LAKSHMI NARAIN COLLEGE OF TECHNOLOGY & SCIENCE, BHOPAL

CS-303

DATA STRUCTURE AND ALGORITHM

Vision of the Department

To be recognized for keeping innovation, research and excellence abreast of learning in the field of computer science & engineering to cater the global society.

Mission of the Department

M1: To provide an exceptional learning environment with academic excellence in the field of computer science and engineering.

M2: To facilitate the students for research and innovation in the field of software, hardware and computer applications and nurturing to cater the global society.

M3: To establish professional relationships with industrial and research organisations to enable the students to be updated of the recent technological advancements.

M4: To groom the learners for being the software professionals catering the needs of modern society with ethics, moral values and full of patriotism.

Program Educational Objectives (PEO's)

PEO1: The graduate will have the knowledge and skills of major domains of computer science and engineering in providing solution to real world problems most efficiently.

PEO2: The graduate will be able to create and use the modern tools and procedures followed in the software industry in the relevant domain.

PEO3: The graduate will be following the ethical practices of the software industry and contributing to the society as a responsible citizen.

PEO4: The graduate will have the innovative mindset of learning and implementing the latest developments and research outcomes in the computer hardware and software to keep pace with the fast changing socio economic world.

LAKSHMI NARAIN COLLEGE OF TECHNOLOGY & SCIENCE, BHOPAL

CS-303

DATA STRUCTURE AND ALGORITHM

COURSE OUTCOMES

Student will be able to:

CO1: Use primitive operations on arrays, structures, stack and queue data structures

CO2: Compare stack and queue data structures along with their merits and demerits.

CO3: Develop programs to perform primitive operations on linked lists

CO4: Utilize Dijkstra's algorithm to find spanning tree for a given graph.

CO5: Apply quick and merge sorting methods in problem solving.

LIST OF EXPERIMENTS

1. Write a program to insert an element at particular position in one dimensional array.
2. Write a program to insert (Push) an element into a Stack.
3. Write a program to perform insertion operation on linear Queue.
4. Write a program to insert a node/an element it specific position into Singly Linked List.
5. Write a program to implement the Binary search algorithm.
6. Write a program to implement the Quick sort algorithm.
7. Write a program to Construct Max-Heap.
8. Write a program to delete an element from a Max-Heap
9. Write a program to search an element from AVL tree.
10. Write a program to find MST using Prism's algorithm.

EXPERIMENT -1

Arrays:

The array is a data structure where values or items are placed in a linear order, which means the memory assigned to each item is contiguous. The data type of an array is the same for all the elements present in it.

With the contiguous memory allocation, it becomes easier to find out the memory location of any element in the array by just knowing the first memory location and adding the offset.

Array Declaration:

An "array declaration" names the array and specifies the type of its elements. It can also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements.

Syntax: data type array_Name [array_Size];

Points to remember about Array:

1.It can hold multiple values of a single type.

2.Elements are referenced by the array name and an ordinal index.

3.Each element is a value, index pair.

4.Indexing begins at zero.

5. The array forms a contiguous list in memory.

Algorithm:

Here LA is a Linear Array with N elements and K is a positive integer such that K<+N. This algorithm inserts an element ITEM into the Kth position in LA.

```
1. [Initialize counter.] Set J:=N.
2. Repeat Steps 3 and 4 while J>=K.
3. [Move Jth element downward.] Set LA[J+1]:= LA[j].
4. [Decrease counter.]Set J:= J-1.
      [End of Step 2 loop.]
5. [Insert element.] Set LA[K]:=ITEM.
6. [Reset N.] Set N:N+1.
7.  Exit
```

EXPERIMENT -2

STACK:
Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack

Basic terminology associated with Stacks:
Stack Pointer (TOP) Keeps track of the current position of the stack Overflow. Occurs when we try to insert (push) more information on a stack than it can hold. Underflow. Occurs when we try to delete (pop) an item off a stack, which is empty.

Basic Operation Associated with Stacks:

1) Insert (Push) an item into the stack.
2) Delete (Pop) an item from the stack**.**

Algorithm: For push a element

Push an element:

```
STEP 1 START

STEP 2 Store the element to push into array

STEP 3 Check if top== (MAXSIZE-1) then stack is full else go to step 4

STEP 4 Increment top as top = top+1

STEP 5 Add element to the position stk[top]=num
STEP 6 STOP
```

Algorithm for delete an element:

Pop an element:

```
STEP 1 START
STEP 2 Check if top== (-1) then stack is empty else goto step 4
STEP 3 Access the element top is pointing num = stk[top];
STEP 4 Decrease the top by 1 top = top-1;
STEP 6 STOP
```

EXPERIMENT -3

Queue:

A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.

