SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(An Autonomous Institution)
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution &
Accredited by NAAC with "A" Grade)
Madagadipet, Puducherry - 605 107

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject Name: **Data Structures**
Subject Code: **U20EST356**

Prepared by:

Verified by: **Approved by:**

UNIT – V

Sorting, Hashing and Graphs

Sorting: Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Shell Sort and Radix Sort. Performance and Comparison among the sorting methods. Hashing: Hash Table, Hash Function and its characteristics. Graph: Basic Terminologies and Representations, Graph traversal algorithms

2 Marks

1. **Define Sorting**.

   A sorting algorithm is an algorithm that puts elements of a list either in ascending or descending order.

2. **What are the different types of sorting techniques? Give example.**

   There are 2 sorting techniques:

   - Internal sorting eg: sorting with main memory
   - External sorting eg: sorting with disk, tapes.

3. **Compare internal and external sorting**.

Internal sorting:

- It takes place in the main memory of a computer
- Internal sorting methods are applied for small collections of data (i.e) the entire collection of data to be sorted is small enough that the sorting can take place within main memory.

External sorting:

- Externalsortingmethodsareappliedonlywhenthenumberofdataelementstobesortedistoolarge.
- These sorting methods involves as much external processing.

4. List out some of the sorting algorithms.

1. Insertion sort
2. Selectionsort
3. Bubble sort
4. Shell sort
5. Merge sort
6. Heap sort
7. Quick sort
8. RadixSort

5. Write down the limitations of Insertion sort.

- It is less efficient on list containing more number of elements.
- As thenumber of elementsincreasestheperformanceoftheprogramwould be slow.
- Insertion sort needs a large number of element shifts.

6. What are the basic operations of Insertion sort?

Westartwithwithemptylist,„S‟andunsorted list,„I‟of,„n‟items. For (each item „X‟ of „I‟)

{

We are going to insert „X‟ into „S‟, in sorted order.

}

7. Name the sorting techniques which use Divide and Conquer strategy.

- Merge sort
- Quick sort

8. What is the best case and average case analysis for Quick sort?

- The total time in best case is : O(n logn)
- The total time in worst case is : $\Theta(n^2)$

9. What is the complexity of bubble sort?

- Best case performance: O(n)
- Average case performance: $O(n^2)$
- Worst case performance: $O(n^2)$

10. Compare bubble and Insertion sort.

- Even though both the bubble sort and insertion sort algorithms have average case time Complexities of $O(n^2)$,bubble sort is outperformed bythe insertionsort(i.e)insertion sort isfasterthanbubble sort.

- Thisisduetothenumberofswapsneededbythetwoalgorithms(bubblesortsneeds moreswaps).
- But due to the simplicity of bubble sort, its code size is very small.
- insertionsortisveryefficientforsorting“nearlysorted”lists,whencomparedwiththebubblesort.

11. Explain the concept of merge sort.

- Divide the list in half
- Merger sort the first half
- Merge sort the second half
- Merge both halves back together in sorted order.

12. What is radix sort?

- The Radix Sort performsorting, by using bucket sorting technique.
- InRadixSort,first,bucketsortisperformedbyleastsignificantdigit,thennextdigit andsoon.Itis sometime known as Card Sort.

13. What is Bubble Sort and Quick sort?

Bubble Sort: Thesimplestsortingalgorithm. It involvesthesortingthelist in arepetitivefashion. It compares two adjacent elements in the list, and swaps them if they are not in the designated order. It continues until there are no swaps needed. This is the signal for the list that is sorted. It is also called as comparison sort as it uses comparisons.

Quick Sort: Thebestsortingalgorithmwhichimplementsthe‘divideandconquer’concept.Itfirstdivides the list into two parts by picking an element a ’pivot’. It then arranges the elements those are smaller than pivotintoonesublistandtheelementsthoseareгreaterthanpivotintoonesublistbykeepingthepivotin its originalplace.

14. What is the main idea behind the selection sort

Theideaofselectionsortisrathersimplewhichrepeatedlyfindthenextlargest(orsmallest)elementin the array and move it to its final position in the sorted array.

15. What is the main idea in Bubble sort?

The basic idea underlying the bubble sort is to pass through the file sequentially several times. Each pass consists of comparing each element in the file with its successor (x[i] and x[i+1] and interchangingthe twoelements if they are not in proper order.

16. When can we use insertion sort?

Insertion sort is useful only for small files or very nearly sorted files.

17. What is the main idea behind insertion sort?

The main idea of insertion sort is to insert in the ith pass the ith element in A (1) A (2)...A (i) in its right

place.Aninsertionsortisonethatsortsasetofrecordsbyinsertingrecordsintoanexistingfile.

18. Justify that the selection sort is diminishing increment sort.(Nov 2012)

Selection sort is diminishing increment sort. Because the Number of swapping in selection sort is better than Insertion sort.

19. Define the term sorting.

The term sorting means arranging the elements of the array so that they are placed in some relevant order which may either be ascending order or descending order.

That is, if A is an array then the elements of A are arranged in sorted order (ascending order) in such a way that, A [0] < A [1] < A [2] < …… < A [N].

20. What is sorting algorithm?

A sorting algorithm is defined as an algorithm that puts elements of a list in a certain order that can either be numerical order, lexicographical order or any user-defined order.

21. Where do we use external sorting?

External sorting is applied when there is huge data that cannot be stored in computer's memory.

22. Where do we use external sorting?

External sorting is applied when there is huge data that cannot be stored in computer's memory.

23. List out the different sorting techniques.

- Insertion sort
- Selection sort
- Shell sort
- Bubble sort
- Quick sort
- Merge sort
- Radix sort

24. What is the advantage of quick sort?

Quick sort reduces unnecessary swaps and moves an item to a greater distance, in one move.

25. Define Hashing.

Hashing is used for storing relatively large amounts of data in a table called a hash table. Hashing is a technique used to perform insertions, deletions, and finds the element in **constant average time**

26. What do you mean by hash table?

Hash Table is a data structure in which keys are mapped to array positions by a hash function. Hash table is usually fixed as M-size, which is larger than the amount of data that we want to store.

27. Define hash function.

Hash function is a mathematical formula, produces an integer which can be used as an index for the key in the hash table.

- **Perfect Hash Function**

  Each key is transformed into a unique storage location

- **Imperfect hash Function**

Maps more than one key to the same storage location .

28. What are the different methods of hash function?

- Division Method
- Multiplication Method
- Mid Square Method
- Folding Method

29. Define Division Method

Division method is the most simple method of hashing an integer *x*. The method divides *x* by *M* and then use the remainder thus obtained.

In this case, the hash function can be given as

$$h(x) = x \bmod M$$

30. When collision does occur?

Collision occurs when the hash function maps two different keys to same location.

31. What are the various collision resolution techniques used?

Two major classes of collision resolution

Open Addressing

- When collision occurs, use organized method to find next open space.
- Maximum number of elements equal to table size.

Chained Addressing – Separate Chaining

- Make linked list of all element that hash to same location
- Allows number of elements to exceed table size

32. What are the methods used in Open addressing collision technique?

- Linear Probing
- Quadratic Probing
- Double Hashing

33. What is mean by rehashing?

If the table gets too full, the insertion might fail .A solution is to build another table that is almost twice as big & scan down the entire original table into new table.

34. When do we perform rehashing?

- Rehash as soon as table is half full.
- Rehash when insertion fails
- When table reaches certain load factor
- Performance degrades as the load factor increases

35. When do we use extendible hashing?

When the amount of data is too large to fit in main memory, the previous hashing techniques will not work properly. So we use a new technique called extensible **hashing**. It is one form of dynamic hashing.

15,25,5,40,2,70,18 – Insertion sort

5,15,30,6,3,95- Merge sort

36. What is radix sort?

- The Radix Sort perform sorting, by using bucket sorting technique.
- In Radix Sort, the numbers are sorted on the least significant digit first, followed by secondleast
  significant digit and so on till the most significant digit. It is sometime called as Card Sort.

37. What is bubble sort?

- Bubble sort is otherwise called as sinking sorts.
- The idea of bubble sort is to move the highest element to n**th** position.
- The principle of bubble sort is to scan or read the array in (n-1) times.
- It compares two adjacent elements in the list and swaps them if they are not in the designated order. It continues until there are no swaps needed.
- It is also called as comparison sort, as it uses comparisons.

38. What is quick sort?

- The best sorting algorithm which implements the 'divide and conquer' concept.
- It first divides the list into two parts by picking an element a 'pivot'.
- Then arranges the elements those are smaller than pivot into one sub list and the elements those are greater than pivot into one sub list by keeping the pivot in its original place.
- It is also called as partition exchange sort.

39. What is selection sort ? Why selection sort is better than insertion sort?

The idea of selection sort is rather simple which repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array.

