What about maps

Samarth Pandey

Index

1. Introduction

1.1. **Introduction**

The questioning begins with how is software like google maps,Apple maps etc. are able to provide us with the shorted route? ,how come the route is the least time taking?, how is such complex coding done ? , what all is used? Are some of the question that comes to our mind when we see the magic that these software does.

In this book we would be looking at some basic ideas and their implementation on what all might be done to create such marvellous programmes that provides us with such astonishing results.

Here, it had been tried to explain the basic ideas and comparing them with other ideas and reaching to the conclusion of what is the best thing and how it can be used at theoretical as well as coding level.

The actual implementation of there software might be different from what that had been attempted to be explained in this book.

1.2 General idea

In this section we are going to describe what all things we require for creating a program that helps us to find the shortest or quickest part between 2 places.

While working on or using a software like google maps , it usually require the user to input the source (from where he is present or the place with which he starts his journey) and the destination where he need to travel to.

The basic implementation that most of us would think of would be , make a list of all possible routes and then decide from them which one is the shortest or the quickest one.

The problem that we would face here would be that, on a large case there would be many thousands of routes and going through each of them would take a lot of time and would make our program slow. We need to come up with and effective algorithm to solve this problem.

In this book we would be looking at what different implementation could be there to solve this problem.

In the further chapter we have discussed what different methods could be used and what are the basic requirement for them,

1.3 **Implementation level**

The level of implementation can be divided into few parts. The parts of implementation are as follows :-

- Inputting vertices (may be thought s counter of road or turns)
- Connection the vertices.
- Naming the roads and setting their data..
- Storing the data of each road.(weight of edge)
- searching for the edge from at to, where we have to travel to.
- Searching for the shortest path.

First part involves inputting vertices with their data that would help us to calculate to weight of each edge which would further help in finding the shortest path.

Next we connect all the edges to create some graph like format. It is like a 2-D structure of map where each edge represent a road of map.

Then comes naming of road so that they can uniquely identified and distinguished from the other one.

Then comes calculating of weight of each edge and then storing it as the weight of the edge. The is designed is such a way so that it takes all factor in count.

1.4 **Basic requirements**

You must know the basic coding language like c or c++. The further chapters include detailed description of each data structure required for the designing such program . It involves knowledge of sorting algorithms , search algorithms, different data structure required etc and finally we would move to description of graph algorithms.

2. Pre-requisites

2.1 Coding and software

There general requirement is a software to write the code of the program . Software like vs code or vim editor can be used for that.

Basic knowledge of c is required for coding the algorithm.

2.2 Data structures required

- **Array.**
- **Linked list.**

Array:-An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

Linked list:-

A linked list is represented by a pointer to the first node of the linked list. The first node is …..

called the head. If the linked list is empty, then the value of the head is NULL.

Each node in a list consists of at least two parts:

1) data

2) Pointer (Or Reference) to the next node .

Advantage of linked list over array is that there is less problem in inserting a new element and a single packet can have multiple type of variables.

2.3 Data structures implementation

2.3.1 Single linked list

A linked list is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a node.

Declaration:-

Code :-

```
struct LinkedList{
    int data;
    struct LinkedList *next;
}
```

The data field stores the element and the next is a pointer to store the address of the next node.

Creating a node :-

```
typedef struct LinkedList *node; //used to define node pointer

node createNode(){

   node temp; // initialising the node

   temp = (node)malloc(sizeof(struct LinkedList)); // allocating memory to node pointer

   temp->next = NULL;// mae next node to it null

   return temp;//return the struct pointer

}
```

Linking the nodes of the link list:-

```
node addNode(node head, int value){
   node temp,p;// declare a temp node of traversal storing
   temp = createNode();//createNode will return a new node with data = value and next pointing to NULL.
   temp->data = value; // add element's value to data part of node
   if(head == NULL){
      head = temp;     //when 0 node are present
   }
   else{
      p  = head;//aggigning head to p
      while(p->next != NULL){
         p = p->next;//traverse until the last node is reached
      }
      p->next = temp;//Point the previous last node to the new node created.
   }
   return head;
}
```

You can add a node at the front, the end or anywhere in the linked list.

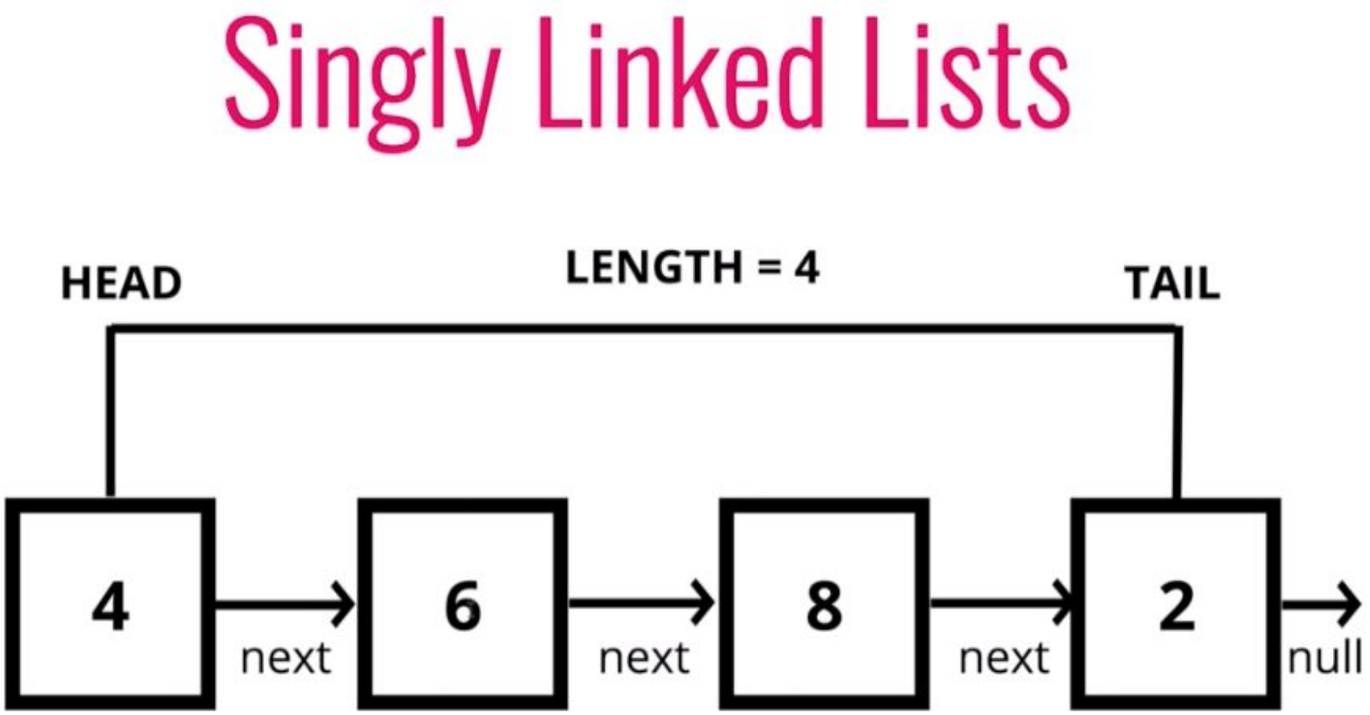
The worst case ***Time Complexity*** for performing these operations is as follows:

- Add item to the front of the list: O(1)
- Add item to the end of the list: O(n)
- Add item to anywhere in the list: O(n)
-

You can remove a node either from the front, the end or from anywhere in the list.