1. Queue is referred to be as First In First Out list.

2. For example, people waiting in line for a rail ticket form a queue.

Basic operation associated on Queues:

1) Insertion an item into Queue.

2) Delete an item into the Queue.

Algorithm for Enqueue :

```
STEP 1 START
STEP 2 Store the element to insert
STEP 3 Check if REAR= MAX-1 then write Overflow else goto step 5
STEP 4 Check if REAR= -1 then
          set FRONT=REAR=0
          else
          set REAR=REAR+1
STEP 5 set QUEUE [REAR]=NUM
STEP 6 STOP
```

Algorithm for delete a element in linear Queue:

Algorithm for Dequeue

```
Dequeue for an element
STEP 1 START
STEP 2 Store the element to insert
STEP 3 Check if FRONT=-1 or FRONT > REAR writes Underflow else goto
step 4
STEP 4 Set VAL=QUEUE [FRONT]
          Set FRONT= FRONT + 1
```

STEP 5 STOP

<u>EXPERIMENT -4</u>

Link List:

Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.

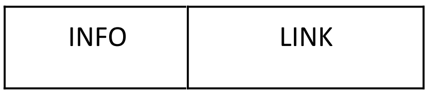
A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.

The last node of the list contains pointer to the null.

Components of a node:

Data- stores the information

Link-stores the address of a next node.



Structure of node

Syntax:

```
Struct node {

Int data:

Node*next;

}
```

Types of linked lists

1. Singly linked list:
2. Doubly linked list:
3. Circular linked list:

Key points of linked list in data Structure

- Always have a reference to the first node.
- Check whether your dynamic allocation of a node did succeed.

- Check whether you are dereferencing a NULL pointer
- Check whether you are freeing a NULL pointer,
- Check whether you are freeing a node after its required use.

Algorithm to insert a node into Singly linked list

```
Step 1: IF PTR = NULL
Write OVERFLOW
     Go to Step 7
    [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET NEW_NODE → NEXT = HEAD
Step 6: SET HEAD = NEW_NODE
Step 7: EXIT
```

EXPERIMENT -5

Binary Search :
Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm for Binary Search

```
#include <iostream>

using namespace std;

int binarySearch(int array[], int low, int high, int number_to_search_for) {

while (low <= high) {

int mid = low + (high - low) / 2;

if (number_to_search_for == array[mid]){

return mid;

}

if (number_to_search_for> array[mid]){

low = mid + 1;

}

if (number_to_search_for< array[mid]){

high = mid - 1;

}

}

return -1;

}

int main(void) {

int arrayOfNums[] = {2,4,7,9,10,13,20};

int n = sizeof(arrayOfNums) / sizeof(arrayOfNums[0]);
int result = binarySearch(arrayOfNums, 0, n - 1, 13);

if (result == -1){

printf("Element doesn't exist in the array");

}

else{

printf("The index of the element is %d", result);

}

// The index of the element is 5

}
```

EXPERIMENT -6

SORTING:

Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order. For example, consider an array A = {A1, A2, A3, A4, ??An }, the array is called to be in ascending order if element of A are arranged like A1 > A2 > A3 > A4 > A5 > ? >An .

QUICK SORT:

Quicksort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

The recursive algorithm consists of four steps:

1. If there are one or less elements in the array to be sorted, return immediately.

2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)

3. Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.

4. Recursively repeat the algorithm for both halves of the original array.

The quick sort is by far the fastest of the common sorting algorithms.

Algorithm for Quick Sort

```
PARTITION (array A, start, end)
{
 1 pivot ? A[end]
 2 i ? start-1
 3 for j ? start to end -1 {
 4 do if (A[j] < pivot) {
 5 then i ?i + 1
 6 swap A[i] with A[j]
7  }}
 8 swap A[i+1] with A[end]
 9 return i+1
}
```

EXPERIMENT -7

HEAP:

Heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

There are two types of Heap

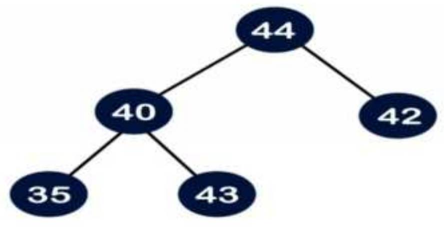
1. Max-Heap
2. Min-Heap

MAX-HEAP

The value of the parent node is greater than or equal to its children.

other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]

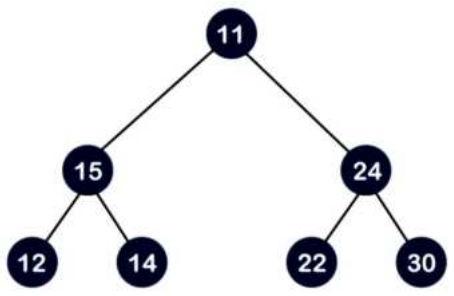


MIN-HEAP

Min Heap: The value of the parent node should be less than or equal to either of its children.

