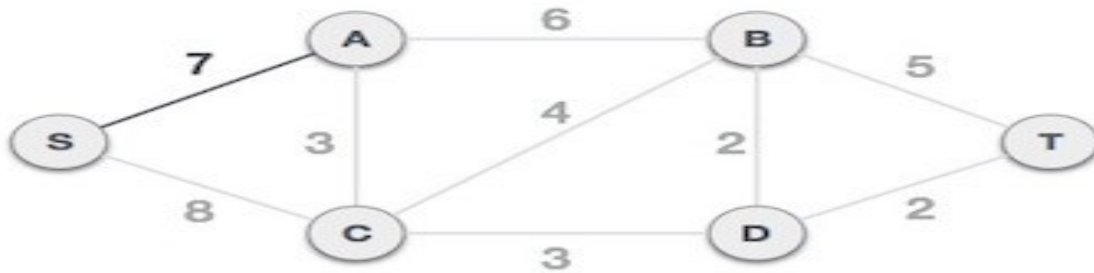
### Step 3 - Check outgoing edges and select the one with less cost

After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

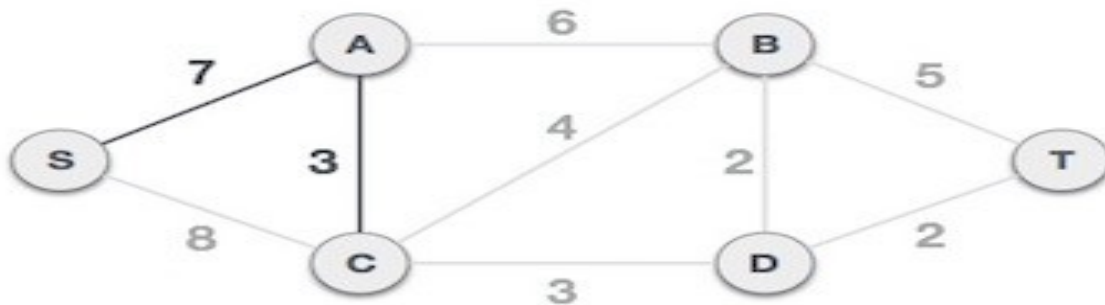
Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

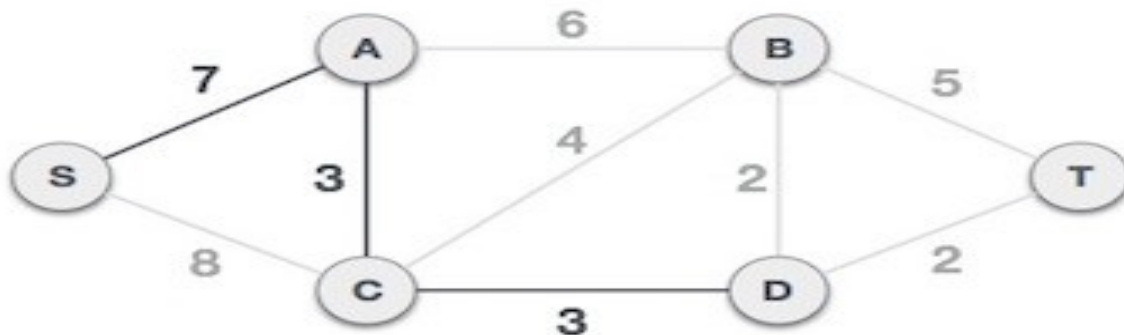
Academic Year: \_\_\_\_\_



Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.