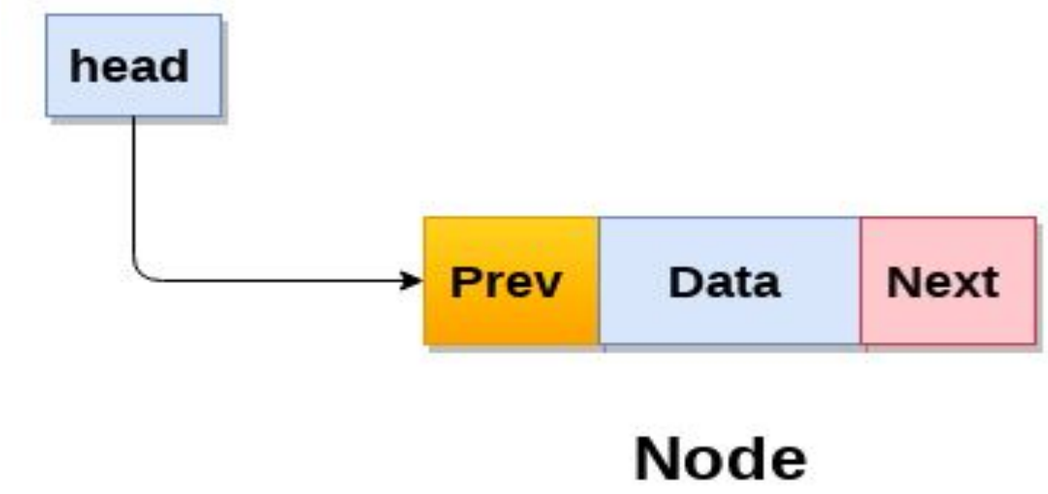
The worst case ***Time Complexity*** for performing this operation is as follows:

- Remove item from the front of the list: O(1)



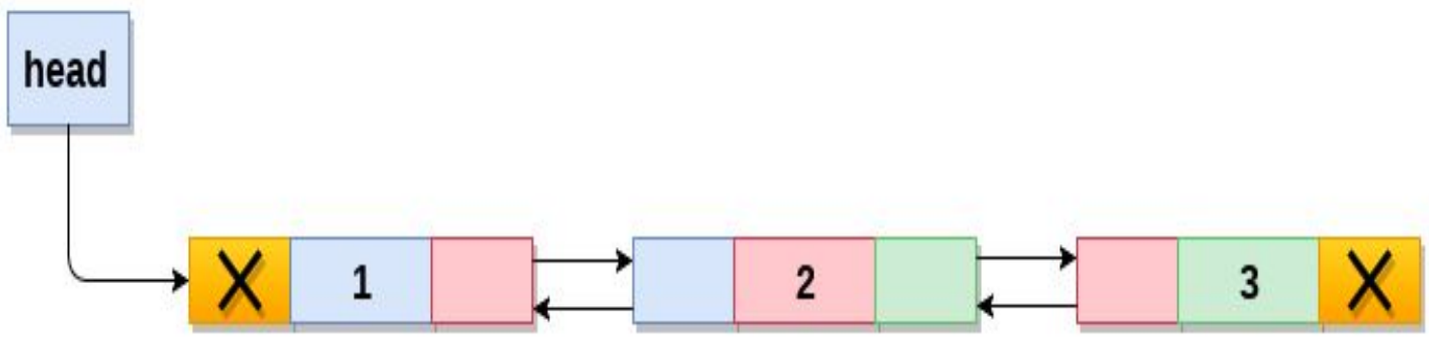
2.3.1.2 Double linked-list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