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] <= A[i]



Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Mis Heap is similar but we go for min values instead of max ones. We are going to derive an algorithm for max-heap by inserting one element at a time. At any point of

time, heap must maintain its property. While insertion, we also assume that we are inserting a node in

already heapified tree

Step 1-Create a new node at the end of heap

Step 2-Assign new value to the node

Step 3-Compare the value of this child node with its parent Step 4-If value of parent is less than child, then swap them

Step 5-Repeat step 3 & 4 until Heap property holds.

child node

we use earlier.

Note In Min Heap construction algorithm we expect the value of parent node to be less than that of Let's understand Max Heap construction by an animated illustration. We take the same input sample that

Max Heap Deletion Algorithm

Lets derive an algorithm to delete from max-heap. Deletion in Max (or Min) Heap is always happen at

the root to remove the Maximum (or minimum) value.

Step 1-Remove root node.

Step 2-Move the last element of last level to root.

Step 3-Compare the value of this child node with its parent.

Step 4-If value of parent is less than child, then swap them Step 5-Repeat step 3 & 4 until Heap property holds

EXPERIMENT -8

HEAP:

Heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

There are two types of Heap

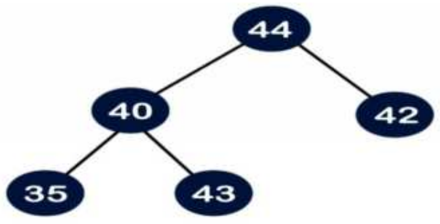
1. Max-Heap
2. Min-Heap

MAX-HEAP

The value of the parent node is greater than or equal to its children.

other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]



Max Heap Deletion Algorithm

Lets derive an algorithm to delete from max-heap. Deletion in Max (or Min) Heap is always happen at

the root to remove the Maximum (or minimum) value.

Step 1-Remove root node.

Step 2-Move the last element of last level to root.

Step 3-Compare the value of this child node with its parent.

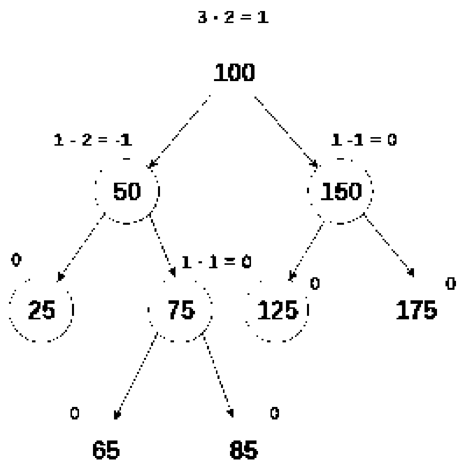
Step 4-If value of parent is less than child, then swap them Step 5-Repeat step 3 & 4 until Heap property holds

EXPERIMENT -9

AVL Tree:

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

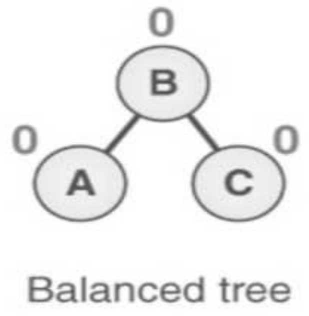


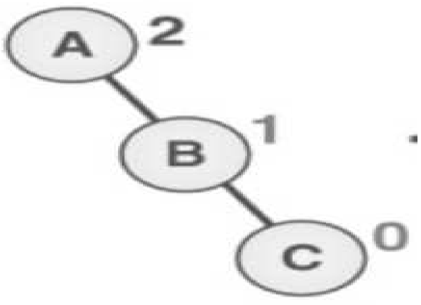
AVL Tree

It is observed that BST's worst-case performance closes to linear search algorithms, that is O(n). In real time data we cannot predict data pattern and their frequencies. So a need arises to balance out existing BST

Named after their inventor Adelson, Velski& Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of left and right sub-trees and assures that the difference is not more than 1. This difference is called Balance Factor

Here we see that the first tree is balanced and next two trees are not balanced-





AVL Rotations

To make itself balanced, an AVL tree may perform four kinds of rotations-

Left rotation Right rotation

Left-Right rotation

. Right-Left rotation

First two rotations are single rotations and next two rotations are double rotations. Two have an unbalanced