40. What is insertion sort ?

- In insertion sort the elements are inserted at an appropriate place similar to card insertion.
- The elements in the list is divided into two parts- sorted and unsorted sub-lists.

- In each pass, the first element of unsorted sub-list is picked up and moved into the sorted sub-list by inserting it in suitable positon.

41. What is shell sort?

The shell sort improves upon bubble sort and insertion sort, by moving out of order elements more than one position at a time.

It compares the elements that are at a specific distance from each other, and interchanges them if necessary. The shell sort divides the list into smaller sub lists, and then sorts the sub lists separately using the insertion sort

42. What is Selection sort?

Selection sort is sorted by scanning the entire list to find the smallest element and exchange it with the first element, putting the first element in the final position in the sorted list. Then the scan starts from the second element to find the smallest among n-1 elements and exchange it with the second element.

43. What is Merge sort?

- It follows divide and conquer method for its operation.
- In Dividing phase, the problem is divided into smaller problem and solved recursively.
- In conquering phase, the partitioned array is merged together recursively.
- Merge sort is applied to first half and second half of the array.
- It gives two sorted halves which can then be recursively merged together using the merging algorithm.

44. What is the performance analysis of merge sort?

- Best case Performance: O(n log n)
- Average Case Performance: O(n log n)
- Worst case performance: O(n log n)

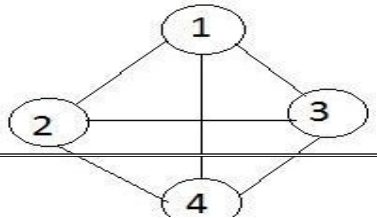
45. Define graph

o A graph consists of two sets V, E.

o V is a finite and non empty set of vertices.

o E is a set of pair of vertices; each pair is called an edge. o V(G),E(G) represents set of vertices ,set of edges .

G= (V,E)



46. Define digraph (Nov 13)

If an edge between any two nodes in a graph is directionally oriented, a graph is called as directed .it is also referred as digraph.

-

47. Define undirected graph.

If an edge between any two nodes in a graph is not directionally oriented, a graph is called as undirected .it is also referred as unqualified graph.

48. Define path in a graph

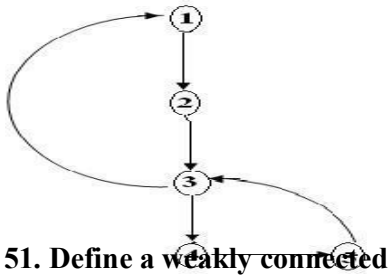
A path in a graph is defined as a sequence of distinct vertices each adjacent to the next except possibly the first vertex and last vertex is different.

49. Define a cycle in a graph

A cycle is a path containing atleast three vertices such that the starting and the ending vertices are the same. The cycles are (1, 2, 3, 1), (1, 2, 3, 4, 5, 3, 1)

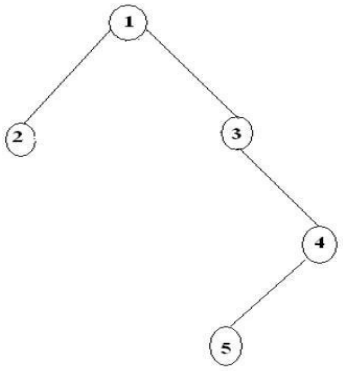
50. Define a strongly connected graph (Nov 14)

A graph is said to be a strongly connected graph, if for every pair of distinct vertices there is a directed path from every vertex to every other vertex. It is also referred as a complete graph.



51. Define a weakly connected graph. (May 14)

A directed graph is said to be a weakly connected graph if any vertex doesn't have a directed path to any other vertices.



52. Define a weighted graph.

A graph is said to be a weighted graph if every edge in the graph is assigned some weight or value. The weight of an edge is a positive value that may be representing the distance between the vertices or the weights of the edges along the path.

-

53. How we can represent the graph?

We can represent the graph by three ways

1.adjacent matrix

2.adjacent list

3. adjacent multi list

54. Define adjacency matrix

Adjacency matrix is a representation used to represent a graph with zeros and ones.

A graph containing n vertices can be represented by a matrix with n rows and n columns.

The matrix is formed by storing 1 n its i^{th} and j^{th} column of the matrix, if there exists an edge between ith and jth vertex of the graph and 0 if there is no edge between i^{th} and j^{th} vertex of the graph .

Adjacency matrix is also referred as incidence matrix.

55. What is meant by traversing a graph? State the different ways of traversing a graph. (Nov 12)

In undirected graph ,G=(V,E)

The vertex V in V(G)

To visit all the vertices that are reached from

the vertex V, that is all the vertices are
connected to vertex V

There are two types of graph traversals

1) Depth first search

2) Breadth first search

56. Define depth first search?

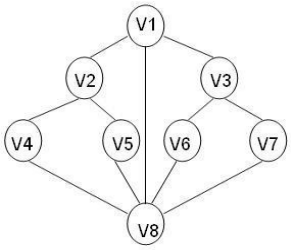
In DFS, we don't have any special vertex, we can start from any vertex.

Let us start from vertex (v), its adjacent vertex is selected and DFS is initialized.

Let us consider the adjacent vertices to V are V1, V2, and V3 …Vk.

Now pick V1 and visit its adjacent vertices then pick V2 and visit the adjacent vertices .to continue and process till the nodes are visited.

Consider the graph



V1	V2, V3, V8
V2	V4, V5
V4	V2, V8
V8	V4, V5, V1, V6, V7

V5	V2, V8
V6	V3, V8
V3	V6, V7, V1
V7	V8, V3

57. Define breadth first search?

Is BFS ,an adjacent vertex is selected then visit its adjacent vertices then backtrack the unvisited adjacent vertex

In BFS ,to visit all the vertices of the start vertex ,then visit the unvisited vertices to those adjacent vertices

58. Define topological sort?

A directed graph *G* in which the vertices represent tasks or activities and the edges represent activities to move from one event to another then the task is known as *activity on vertex network* or AOV-network/pert network

5 Marks

SORTING

1. BUBBLE SORT

1. The easiest and the most widely used sorting technique among students and engineers is the bubble sort.
2. ***This sort is also referred as sinking sort. The idea of bubble sort is to repeatedly move the smallest element to the lowest index position in the list.***
3. To find the smallest element, the bubble sort algorithm begins by comparing the first element of the list with its next element, and upto the end of the list and interchanges the two elements if they are not in proper order.
4. In either case, after such a pass the smaller element will be in the lowest index position of the list.
5. The focus then moves to the next smaller element, and the process is repeated. Swapping occurs only among successive elements in the list and hence only one element will be placed in its sorted order each pass.

BUBBLE (ARR,N)

where ARR is an arrary of N elements

Step 1. REPEAT FOR I=0,1,2….. N-1

Step 2. REPEAT FOR J=I+1 TO N-1

Step 3. IF(ARR[I]>ARR[J] THEN INTERCHANGE ARR[I] AND ARR[J]
(END OF IF STRUCTURE)

Step 4. INCREMENT J BY1

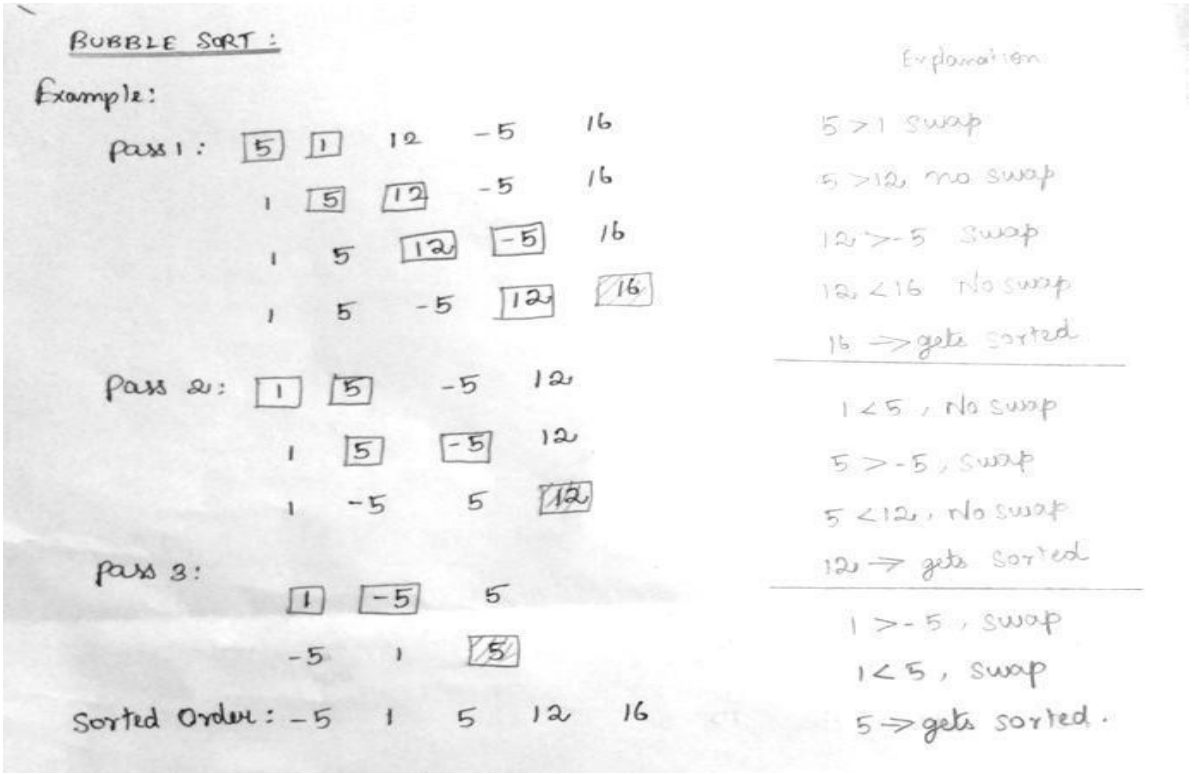
Step 5. (END OF STEP 2 FOR LOOP)