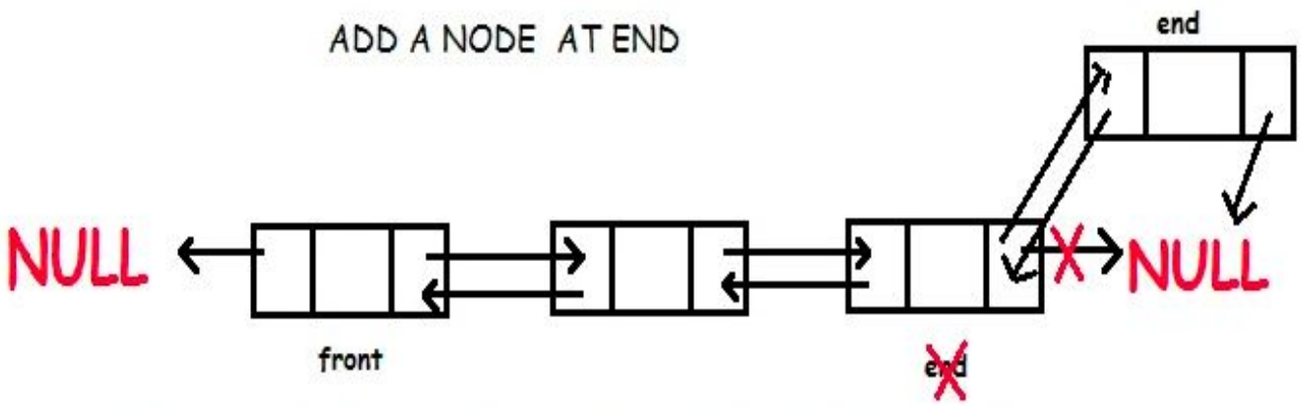


Struct Code :-

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
}
```

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

1. The next pointer of last node will always be NULL and prev will point to end.
2. If the node is inserted is the first node of the list then we make front and end point to this node.
3. Else we only make end point to this node.



2.4 Comparing sorting algorithms

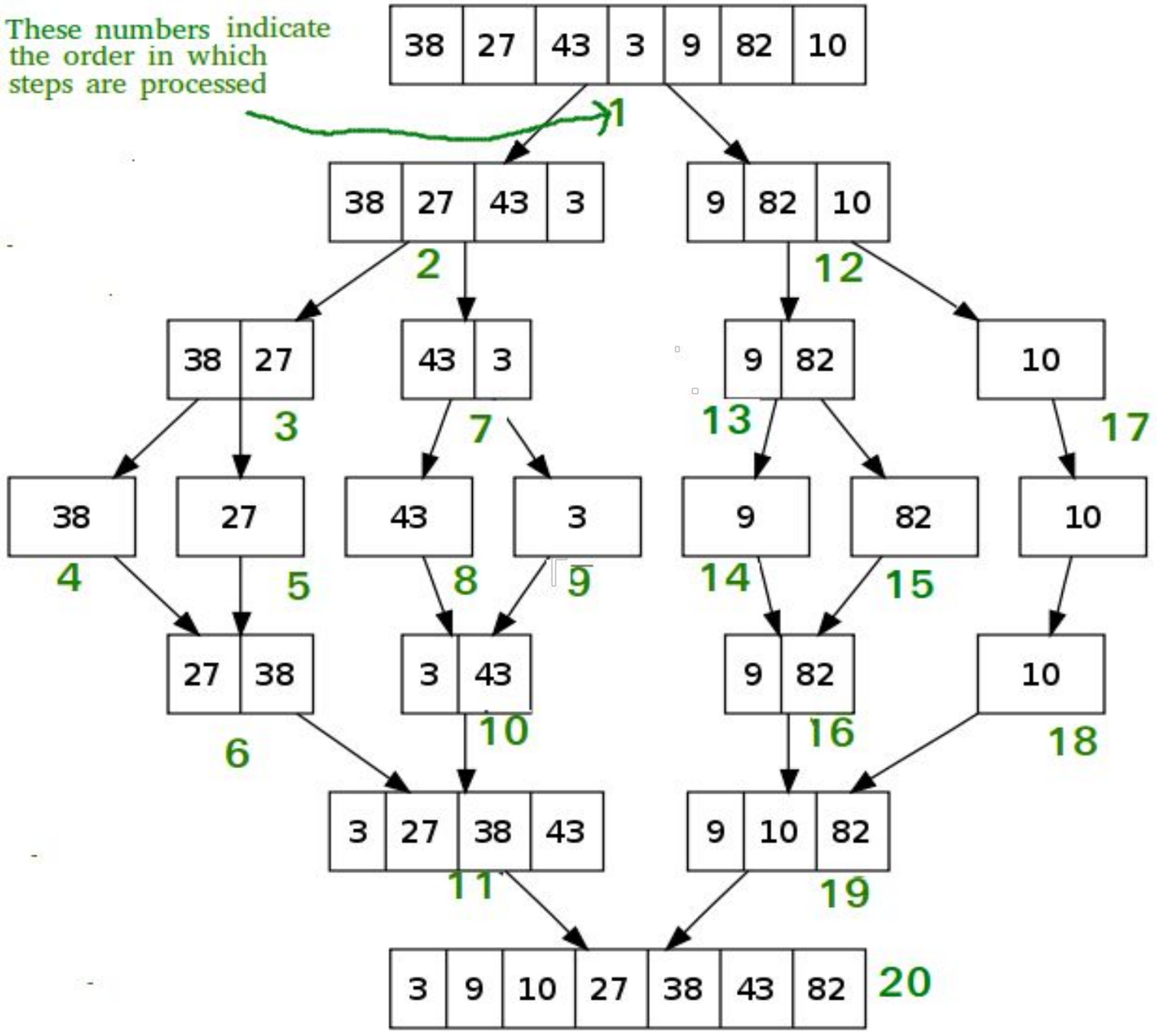
2.4.1 Merge sort

Merge Sort is a Divide and Conqueralgorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

```
void merge(int a[], int beg, int mid, int end)
{
   int i, j, k;
   int n1 = mid - beg + 1;
   int n2 = end - mid;

   int LeftArray[n1], RightArray[n2];

   for (int i = 0; i < n1; i++)
   LeftArray[i] = a[beg + i];
   for (int j = 0; j < n2; j++)
   RightArray[j] = a[mid + 1 + j];

   i = 0;
   j = 0;
   k = beg;
```

```
    while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}
```

Pictorial representation of merge sort algorithm



Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \theta(n)$

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

Stable: Yes

Drawbacks of merge sort:-

- Slower comparative to the other sort algorithms for smaller tasks.
- Merge sort algorithm requires an additional memory space of 0(n) for the temporary array.
- It goes through the whole process even if the array is sorted.

2.4.3 Bubble sort

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is **$O(n^2)$,** where **n** is a number of items.

VIDEO EXPLANATION :- Bubble Sort

CODE:-

```c
#include<stdio.h>
 void print(int a[], int n) //function to print array elements
   {
    int i;
    for(i = 0; i < n; i++)
    {
       printf("%d ",a[i]);
    }
    }
 void bubble(int a[], int n) // function to implement bubble sort
 {
   int i, j, temp;
   for(i = 0; i < n; i++)
    {
      for(j = i+1; j < n; j++)
        {
           if(a[j] < a[i])
           {
              temp = a[i];
              a[i] = a[j];
              a[j] = temp;
           }
        }
    }
}
```

Time complexity:-

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Space complexity:-

Space Complexity	O(1)
Stable	YES

The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1)

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

2.4.3 Quick sort

Sorting is a way of arranging items in a systematic manner. 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.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

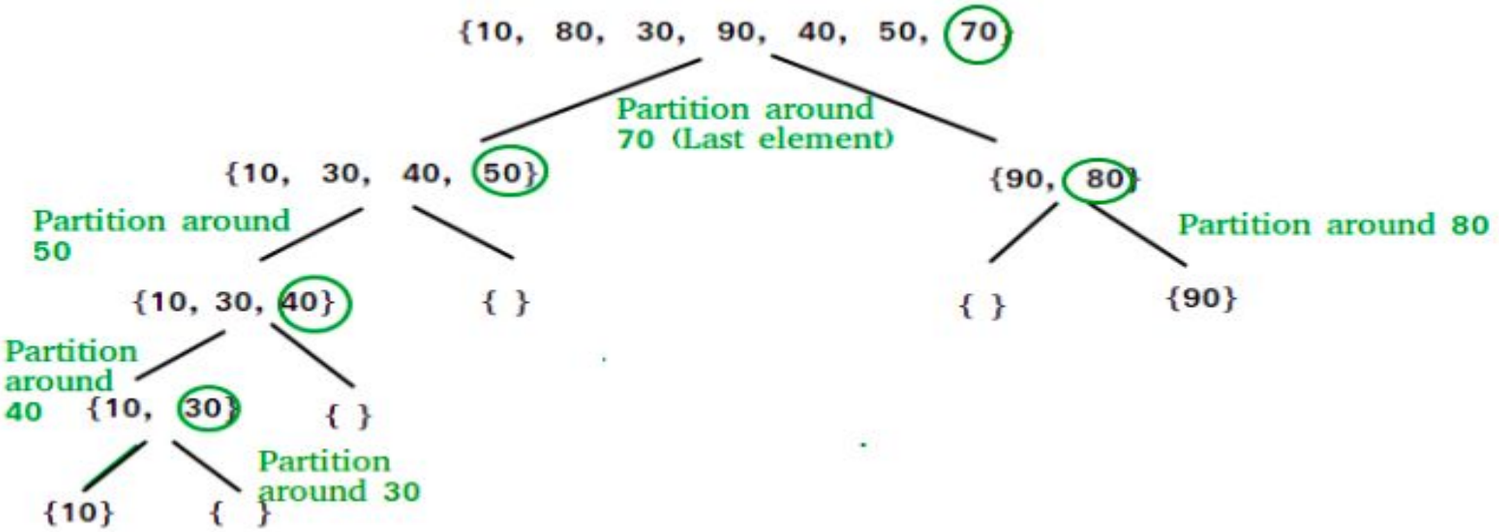
Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

CHOOSING PIVOT :-

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