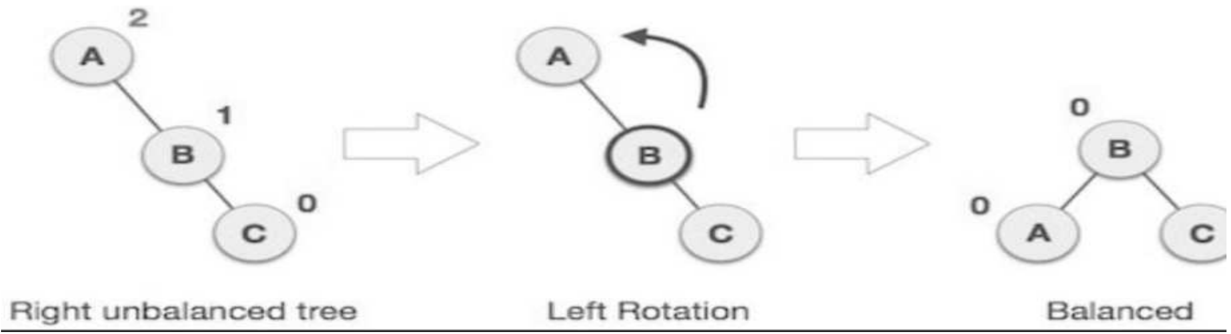
tree we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

If a tree become unbalanced, when a node is inserted into the right subtree of right subtree, then we

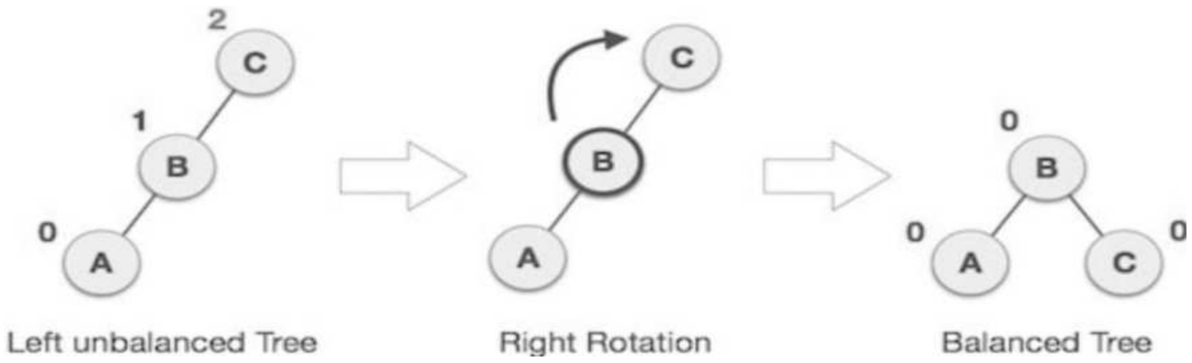
perform single left rotation

-



Right Rotation

AVL tree may become unbalanced if a node is inserted in the left subtree of left subtree. The tree then needs a right rotation**.**



Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

Step 1:

Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4 If both are not matching, then check whether search element is smaller or larger than that node

value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6 If search element is larger, then continue the search process in right subtree.
Step 7 Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9 If we reach to a leaf node and it is also not matching, then display "Element terminate the function.

Insertion Operation in AVL Tree

not found" and

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.
Step 2 After insertion, check the Balance Factor of every node.

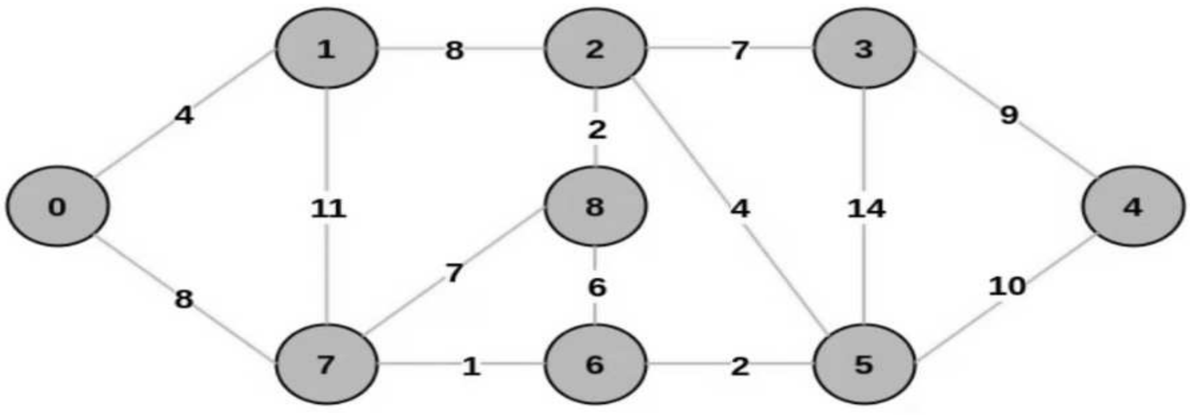
Step 3 If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
Step 4: If the Balance Factor of any node is other than 0 or 1 or-1 then tree is said to be imbalanced.