Step 6. (END OF STEP 1 FOR LOOP)

Step 7. PRINT THE SORTED ARRAY ARR

END BUBBLE()

Example :-



//Bubble Sort

#include<stdio.h>

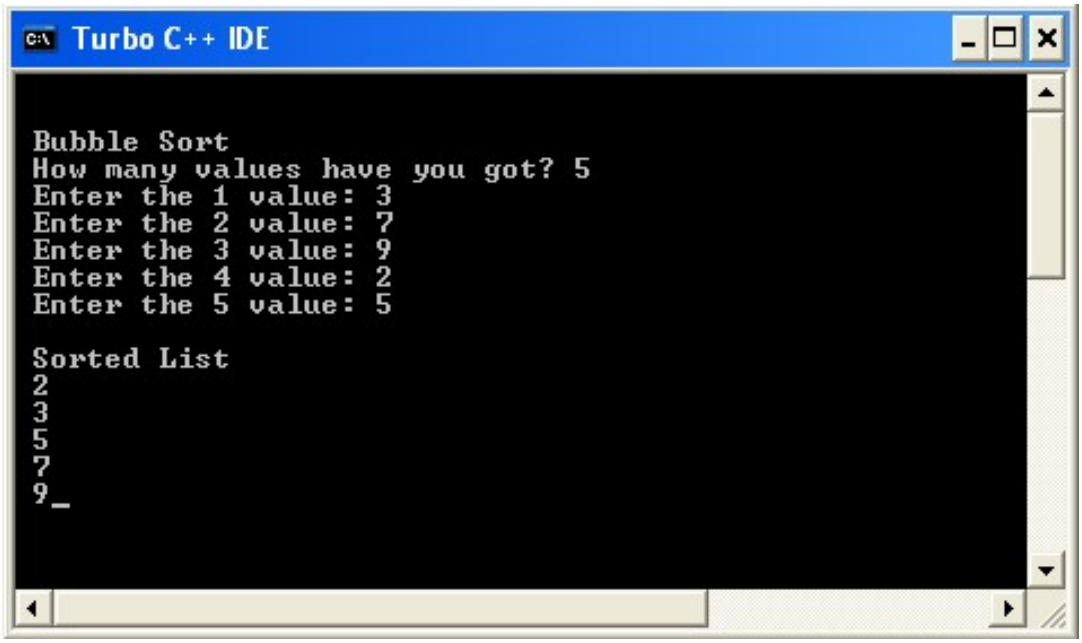
#include<conio.h>

int main()

```
{
 int i,j,temp,n,a[200];
 clrscr();
 printf("\n\n Bubble Sort");
 printf("\n How many values have you got? ");
 scanf("%d",&n);
 for(i=0;i<n;i++)
       {
         printf(" Enter the %d value: ",i+1);
         scanf("%d",&a[i]);
       }
 printf("\n Sorted List");
 for(i=0;i<n;i++)
      {
           for(j=0;j<n-1;j++)
                {
                    if(a[j]>a[j+1])
                        {
                              temp=a[j];
                              a[j]=a[j+1];
                              a[j+1]=temp;
                         }
                }
       }
 for(i=0;i<n;i++)
```

```
  printf("\n %d",a[i]);

 getch();

 return 0;  }
```

OUTPUT:



2. INSERTION SORT

1. The main idea of insertion sort is to consider each element at a time, into the appropriate position relative to the sequence of previously ordered elements, such that the resulting sequence is also ordered.
2. The insertion sort can be easily understood if you know to play cards. Imagine that you are arranging cards after it has been distributed before you in front of the table.
3. As each new cards is taken, it is compared with the cards in hand. The card is inserted in proper place within the cards in hand, by pushing one position to the left or right. This procedure proceeds until all the cards are placed in the hand are in order.

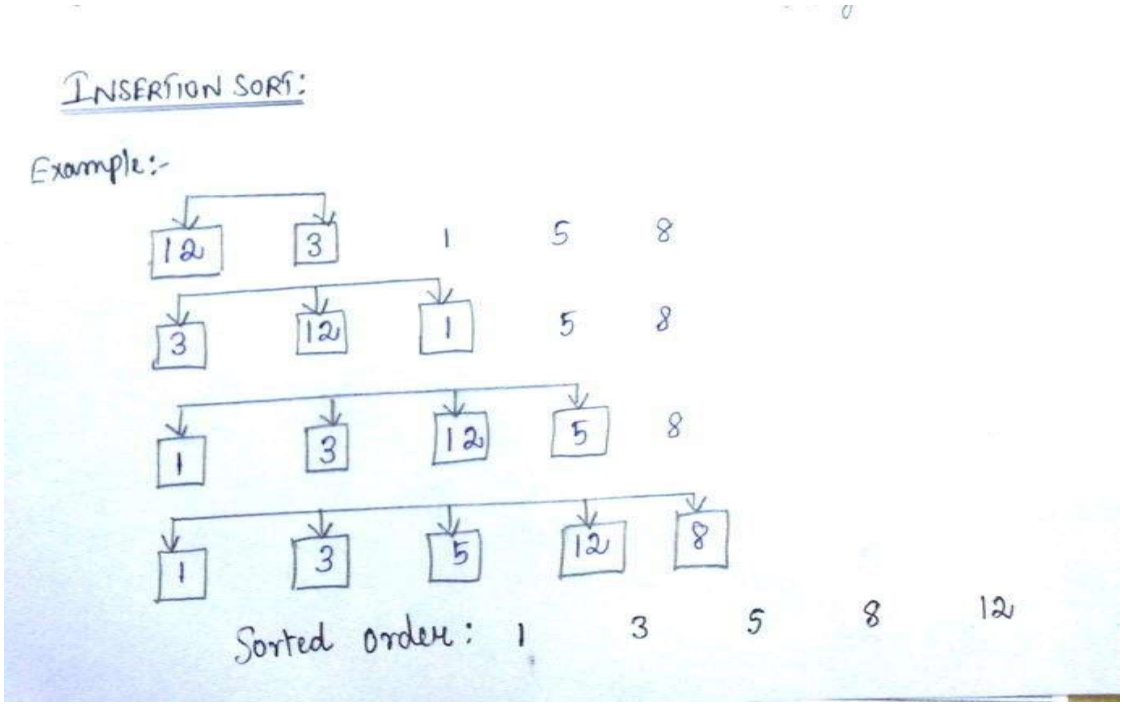
Algorithm

INSERT(ARR,N)

Where ARR is an array of N elements

Step 1. REPEAT FOR I=1,2,3..N+1
Step 2. ASSIGN TEMP=ARR[I] Step
3. REPEAT FOR J=I TO 1 Step 4.
IF(TEMP < ARR[J-1] THEN
ARR[J] = ARR[J-1]
ELSE
GOTO STEP7
(END OF IF STRUCTURE)
Step 5. DECREMENT J BY 1
Step 6. (END OF STEP 3 FOR LOOP)
Step 7. ASSIGN ARR[J] = TEMP Step
8. (END OF STEP 1 FOR LOOP)
Step 9. PRINT THE SORTED ARRAY ARR