```c
#include <stdio.h>
int partition (int a[], int start, int end)
{
    int pivot = a[end];
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {
        if (a[j] < pivot)
        {
            i++;
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
    return (i + 1);
}
void quick(int a[], int start, int end)
{
    if (start < end)
    {
        int p = partition(a, start, end);
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}
```

PICTORIAL EXAMPLE OF QUICK SORT



Time complexity:-

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

Space complexity :-

Space Complexity	O(n*logn)
Stable	NO

The space complexity of quicksort is O(n*logn).

2.4 Binary search

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty

Working:-

Given an array:-


```
mid = low + (high - low) / 2
```

```
Applying this formula
Mid = (9-0)/2
Mid = 4
```

Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
```

```
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.

We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Pseodocode :-

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1,x);
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}
```

Space complexity :-

The space complexity of the binary search algorithm depends on the implementation of the algorithm. There are two ways of implementing it:

1. Iterative method
2. Recursive method

Both methods are quite the same, with two differences in implementation. First, there is no loop in the recursive method. Second, rather than passing the new values to the next iteration of the loop, it passes them to the next recursion. In the iterative method, the iterations can be controlled through the looping conditions, while in the recursive method, the maximum and minimum are used as the boundary condition.

In the iterative method, the space complexity would be O(1). While in the recursive method, the space complexity would be O(log n).

Time complexity :-

The time complexity of the binary search algorithm is O(log n). The best-case time complexity would be O(1) when the central index would directly match the desired value. The worst-case scenario could be the values at either extremity of the list or values not in the list.

3. Introduction to graph

3.1 types of graph algorithms

3.1.1 Breadth first search :-

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

3.1.2 Depth First search :-

The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

3.1.3 Dijkstra's Algorithm :-

Dijkstra's algorithm is also known as the shortest path algorithm. It is an algorithm used to find the shortest path between nodes of the graph. The algorithm creates the tree of the shortest paths from the starting source vertex from all other points in the graph. It differs from the minimum spanning tree as the shortest distance between two vertices may not be included in all the vertices of the graph. The algorithm works by building a set of nodes that have a minimum distance from the source. Here, Dijkstra's algorithm uses a greedy approach to solve the problem and find the best solution.

4 Breadth First Search :-

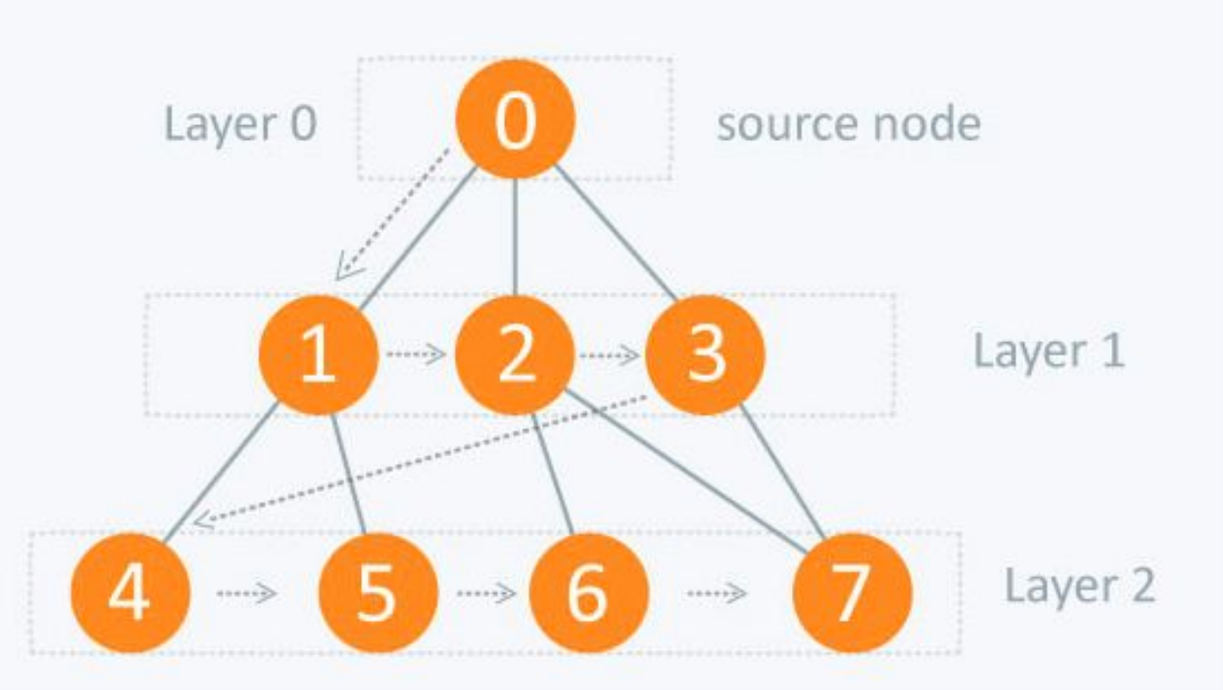
BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