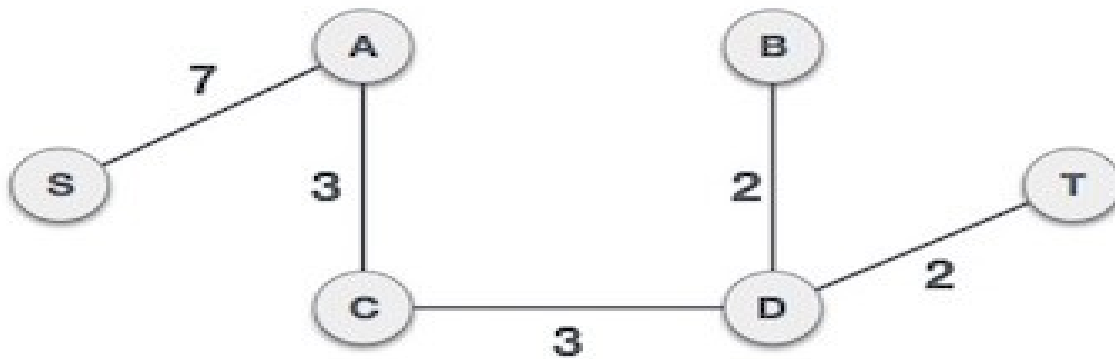


After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_



We may find that the output spanning tree of the same graph using two different algorithms is same.

## Kruskal's Algorithm

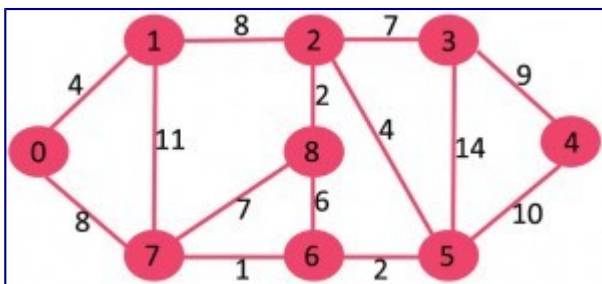
Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

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 cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are  $(V-1)$  edges in the spanning tree.

### **Consider following example:**

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example:  
Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having

Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

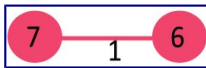
$(9 - 1) = 8$  edges.

After sorting:

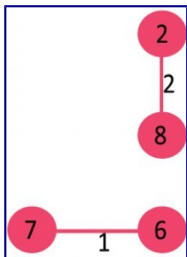
Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges

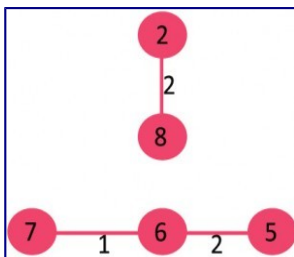
1. Pick edge 7-6: No cycle is formed, include it.



2. Pick edge 8-2: No cycle is formed, include it.



3. Pick edge 6-5: No cycle is formed, include it.



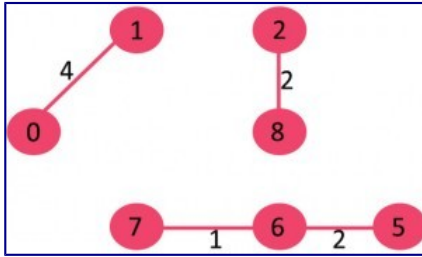
4. Pick edge 0-1: No cycle is formed, include it.



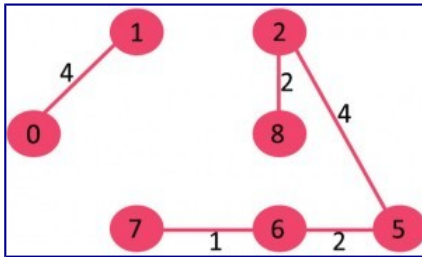
Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

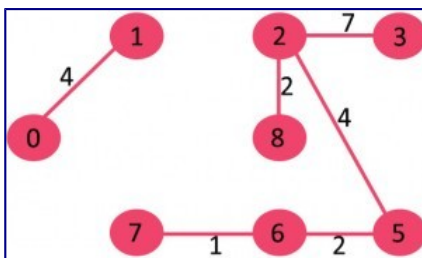


5. Pick edge 2-5: No cycle is formed, include it.



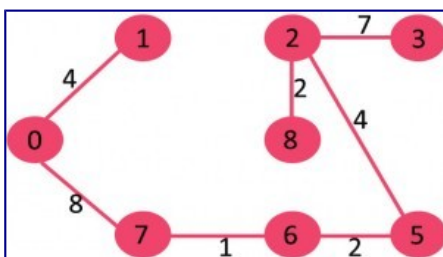
6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

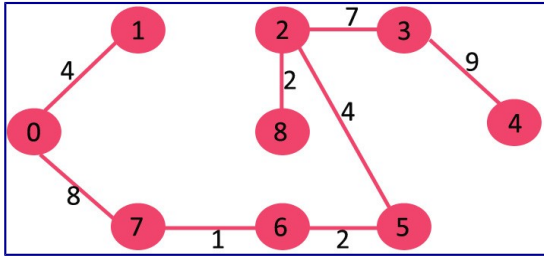
11. Pick edge 3-4: No cycle is formed, include it.