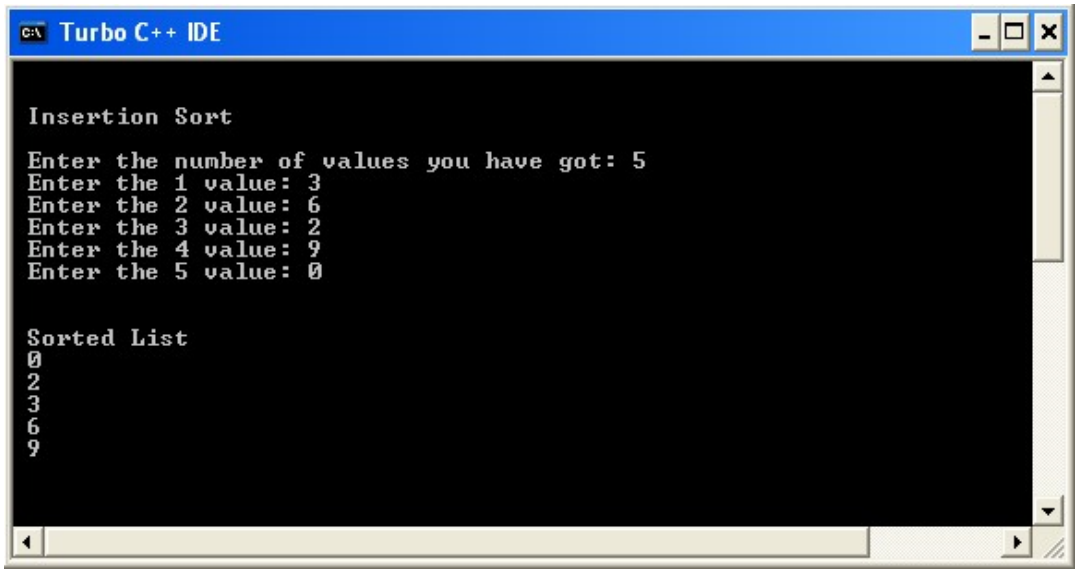
END INSERT()



// INSERTION SORT

```
#include<stdio.h>
#include<conio.h>
int main()
{          int i,j,key,a[200],n;
          clrscr();
          printf("\n\n Insertion Sort");
          printf("\n\n Enter the number of values you have got: ");
          scanf("%d",&n);
          for(i=0;i<n;i++)
                {
                     printf(" Enter the %d value: ",i+1);
                    scanf("%d",&a[i]);
                }
          printf("\n\n Sorted List");
          for(i=1;i<n;i++)
              {
                   key=a[i];
                   j=i-1;
                   while((j>=0)&&(key<a[j]))
                              {
                                   a[j+1]=a[j];
                                    j=j-1;
                              }
                     a[j+1]=key;
              }
          for(i=0;i<n;i++)
                 {
                     printf("\n %d",a[i]);
                 }
         getch();
        return 0;
}
```

OUTPUT:



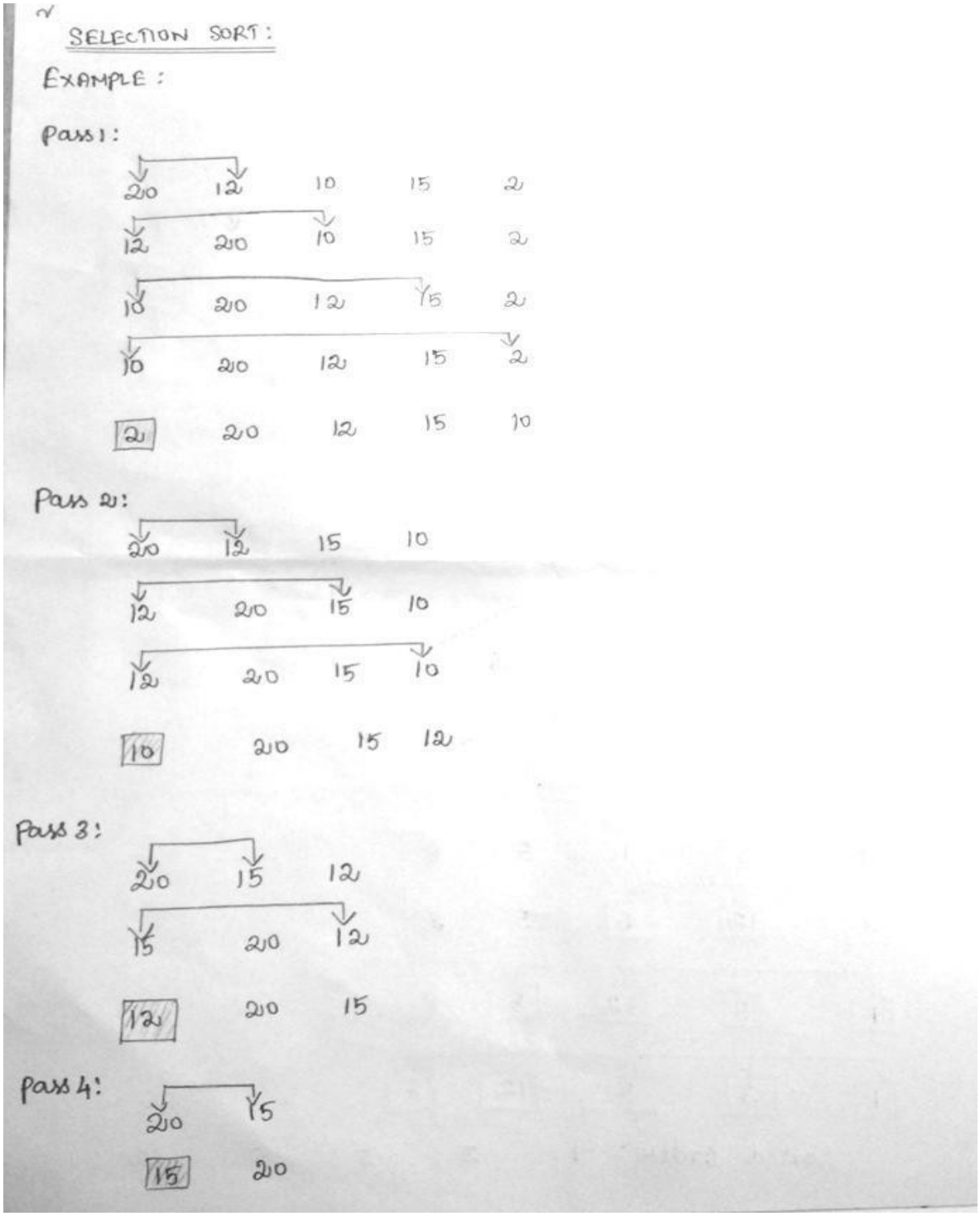
3. SELECTION SORT

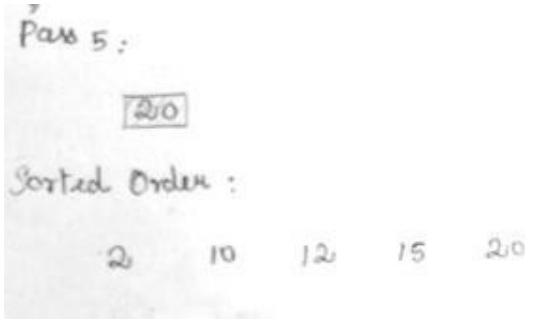
1. Another easiest method for sorting elements in the list is the selection sort. The main idea of selection sort is to search for the smallest element in the list.
2. When the element is found, it is swapped with the first element in the list. The second smallest element in the list is then searched.
3. When the element is found, it is swapped with the second element in the list.
4. The process of searching the next smallest element is repeated, until all the elements in the list have been sorted in ascending order.

Algorithm

```
SELECT(ARR,N)

        WHERE ARR IS AN ARRAY OF N ELEMENTS
        1. REPEAT FOR I=1,2,3...N-1
        2. ASSIGN K=1, MIN =ARR[I]
        3. REPEAT FOR J=I+1 TO N-1
        4. IF ARR[J] <MIN THEN
             MIN=ARR[J]
             K=J
             (END OF IF STRUCTURE)
        5. (END OF STEP 3 FOR LOOP)
        6. ASSIGN ARR[K]=ARR[I], ARR[I]=MIN
        7. (END OF STEP 1 FOR LOOP)
        8. PRINT THE SORTED ARRARY ARR
END SELECT()
```

Example :