Then perform the suitable Rotation to make it balanced. And go for next operation

EXPERIMENT -10

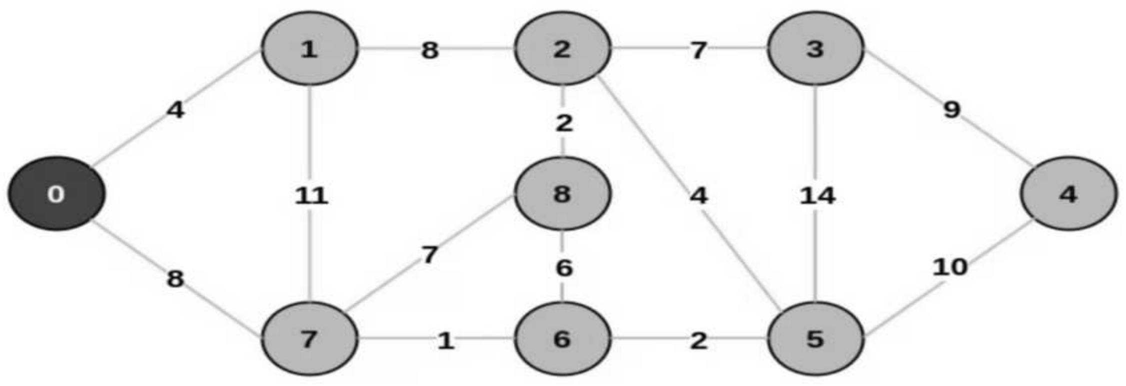
PRISM'S Algorithm:
Prim's algorithm is a greedy algorithm that is used to find the minimum spanning tree for a network. It is used to connect every node together using the smallest possible total obtained when the edges are added together. For example, Prim's algorithm has been used on the network below to find the minimum spanning tree.

Step 1: Determine an arbitrary vertex as the starting vertex of the MST.
Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
Step 3: Find edges connecting any tree vertex with the fringe vertices.
Step 4: Find the minimum among these edges.
Step 5: Add the chosen edge to the MST if it does not form any cycle.
Step 6: Return the MST and exit



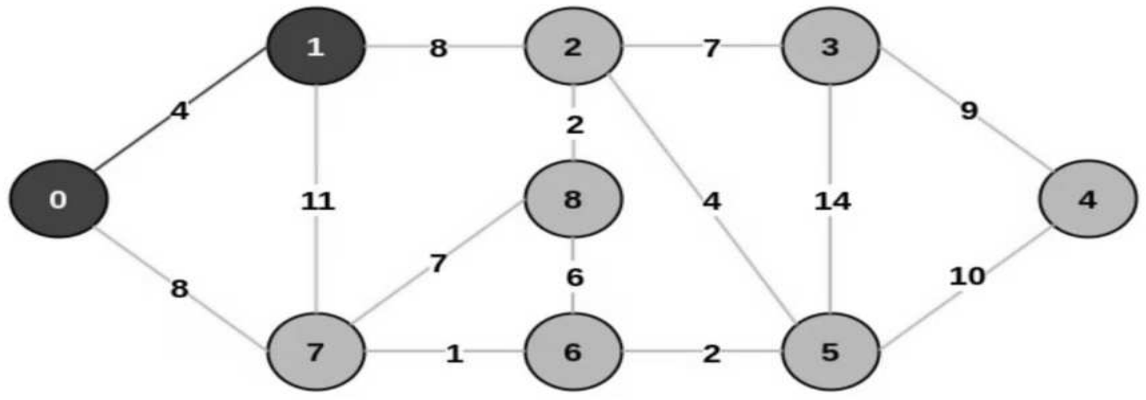
Example of a Graph

Step 1: Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex 0 as the starting vertex.