First move horizontally and visit all the nodes of the current layer

Move to the next layer

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer



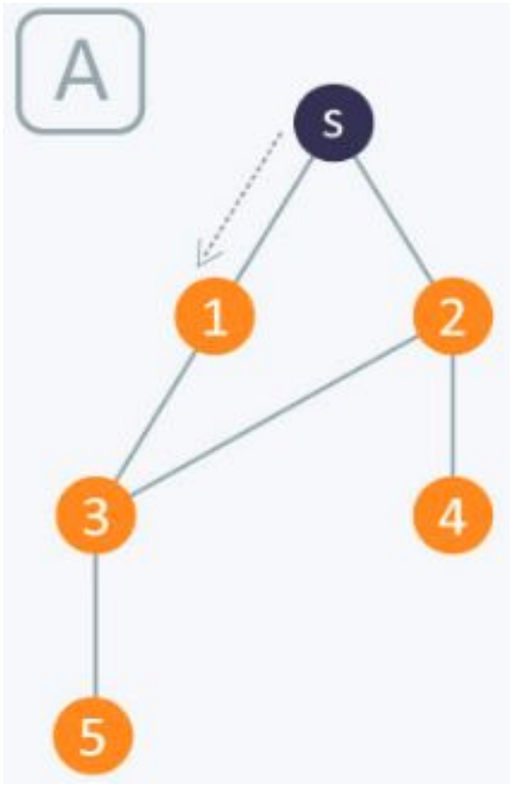
The distance between the nodes in layer 1 is comparitively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

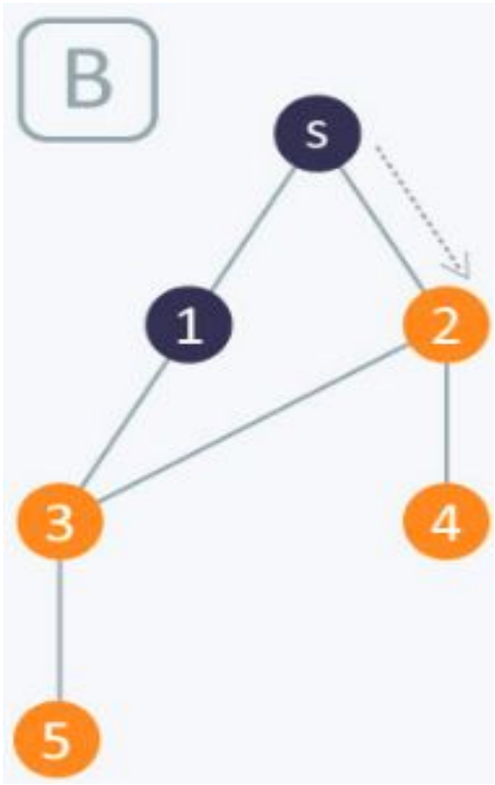
In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

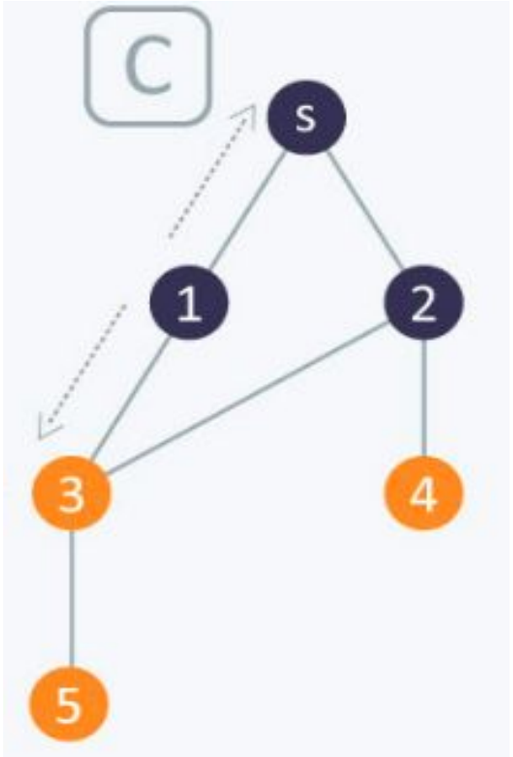
To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neigbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Traversing Node :-