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_



Since the number of edges included equals  $(V - 1)$ , the algorithm stops here.



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

## **Shortest Path Algorithm – Dijkstra's Algorithm**

**Dijkstra's algorithm** is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

### **Algorithm**

Let the node at which we are starting be called the **initial node**. Let the **distance of node Y** be the distance from the **initial node** to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be  $6 + 2 = 8$ . If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3



Semester: \_\_\_\_\_

Subject: \_\_\_\_\_

Academic Year: \_\_\_\_\_

## **Applications of graph**

1. Network design.

– *telephone, electrical, hydraulic, TV cable, computer, road*

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

2. Approximation algorithms for NP-hard problems.

– *traveling salesperson problem, Steiner tree*

3. max bottleneck paths

4. LDPC codes for error correction

5. image registration with Renyi entropy

6. learning salient features for real-time face verification

7. reducing data storage in sequencing amino acids in a protein

8. model locality of particle interactions in turbulent fluid flows

9. autoconfig protocol for Ethernet bridging to avoid cycles in a network

### **10. Cluster analysis**

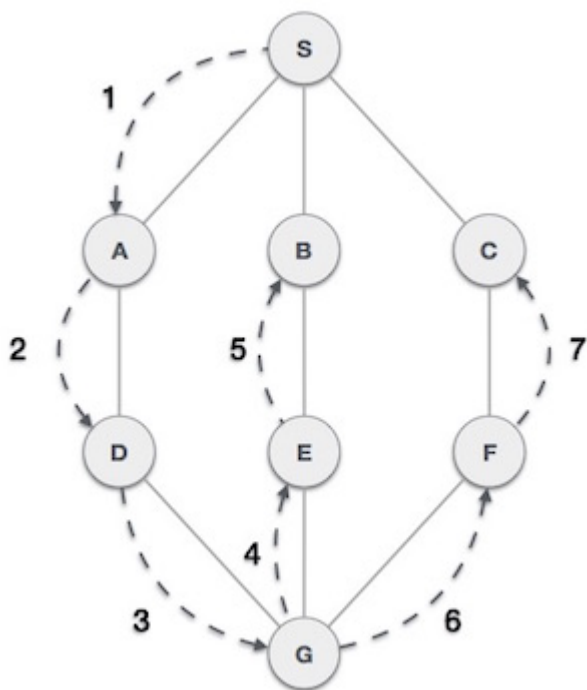
k clustering problem can be viewed as finding an MST and deleting the k-1 most expensive edges.

## DATA STRUCTURE - DEPTH FIRST TRAVERSAL

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/depth\\_first\\_traversal.htm](https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm)