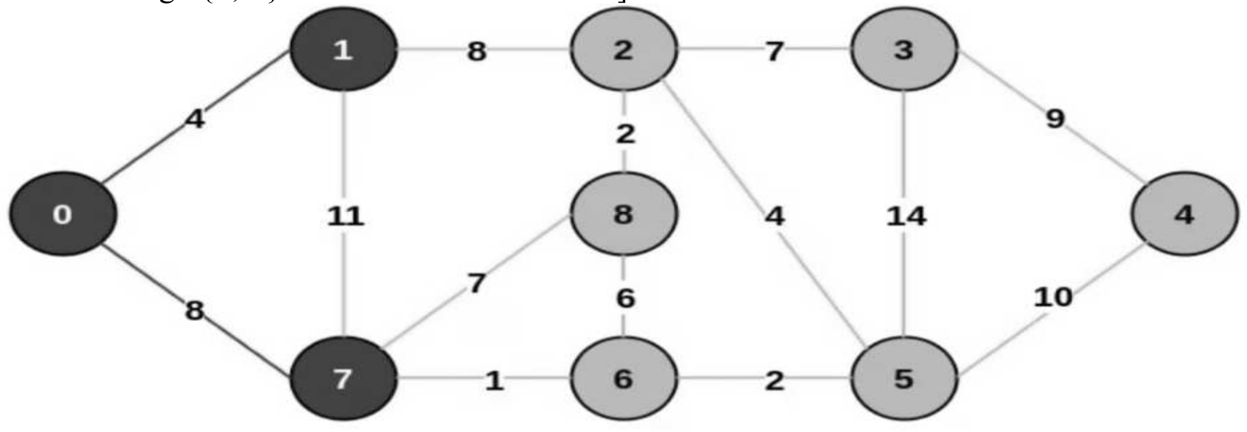
Select an arbitrary starting vertex. Here we have selected 0

Step 2: All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.



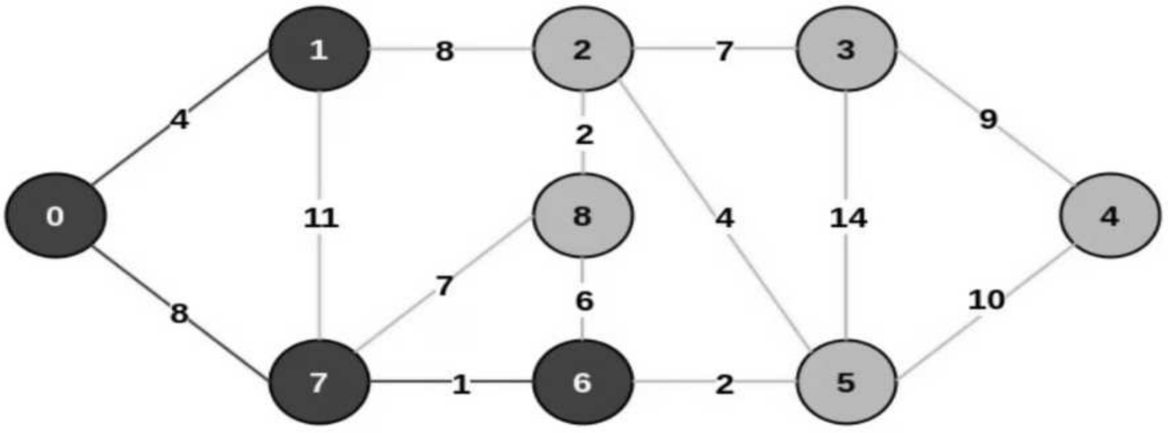
Minimum weighted edge from MST to other vertices is 0-1 with weight 4

Step 3: The edges connecting the incomplete MST to other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].



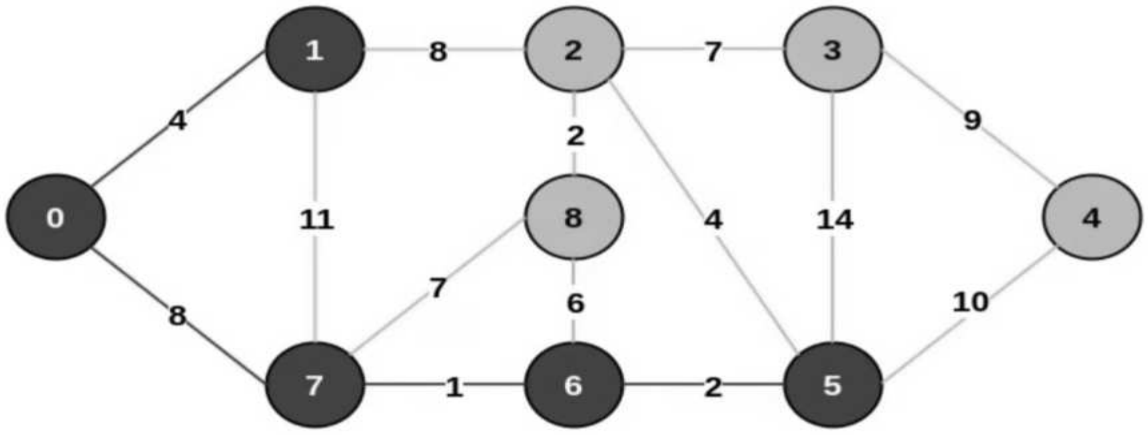
Minimum weighted edge from MST to other vertices is 0-7 with weight 8

Step 4: The edges that connect the incomplete MST with the fringe vertices are {1, 2}, {7, 6} and {7, 8}. Add the edge {7, 6} and the vertex 6 in the MST as it has the least weight (i.e., 1).