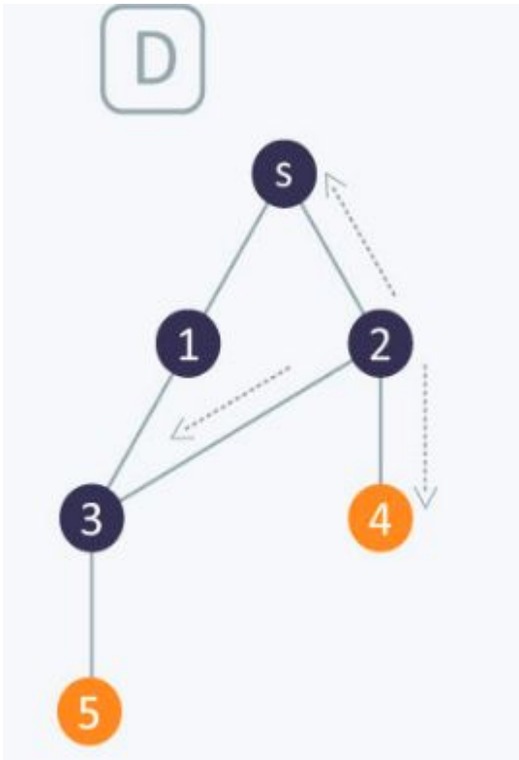


==========>

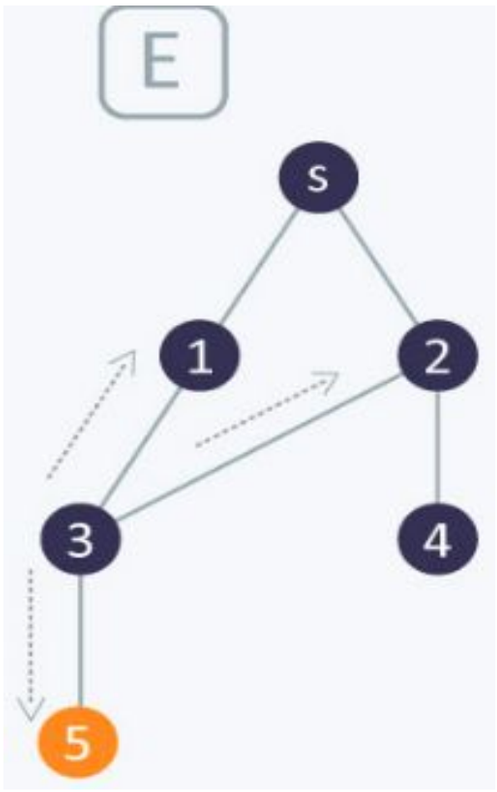




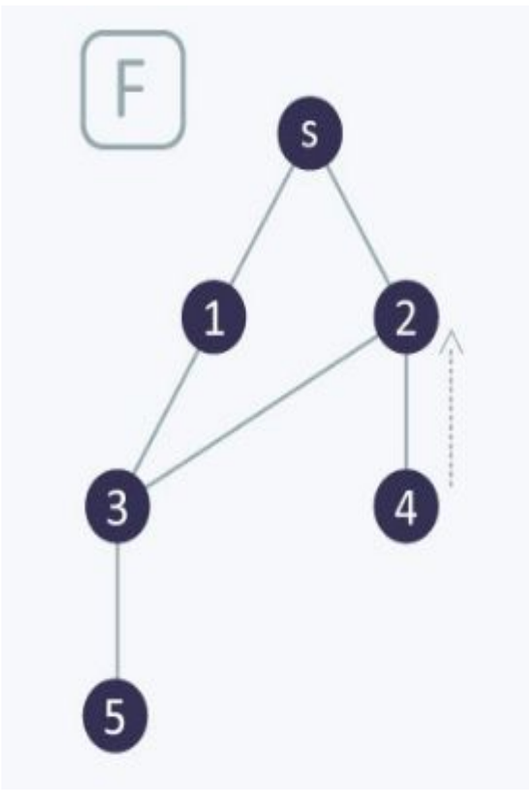
==========>

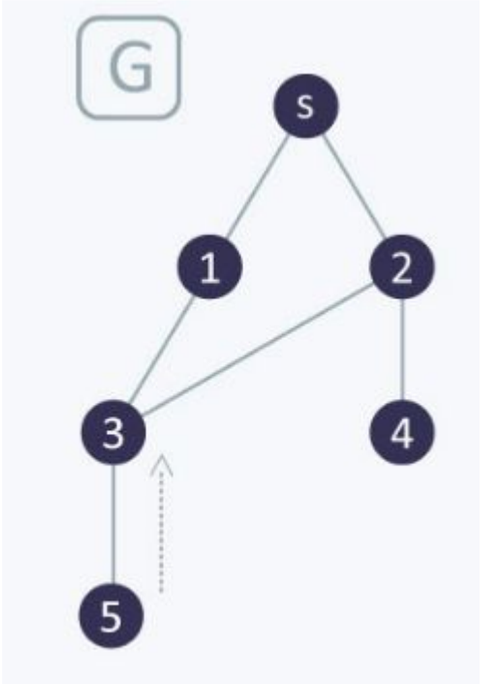


========


E
s
1
2
3
4
5

==========>


F
s
1
2
3
4
5


G
s
1
2
3
4
5

First iteration

- s will be popped from the queue
- Neighbors of s i.e. 1 and 2 will be traversed
- 1 and 2, which have not been traversed earlier, are traversed. They will be:
    - Pushed in the queue
    - 1 and 2 will be marked as visited

Second iteration

- 1 is popped from the queue
- Neighbors of 1 i.e. s and 3 are traversed
- s is ignored because it is marked as 'visited'
- 3, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited

Third iteration

- 2 is popped from the queue
- Neighbors of 2 i.e. s, 3, and 4 are traversed
- 3 and s are ignored because they are marked as 'visited'
- 4, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited

Fourth iteration

- 3 is popped from the queue
- Neighbors of 3 i.e. 1, 2, and 5 are traversed
- 1 and 2 are ignored because they are marked as 'visited'
- 5, which has not been traversed earlier, is traversed. It is:
    - Pushed in the queue
    - Marked as visited

Fifth iteration

- 4 will be popped from the queue
- Neighbors of 4 i.e. 2 is traversed
- 2 is ignored because it is already marked as 'visited'

Sixth iteration

- 5 is popped from the queue
- Neighbors of 5 i.e. 3 is traversed
- 3 is ignored because it is already marked as 'visited'

Complexity

The time complexity of BFS is O(V + E), where V is the number of nodes and E is the number of edges.

Depth First search:-

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

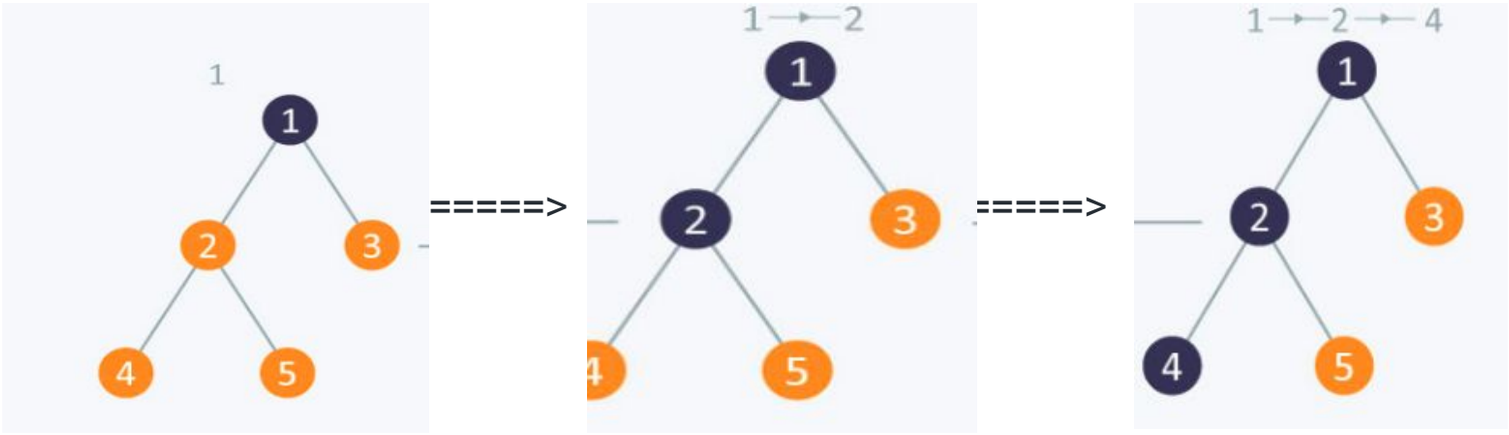
Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

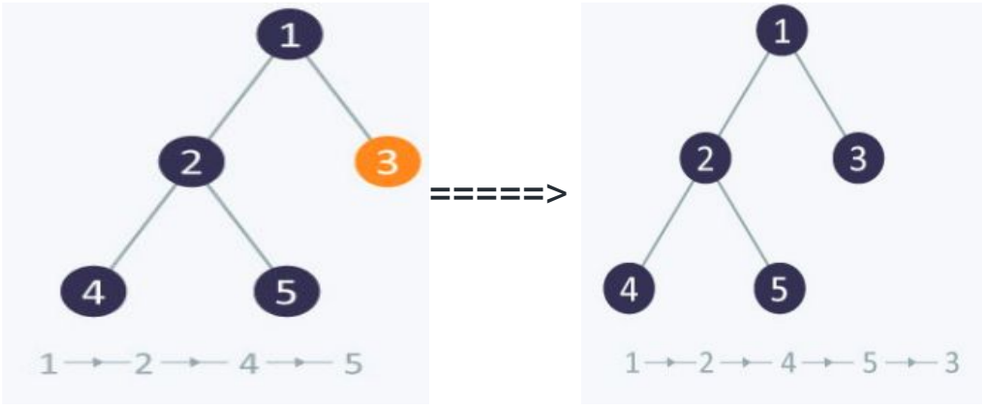
This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.