Copyright © tutorialspoint.com


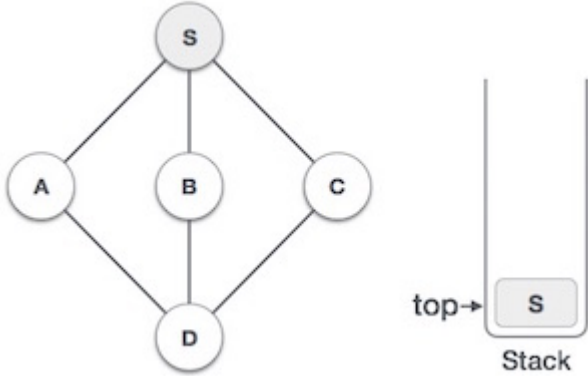
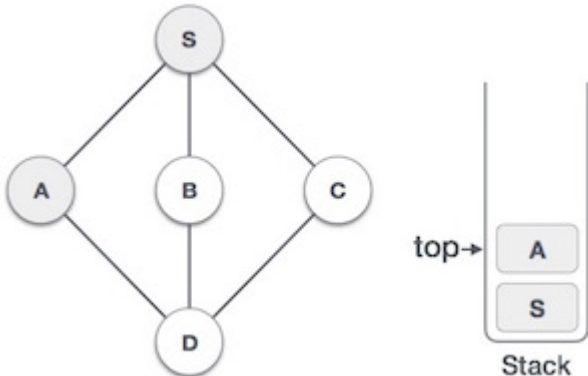
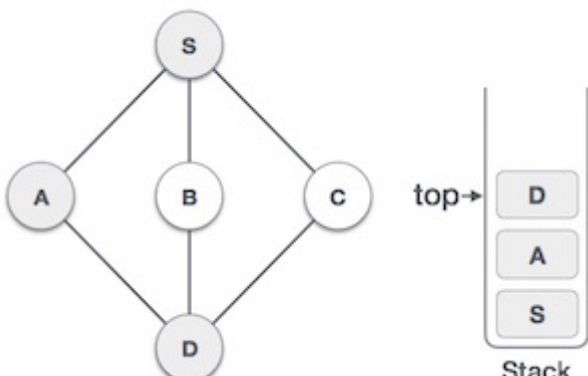
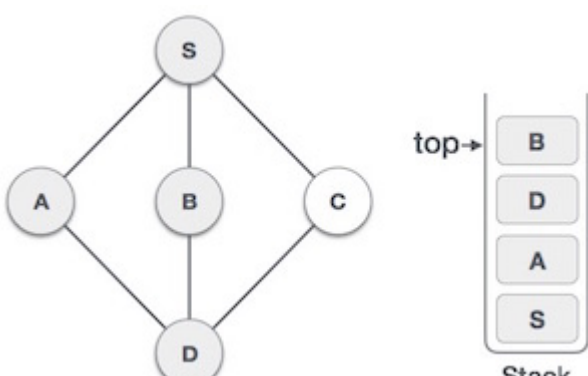
Depth First Search *DFS* algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

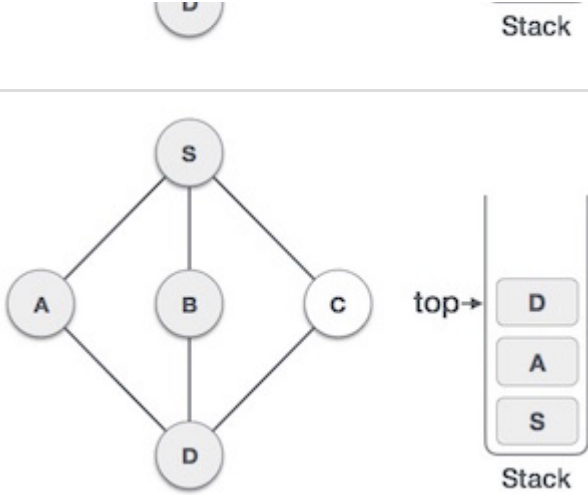
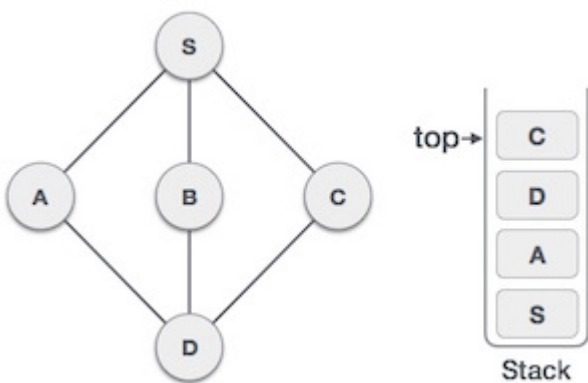
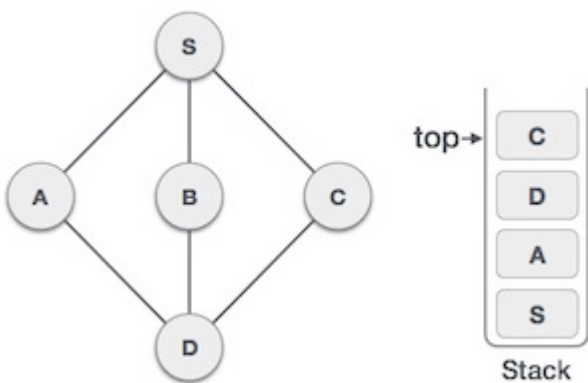


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to G. It employs the following rules.