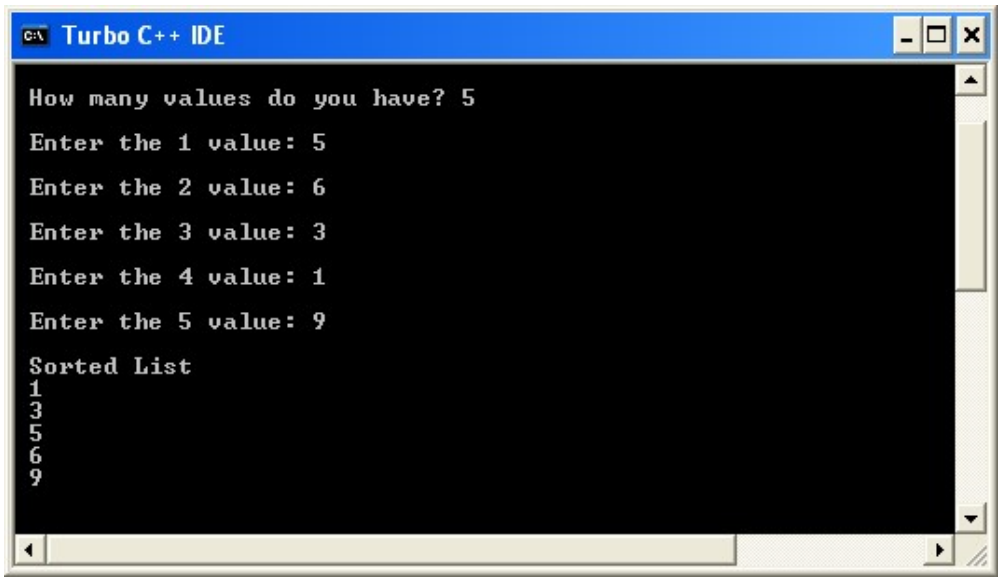

SELECTION SORT:
EXAMPLE:
Pass 1:
20 12 10 15 2
12 20 10 15 2
10 20 12 15 2
10 20 12 15 2
2 20 12 15 10
Pass 2:
20 12 15 10
12 20 15 10
12 20 15 10
10 20 15 12
Pass 3:
20 15 12
15 20 12
12 20 15
Pass 4:
20 15
15 20




```
// Selection Sort
#include<stdio.h>
#include<conio.h>
int main()
{
  int i,j,k,min,n,a[200];
  clrscr();
  printf("\n\n Selection Sort \n ");
  printf("\n How many values do you have? ");
  scanf("%d",&n);
  for(i=0;i<n;i++)
        {
          printf("\n Enter the %d value: ",i+1);
          scanf("%d",&a[i]);
        }
  printf("\n Sorted List");
  for(i=0;i<n;i++)
         {
          k=i;
          min=a[i];
```

```
for(j=i+1;j<n;j++)
                {
                    if(a[j]<min)
                            {
                             min=a[j];
                               k=j;
                            } }
             a[k]=a[i];
            a[i]=min;


            printf("\n %d",a[i]);
         }
  getch();
  return 0;
}
```

OUTPUT:



4.SHELL SORT

1. The shell sort , named after its developer Donald.L.shell in 1959
2. This sort is an extension of the insertion sort, which has the limitation, that it compares only the consecutive elements and interchange the elements by only one space.
3. The smaller elements that are far away require many passes through the sort, to properly insert them in its correct position.
4. The shell sort overcomes this limitation, gains speed than insertion sort, by comparing elements that are at a specific distance from each other, and interchanges them if necessary.
5. The shell sort divides the list into smaller sub lists, and then sorts the sub lists separately using the insertion sort. This is done by considering the input list being n-sorted.
6. This method splits the input list into h-independent sorted files. The procedure of h-sort is insertion sort considering only the h th element (starting anywhere).
7. The value of h will be initially high and its repeatedly decremented until it reaches 1. When h is equal to 1, a regular insertion sort is performed on the list, but by then the list of data is guaranteed to be almost sorted.
8. Using the above procedure for any sequence values of h, always ending in 1 will produce a sorted list.

Algorithm :-

SHELL(ARR,N)

WHERE ARR IS AN ARRAY OF N ELEMENTS.

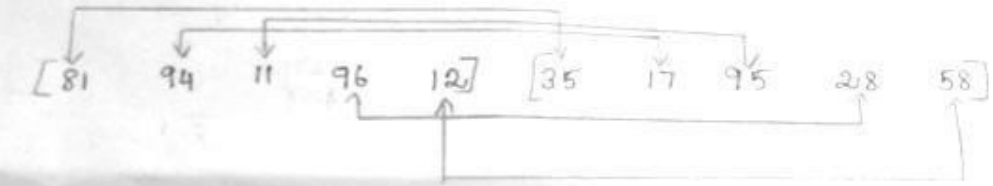
1. REPEAT FOR I=(N+1)/2 TO 1
2. REPEAT FOR J=I TO N-1
3. ASSIGN TEMP=ARR[J] , K=J-1
4. REPEAT WHILE (K>=0 AND TEMP < ARR[K])
5. ASSIGN ARR[K+1] = ARR[K], K=K-1
6. (END OF STEP4 WHILE LOOP)
7. ASSIGN ARR[K+1] =TEMP
8. INCREMENT J BY 1
9. (END OF STEP 2 FOR LOOP)
10. ASSIGN I = I/2
11. (END OF STEP1 FOR LOOP)
12. PRINT THE SORTED ARRAY ARR.

Example :-

SHELL SORT:

81 94 11 96 12 35 17 95 28 58

pass 1:

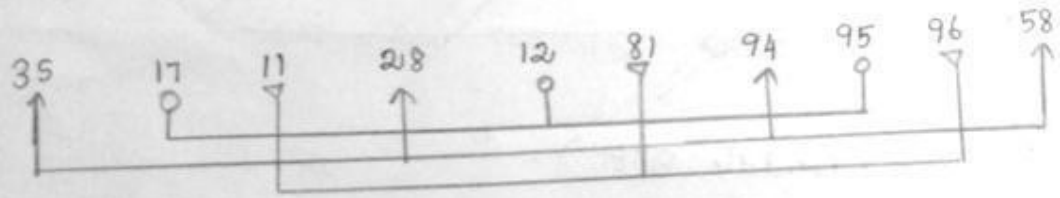


Array 1

$n = 10$

$K = n/2 = 10/2 = 5$ (If K = Prime)

Pass 2:



$K = \frac{K}{2} = \frac{5}{2} = 3$

ss 3:

28 12 11 35 17 81 58 95 96 94

$K = 3/2 = 1.5$

By Performing Insertion Sort, The list is sorted.

IX

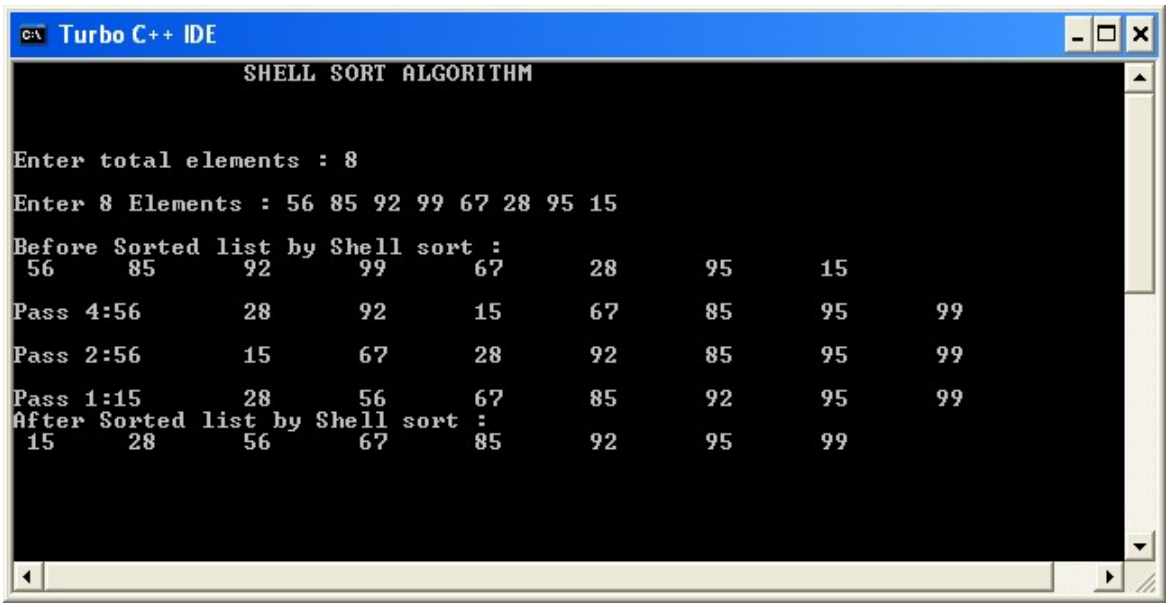
Ans:- 11 12 17 28 35 58 81 94 95 96

//Shell Sort

```
#include<stdio.h>
#include<conio.h>
void shellsort(int a[],int n)
{
  int j,i,k,m,mid;
  for(m = n/2;m>0;m/=2)
  {
        for(j = m;j< n;j++)
        {
            for(i=j-m;i>=0;i-=m)
            {
                if(a[i+m]>=a[i])
                        break;
                else
                {
                 mid = a[i];
                 a[i] = a[i+m];
                 a[i+m] = mid;
                }
            }
        }
    printf("\n\nPass %d:",m);
    for(k=0;k<n;k++)
```

```
            printf("%d\t",a[k]);
    }
}
int main()
{
    int a[20],i,n;
    clrscr();
    printf("\t\tSHELL SORT ALGORITHM\n\n\n\n");
    printf("Enter total elements : ");
    scanf("%d", &n);
    printf("\nEnter %d Elements : ", n);
    for (i = 0; i < n; i++)
     scanf("%d", &a[i]);
    printf("\nBefore Sorted list by Shell sort :\n ");
    for(i=0;i< n;i++)
    printf("%d\t",a[i]);
    shellsort(a,n);
    printf("\nAfter Sorted list by Shell sort :\n ");
    for(i=0;i< n;i++)
    printf("%d\t",a[i]);
    getch();
    return 0;
}
```