Let 1 be the staring node

First iteration

- We are at 1.
- Then node left to is is traversed, i.e. 2 is reached.
- 2 is pushed to stack.

Second iteration

- Now we are at 2.
- Node left to is visited, i.e. 4.
- Now 4 is pushed to stack.

Third Iteration.

- Now we are at 4.
- Now there is no node present to left of it, so now the next element is popped from stack i.e 2.
- Now node right of it is 5 is reached.
- Now 2 is pushed to stack.

Fourth iteration

- Now we are at 5 and no node is present below it.
- Now 2 is popped from stack and has been already visited.
- Now 1 is popped from stack.
- Now right of it i.e. 3 is reached.

Time complexity

The time complexity of `DFS` if the entire tree is traversed is

O(V)

$O(V)$ where V is the number of nodes. In the case of a graph, the time complexity is

O(V + E)

$O(V+E)$ where V is the number of vertexes and E is the number of edges.

8. Dijkstra Algorithm or greedy algorithm 7

Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.
Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Short working o this algorithm :-

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

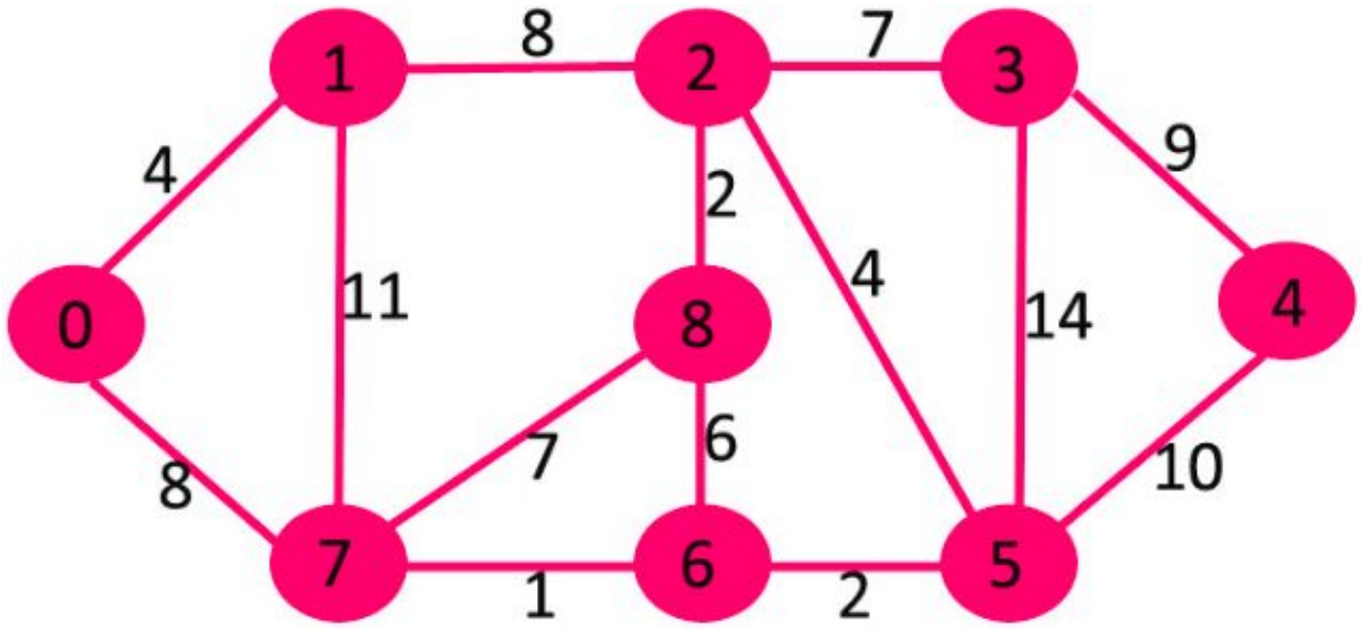
3) While sptSet doesn't include all vertices

….a) Pick a vertex u which is not there in sptSet and has a minimum distance value.

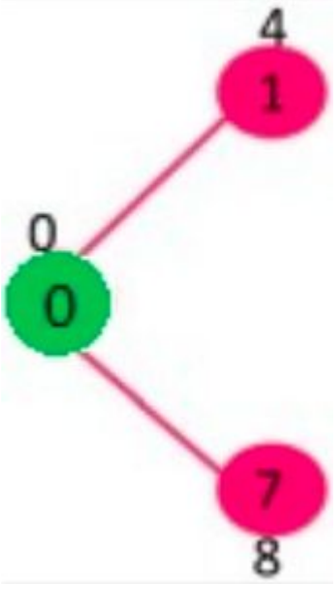
….b) Include u to sptSet.