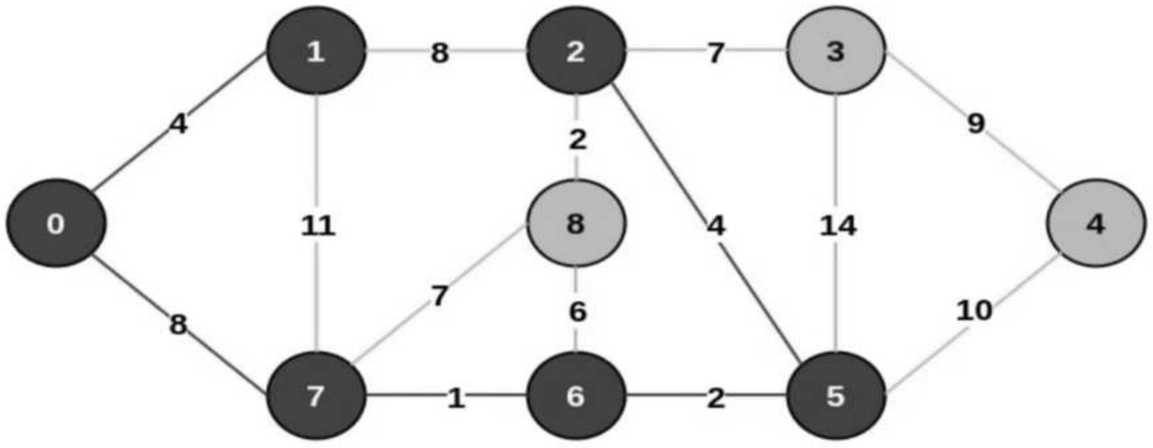
Minimum weighted edge from MST to other vertices is 7-6 with weight 1

Step 5: The connecting edges now are {7, 8}, {1, 2}, {6, 8} and {6, 5}. Include edge {6, 5} and vertex 5 in the MST as the edge has the minimum weight (i.e., 2) among them.



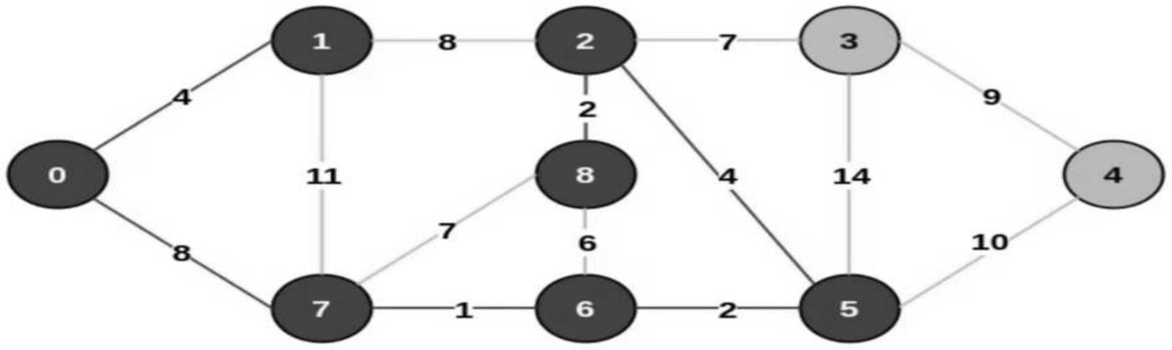
Minimum weighted edge from MST to other vertices is 6-5 with weight 2

Step 6: Among the current connecting edges, the edge {5, 2} has the minimum weight. So include that edge and the vertex 2 in the MST.



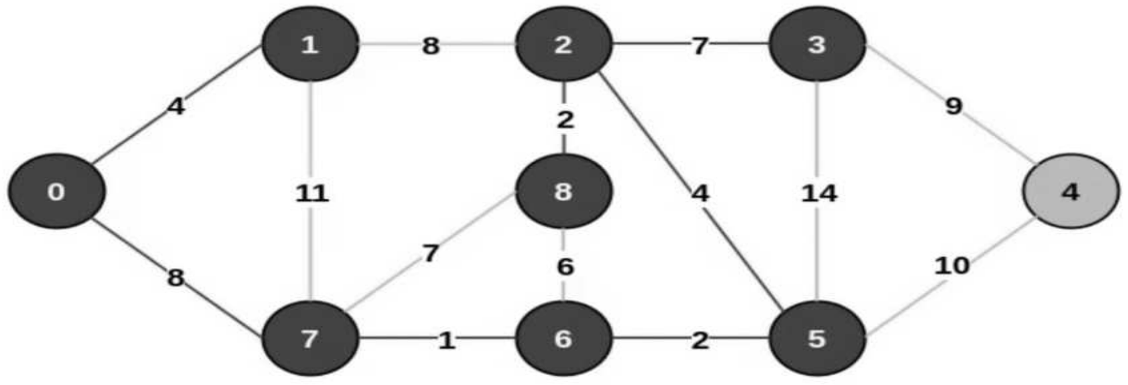
Minimum weighted edge from MST to other vertices is 5-2 with weight 4

Step 7: The connecting edges between the incomplete MST and the other edges are {2, 8}, {2, 3}, {5, 3} and {5, 4}. The edge with minimum weight is edge {2, 8} which has weight 2. So include this edge and the vertex 8 in the MST.