OUTPUT


Turbo C++ IDE
SHELL SORT ALGORITHM
Enter total elements : 8
Enter 8 Elements : 56 85 92 99 67 28 95 15
Before Sorted list by Shell sort :
56 85 92 99 67 28 95 15
Pass 4:56 28 92 15 67 85 95 99
Pass 2:56 15 67 28 92 85 95 99
Pass 1:15 28 56 67 85 92 95 99
After Sorted list by Shell sort :
15 28 56 67 85 92 95 99

5.RADIX SORT

The sorting techniques that we have seen so far sorts the list of elements by comparing the sequence of elements and swaps them if necessary. The radix sort also referred, as the **bucket sort** is little bit different. It manages to sort values without actually performing any comparisons on the input data. The values are successively ordered on digit positions from right to left (i.e., lower order byte to higher order byte). This is accomplished by copying the values into buckets, where the index is given by the position of the digit being sorted. Once all digit positions have been examined, the list must be sorted.

Radix is just a position in a number. In hexadecimal representation of a decimal number, a radix indicates a digit. Radix sort gets its name from the radices, because the method first sorts the input values according to their first radix, then according to the second and so on. The number of passes in the radix sort equals the number of radices in the input values. For example you will need 2 passes to sort 16-bit integers and 4 passes to sort 32-bit integers. In a hexadecimal number system, radix is referred as a byte, and hence the radix sort is often referred as a **byte sort**. Radix sort uses 255 buckets for placing 1 byte data from 00 to FF.

Algorithm for radix sort

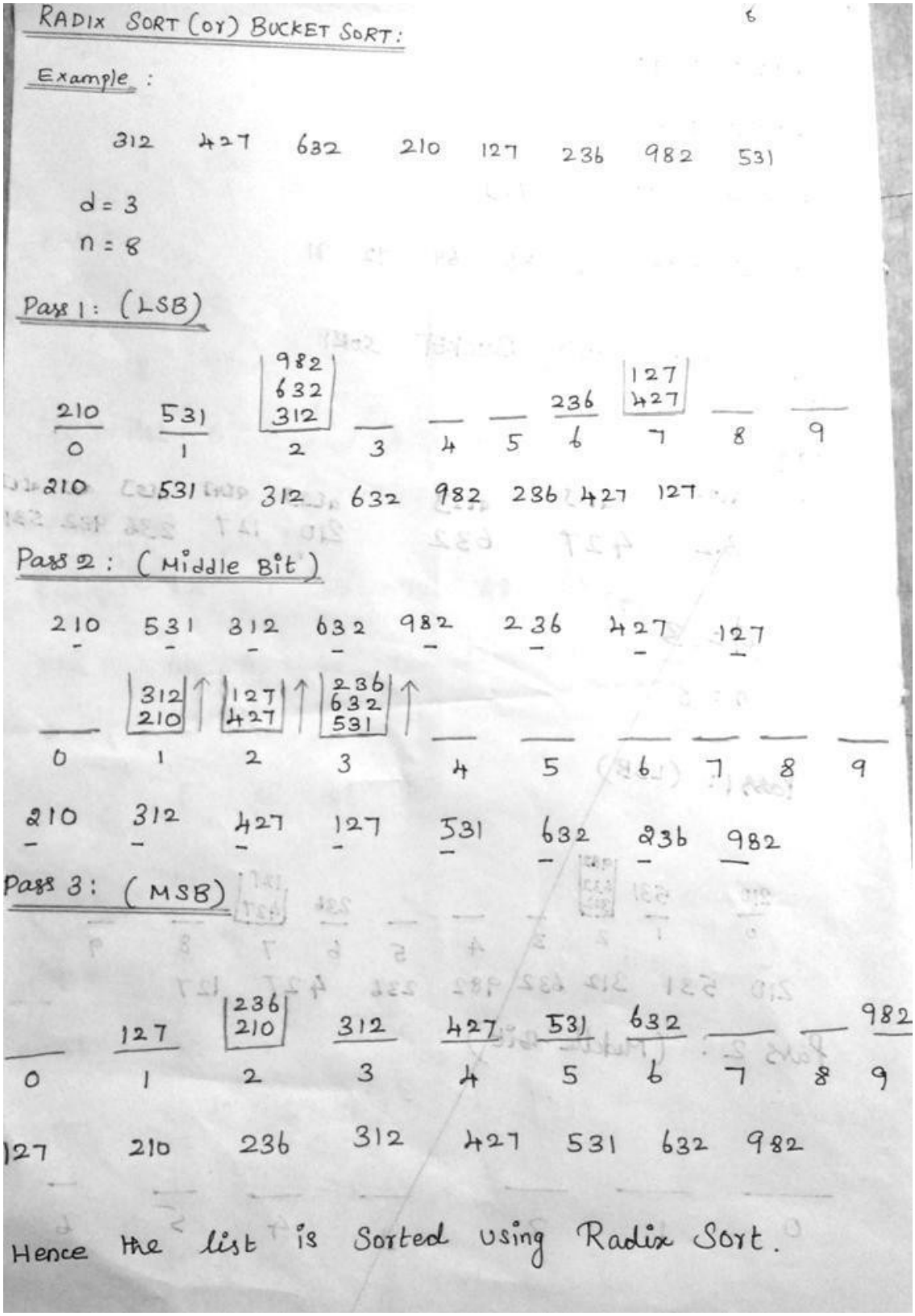
RADIX (ARR, N)

where ARR is an array of N elements

STEP 1 : Repeat For I = 1, 2, 3, . . ., 255
STEP 2 : Initialize bucket_size[I] = 0
STEP 3 : [End of **STEP 1** For loop]
STEP 4 : Repeat For M = 1 to 4 where M refers to the byte position (from lower order to higher order)
STEP 5 : Assign DATA1 = Mth lower order byte of ARR[J], bucket_size[DATA1] = bucket_size[DATA1]+1, TEMP[J] = ARR[J]
STEP 6 : [End of **STEP 4** For loop]
STEP 7 : Assign bucket_size[0] = 0
STEP 8 : Repeat For K = 1, 2, 3, . . ., 255
STEP 9 : Assign FIRST_IN_BUCKET[K] = FIRST_IN_BUCKET[K–1] + BUCKET_SIZE[K–1]
STEP 10 : [End of **STEP 1** For loop]

STEP 11 : Repeat For R = 1, 2, . . ., N–1
STEP 12 : Assign DATA2 = Mth lower order byte of TEMP[R]
ARR[FIRST_IN_BUCKET[DATA2]] = TEMP[R]
FIRST_IN_BUCKET[DATA2] = FIRST_IN_BUCKET[DATA2]+1
STEP 13 : [End of **STEP 11** For loop]
STEP 14 : Print the sorted array ARR
END RADIX()

Example :-



RADIX SORT (OR) BUCKET SORT:

Example :

312 427 632 210 127 236 982 531

d = 3

n = 8

Pass 1: (LSB)

0	1	2	3	4	5	6	7	8	9
210	531	982, 632, 312				236	127, 427		

210 531 312 632 982 236 427 127

Pass 2: (Middle Bit)

0	1	2	3	4	5	6	7	8	9
	312, 210 ↑	127, 427 ↑	236, 632, 531 ↑						

210 312 427 127 531 632 236 982

Pass 3: (MSB)