- **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.		Initialize the stack.

		
2.		<p>Mark <b>S</b> as visited and put it onto the stack.</p> <p>Explore any unvisited adjacent node from <b>S</b>. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.</p>
3.		<p>Mark <b>A</b> as visited and put it onto the stack.</p> <p>Explore any unvisited adjacent node from <b>A</b>. Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.</p>
4.		<p>Visit <b>D</b> and mark it as visited and put onto the stack. Here, we have <b>B</b> and <b>C</b> nodes, which are adjacent to <b>D</b> and both are unvisited. However, we shall again choose in an alphabetical order.</p>
5.		<p>We choose <b>B</b>, mark it as visited and put onto the stack. Here <b>B</b> does not have any unvisited adjacent node. So, we pop <b>B</b> from the stack.</p>

		
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of the stack.
7.		Only unvisited adjacent node is from <b>D</b> is <b>C</b> now. So we visit <b>C</b> , 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.

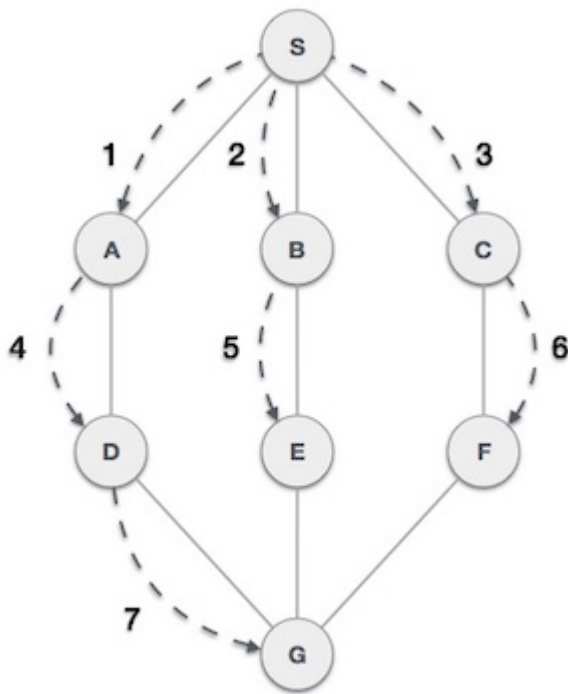
To know about the implementation of this algorithm in C programming language, [click here](#).

## DATA STRUCTURE - BREADTH FIRST TRAVERSAL

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/breadth\\_first\\_traversal.htm](https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm)

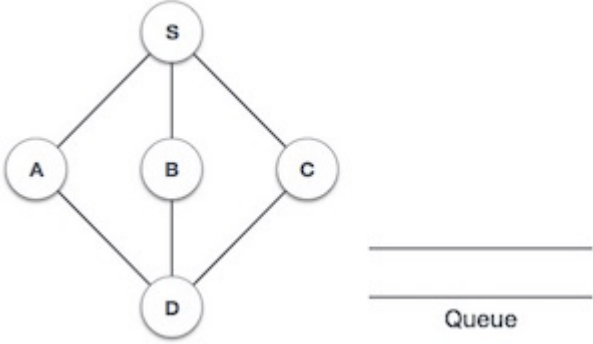
Copyright © tutorialspoint.com

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.