….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

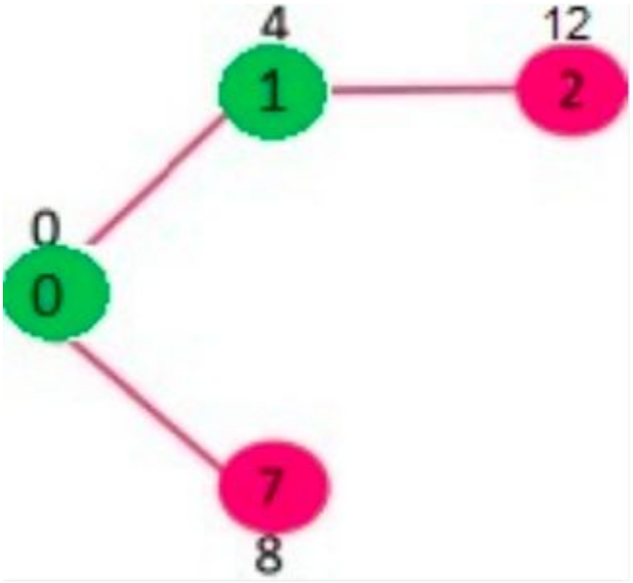
Example:-



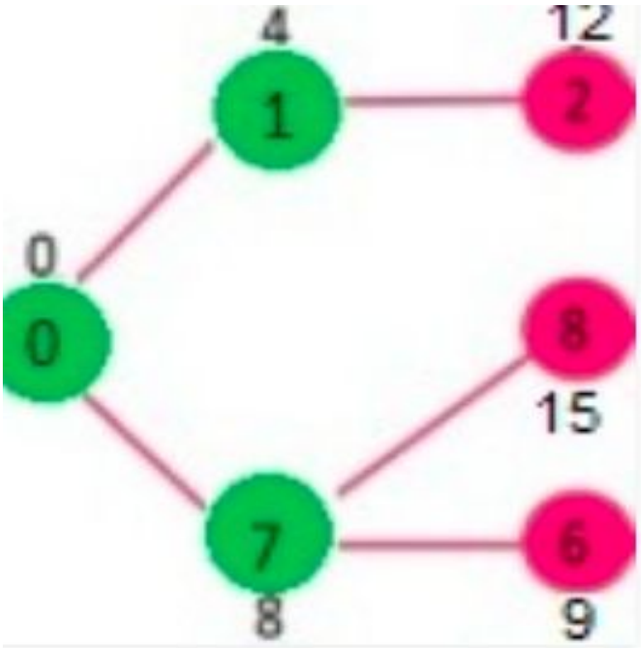
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



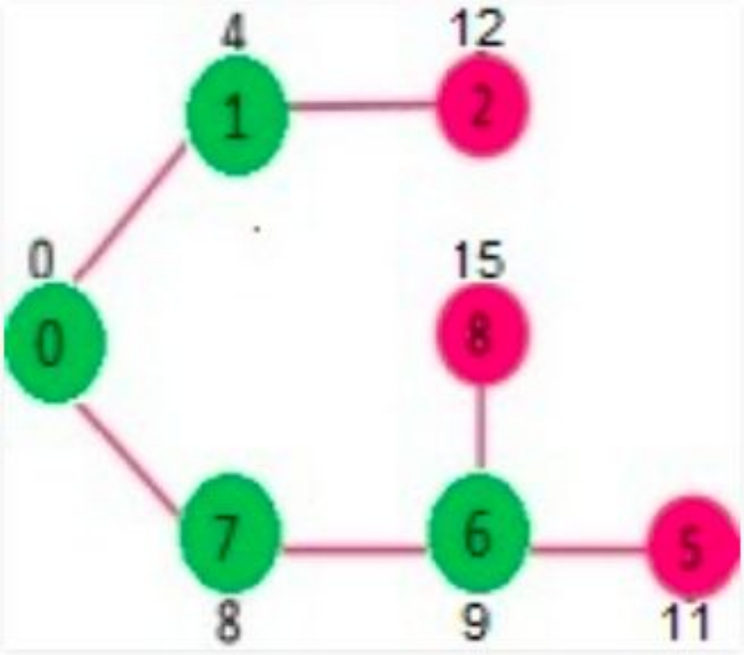
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



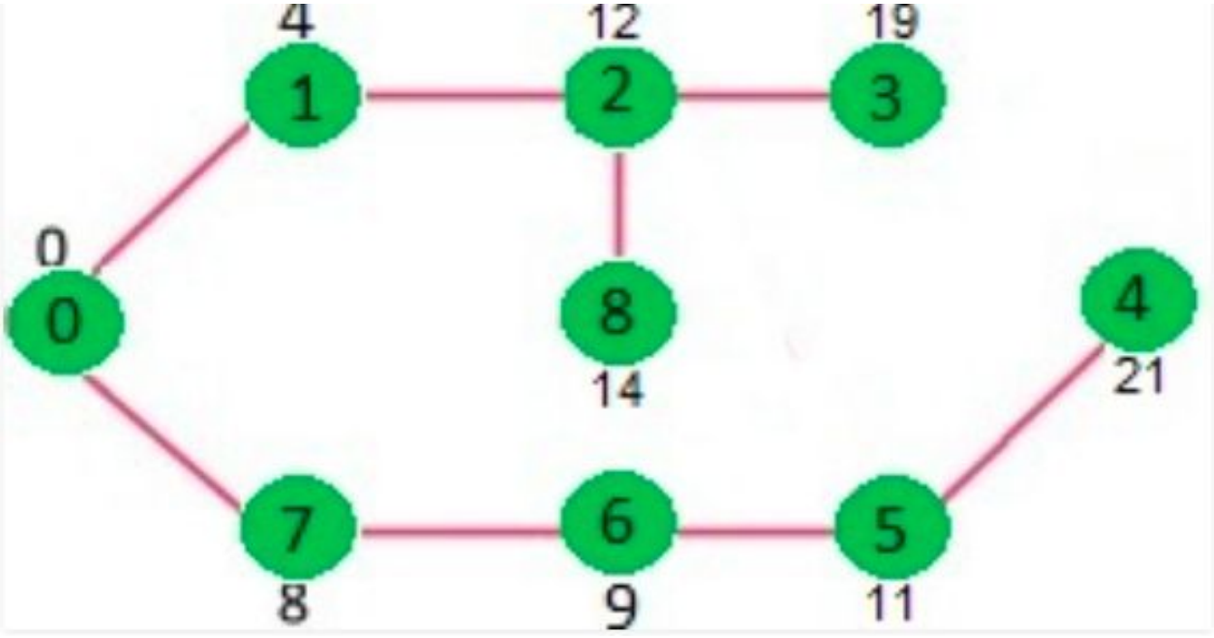
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).



Case-01:

This case is valid when-

- The given graph G is represented as an adjacency matrix.
- Priority queue Q is represented as an unordered list.

Here,

- A[i,j] stores the information about edge (i,j).
- Time taken for selecting i with the smallest dist is O(V).
- For each neighbor of i, time taken for updating dist[j] is O(1) and there will be maximum V neighbors.
- Time taken for each iteration of the loop is O(V) and one vertex is deleted from Q.
- Thus, total time complexity becomes $O(V^2)$.

Case-02:

This case is valid when-

- The given graph G is represented as an adjacency list.
- Priority queue Q is represented as a binary heap.

Here,

- With adjacency list representation, all vertices of the graph can be traversed using BFS in O(V+E) time.
- In min heap, operations like extract-min and decrease-key value takes O(logV) time.
- So, overall time complexity becomes O(E+V) x O(logV) which is O((E + V) x logV) = O(ElogV)
- This time complexity can be reduced to O(E+VlogV) using Fibonacci heap.

References:-

https://www.javatpoint.com/singly-linked-list

https://www.javatpoint.com/doubly-linked-list

https://www.geeksforgeeks.org/merge-sort/

https://www.geeksforgeeks.org/bubble-sort/

https://www.geeksforgeeks.org/quick-sort/

https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/