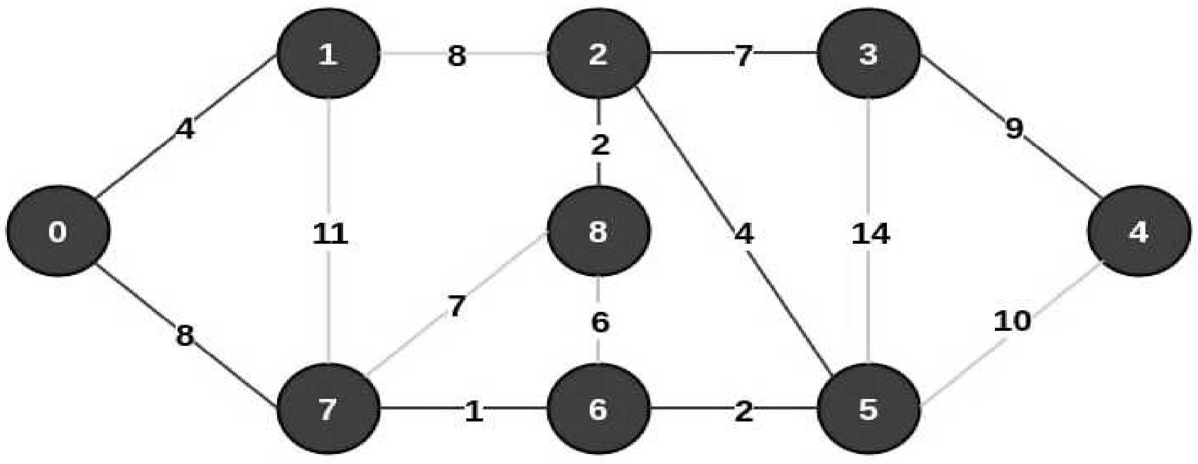
Minimum weighted edge from MST to other vertices is 2-8 with weight 2

Step 8: See here that the edges {7, 8} and {2, 3} both have same weight which are minimum. But 7 is already part of MST. So we will consider the edge {2, 3} and include that edge and vertex 3 in the MST.



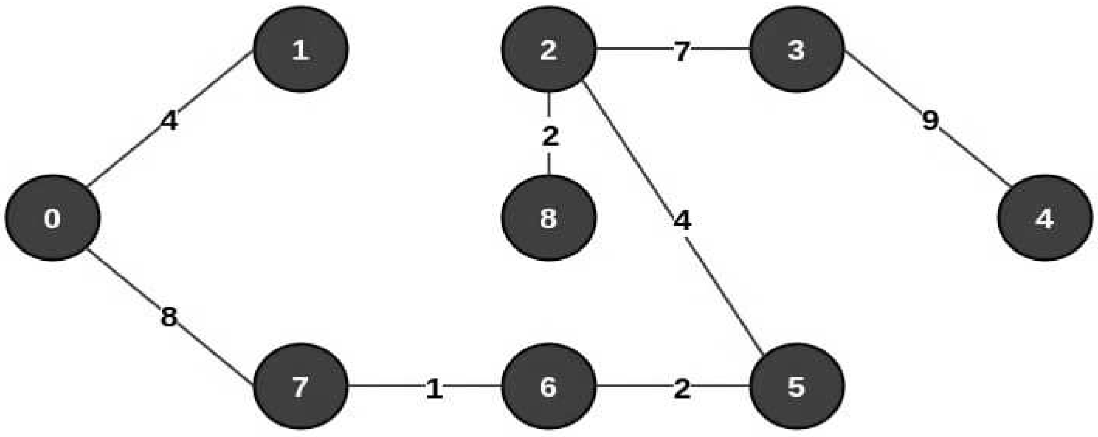
Minimum weighted edge from MST to other vertices is 2-3 with weight 7

Step 9: Only the vertex 4 remains to be included. The minimum weighted edge from the incomplete MST to 4 is {3, 4}.



Minimum weighted edge from MST to other vertices is 3-4 with weight 9

The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = 37.



The final structure of MST