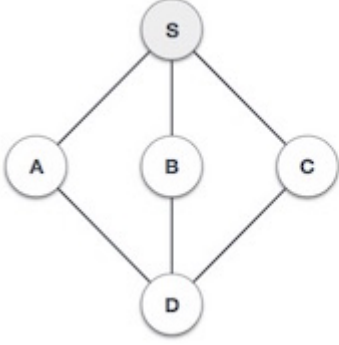
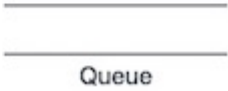
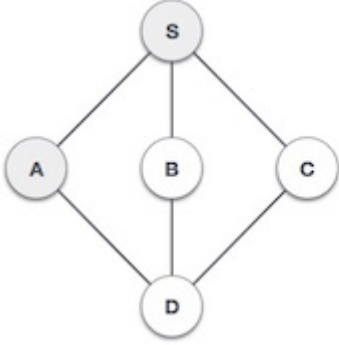
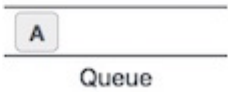
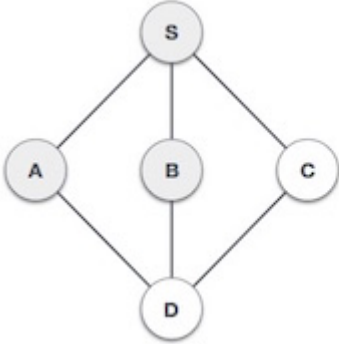
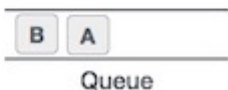
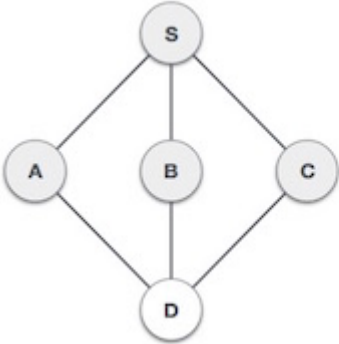

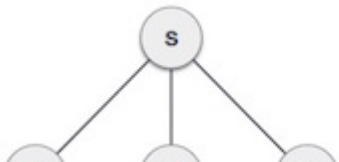


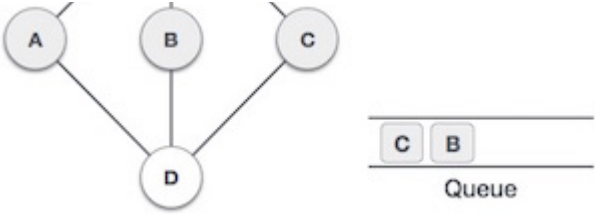
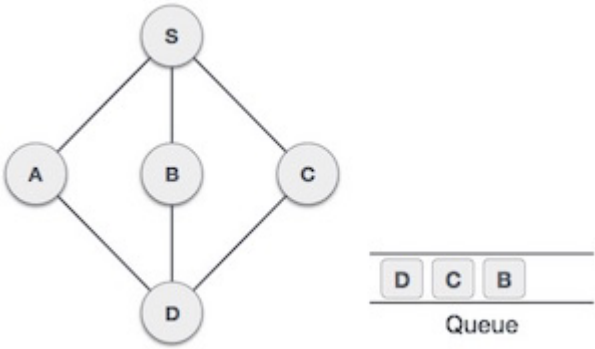
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **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.		Initialize the queue.



2.	 	We start from visiting <b>S</b> <i>starting node</i> , and mark it as visited.
3.	 	We then see an unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> , mark it as visited and enqueue it.
4.	 	Next, the unvisited adjacent node from <b>S</b> is <b>B</b> . We mark it as visited and enqueue it.
5.	 	Next, the unvisited adjacent node from <b>S</b> is <b>C</b> . We mark it as visited and enqueue it.
6.		Now, <b>S</b> is left with no unvisited adjacent nodes. So, we dequeue and find <b>A</b> .

	 <p>Diagram showing nodes A, B, and C connected to node D. A queue contains nodes C and B.</p>	
7.	 <p>Diagram showing a graph with nodes S, A, B, C, and D. Node S is at the top, connected to A, B, and C. A, B, and C are connected to D. A queue contains nodes D, C, and B.</p>	<p>From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it as visited and enqueue it.</p>

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.

The implementation of this algorithm in C programming language can be [seen here](#).