0	1	2	3	4	5	6	7	8	9
	127	236, 210	312	427	531	632			982

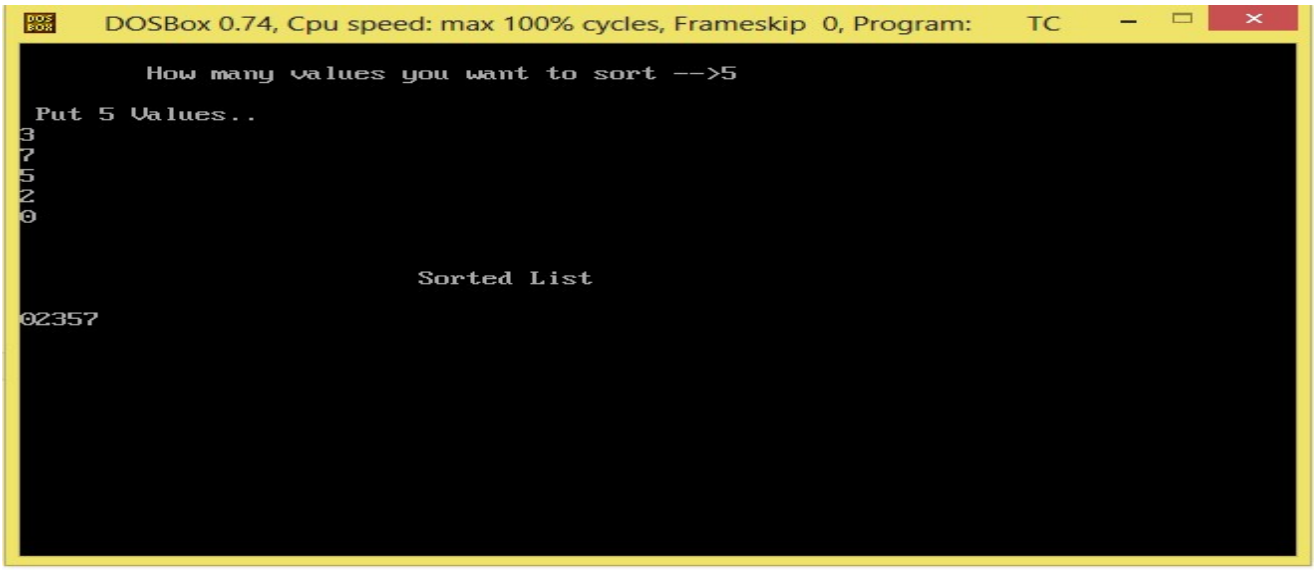
127 210 236 312 427 531 632 982

Hence the list is Sorted using Radix Sort.

```
// Radix Sort

  #include <stdio.h>
  #include <conio.h>
  int MAX  =  100;
  int A[100];
  int C[100][10]; // Two dimensional array, where each row has ten pockets
  int N;
  void get_Values()
          {
                int i=0;
                printf("\n\tHow many values you want to sort -->");
                scanf("%d",&N);
                if(N>MAX)
                        {
                                printf("\n\n \t Maximum values Limits %d so try again...",MAX);
                               get_Values(); // recursive call
                         }
                printf("\n Put %d Values..\n",N);
               for(i=0 ; i< N ; i++)
               scanf("%d",&A[i]);
          }
  void RadixSort (int R)
          {
                int i,j,X,D,m,p;
                int Z=0;
                D=R/10;
                for(i=0;i<N;i++)
                        {
                            for(j=0 ; j < 10 ; j++) C[i][j]=-1;
                        }
          for(i=0; i<N ; i++)
                       {
                            X=A[i]%R;
                            m=X/D;
                            if(m>0)Z=1;
                                    C[i][m]=A[i];
                        }
                 p=-1;
                 for(j=0 ; j < 10 ; j++)
                        {
                                for( i=0 ; i < N ; i++)
                                     {
                                            if(C[i][j]!=-1)
                                                    {
                                                            p++;
                                                            A[p]=C[i][j];
                                                     }
                                      }
                        }
                  if(Z==1)
```

```
                    {
                        R*=10;
                        RadixSort(R); // recursive call
                    }
        } // end of RadixSort( ) function


  void display()
        {
                int i;
                printf("\n\n\t\t\t Sorted List\n\n");
                for(i=0; i<N ; i++)
                printf("%d",A[i]);
          }

  void main()
        {
                clrscr();
                get_Values();
                if(N>1) RadixSort (10); // Here 10 is the Radix of Decimal Numbersl
                        display();
                getch();
        }
```

OUTPUT



6.<u>Explain various hashing techniques with suitable example</u>.

<u>Separate chaining:</u>

Chaining

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

For example:

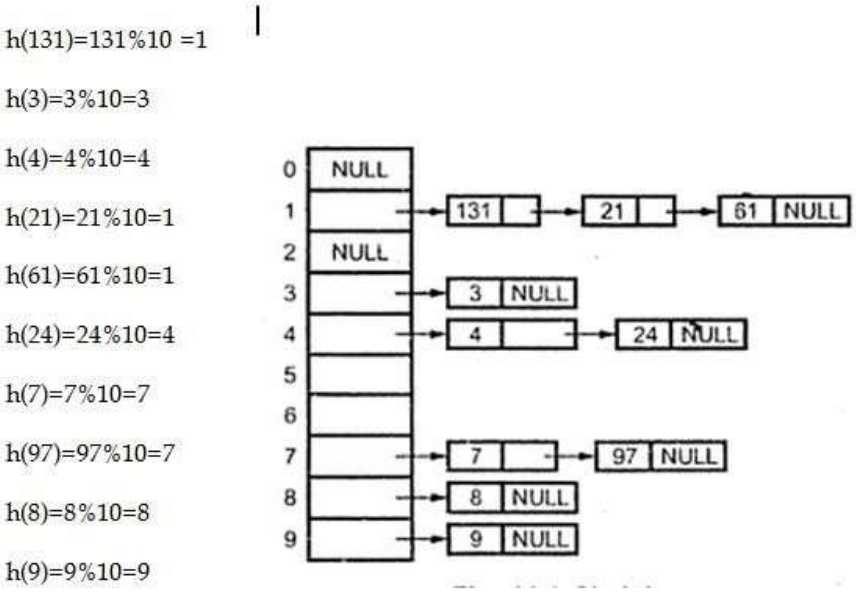
Consider the keys to be placed in their home buckets are

131, 3, 4, 21, 61, 24, 7, 97, 8, 9

Then we will apply a hash function as

H(key) = keytoD

where D is the size of table. The hash table will be: Here D = 10.



A chain is maintained for colliding elements. For instance 131 has a home bucket (key) 1. Similarly keys 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1. Similarly the chain at index 4 and 7 is maintained.

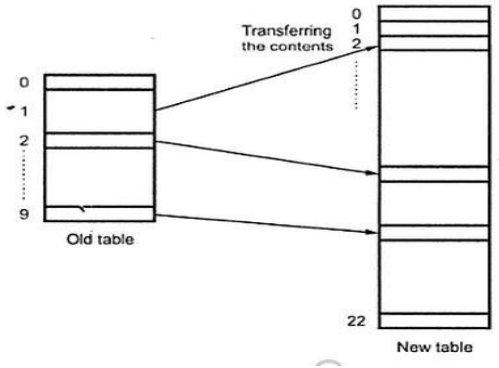
7. Explain in detail about Rehashing

Rehashing

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable if the total size of table is a prime number. There are situations in which the rehashing is required

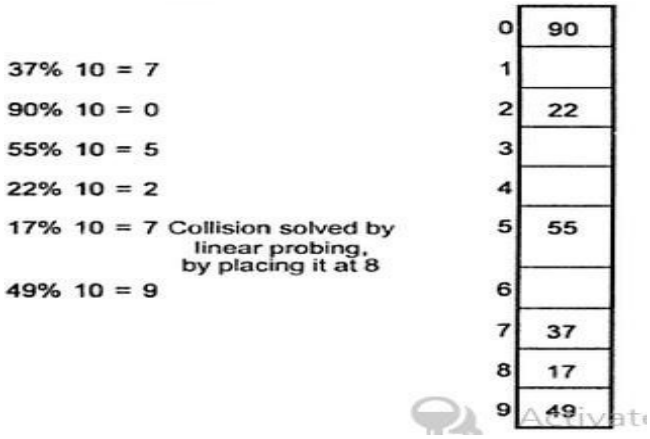
- When table is completely full.
- With quadratic probing when the table is filled half.
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re-computing their positions using suitable hash functions.



Consider we have to insert the elements 37, 90, 55, 22, 17, 49 and 87. The table size is 10 and will use hash function,

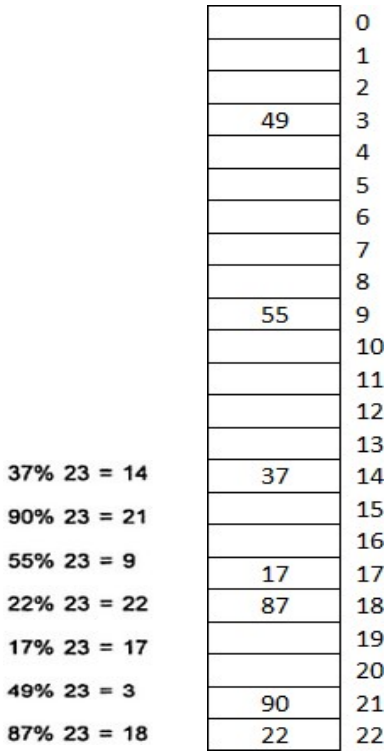
H(key) = key mod tablesize



Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size.

The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be

H(key) = key mod 23



Rehashing Example

8. Explain about Extensible hashing.

Extensible hashing:

- When the amount of data is too large to fit in main memory, the previous hashing techniques will not work properly.
- So we use a new technique called **extensible hashing**. It is one form of dynamic hashing.

Here the

- Keys are stored in buckets.
- Each bucket can only hold a fixed size of keys.
- Consider initially the directory splitting (g) as 2 so the directory is 00,01,10,11.

Example:

Suppose that g=2 and bucket size = 4.

Suppose that we have records with these keys and hash function

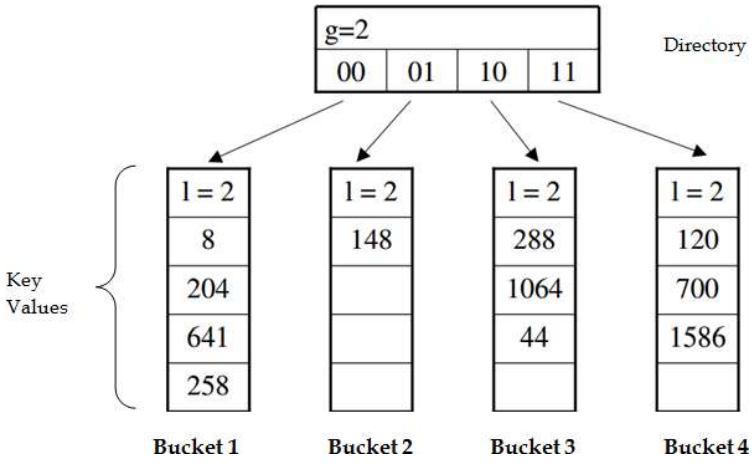
h(key) = key mod 64

the value 64 is obtained by 2^4 X4=16x4=64

key	h(key) = key mod 64	bit pattern
288	32	100000
8	8	001000
1064	40	101000
120	56	111000
148	20	010100
204	12	001100
641	1	000001
700	60	111100
258	2	000010
1586	50	110010
44	44	101010

Calculating bit pattern of each key

Check the first 2 bits(prefix bits) of the bit pattern and then place the key in the corresponding directory's bucket.



Extendible Hashing Example –directory and bucket structure

10-Marks

1. **Briefly explain the three common collision strategies in open addressing hashing.**

(OR)

Explain Linear Probing with an example.

Collision Resolution by Open Addressing

- Once a collision takes place, open addressing computes new positions using a probe sequence and the next record is stored in that position.
- In this technique of collision resolution, all the values are stored in the hash table.
- The process of examining memory locations in the hash table is called **probing.**
- Open addressing technique can be implemented using:
  - Linear probing
  - Quadratic probing
  - Double hashing.

Linear Probing

The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at location generated by *h(k),* then the following hash function is used to resolve the collision.

$$h(k, i) = [h'(k) + i] \bmod m$$

where, *m* is the size of the hash table, *h'(k) = k mod m* and *i* is the probe number and varies from *0* to *m-1*.

Example: Consider a hash table with size = 10. Using linear probing insert the keys 72, 27, 36, 24, 63, 81 and 92 into the table.

Let h'(k) = k mod m, m = 10

Initially the hash table can be given as,

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

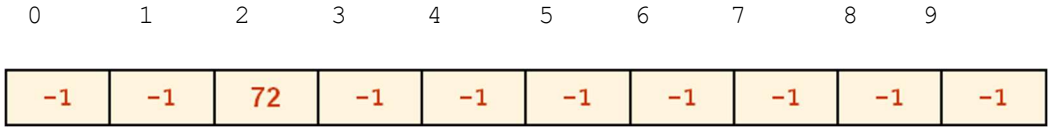
Step1: Key = 72

h(72, 0) = (72 mod 10 + 0) mod 10

= (2) mod 10

= 2

Since, T[2] is vacant, insert key 72 at this location



Step2: Key = 27

h(27, 0) = (27 mod 10 + 0) mod 10

= (7) mod 10

= 7

Since, T[7] is vacant, insert key 27 at this location

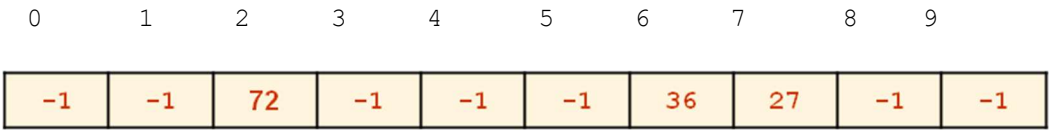
Step3: Key = 36

h(36, 0) = (36 mod 10 + 0) mod 10

= (6) mod 10

= 6

Since, T[6] is vacant, insert key 36 at this location



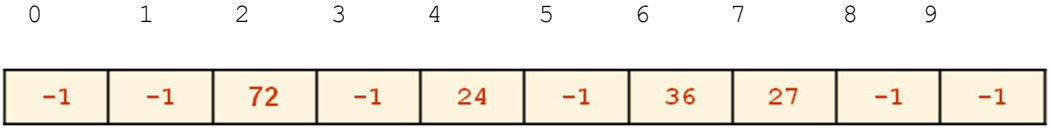
Step4: Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10

= (4) mod 10

= 4

Since, T[4] is vacant, insert key 24 at this location



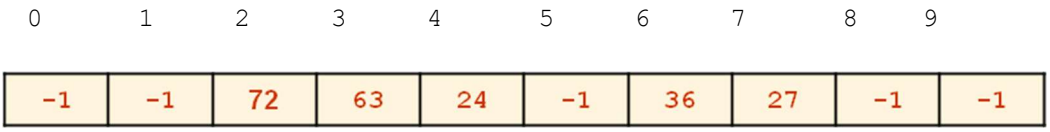
Step5: Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10

= (3) mod 10

= 3

Since, T[3] is vacant, insert key 63 at this location



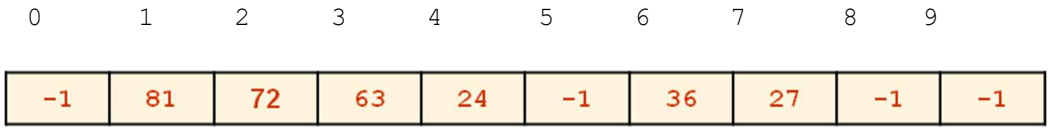
Step6: Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10

= (1) mod 10

= 1

Since, T[1] is vacant, insert key 81 at this location



Step7: Key = 92

h(92, 0) = (92 mod 10 + 0) mod 10

= (2) mod 10

= 2

Now, T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for next location. Thus probe, i = 1, this time.

Key = 92

h(92, 1) = (92 mod 10 + 1) mod 10

= (2 + 1) mod 10

= 3

Now, T[3] is occupied, so we cannot store the key 92 in T[3]. Therefore, try again for next location. Thus probe, i = 2, this time.

Key = 92

h(92, 2) = (92 mod 10 + 2) mod 10

= (2 + 2) mod 10

= 4

Now, T[4] is occupied, so we cannot store the key 92 in T[4]. Therefore, try again for next location. Thus probe, i = 3, this time.

Key = 92

$$h(92, 3) = (92 \bmod 10 + 3) \bmod 10$$

$$= (2 + 3) \bmod 10$$

$$= 5$$

Since, T[5] is vacant, insert key 92 at this location

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	92	36	27	-1	-1

Quadratic probing

- It eliminates the primary clustering problems of linear probing
  $f(i)=i^2$
- If quadratic probing is used and the table size is prime , then a new element can always be inserted if the table is at least half empty.
- If the table is even one more than half full, the insertion could fail [prime].

{250,567,252,763,424}

	Empty table	After 250	567	252	763	424
0		250	250	250	250	250
1						
2			567	567	567	567
3				252	252	252
4					763	763

When 424 collides with 763, the next position attempted in one cell away. But another collision occurs.
$1^2, 2^2, 3^2, \ldots$

Double Hashing:

The double hashing is performed by

$$F(i)= i.hash_2(x)$$

here , $hash_2(x) = R-(x \bmod R)$ with R is a prime smaller than table size.

Example:

Let us consider following key values 89, 18, 49, 58,69

For first value apply the normal hash function i.e **key mod tablesize**

hash(89)=89%10=9

hash(18)=18%10=8

Now the key values 89 and 18 are stored at the corresponding location ,when for inserting the 3rd element 49

hash(49)=49%10=9 , collision occurs

so apply double hashing technique and Choose "R" a prime number smaller than table size, so we choose R=7 then

$hash_2(49)$=7-(49%7)= 7 - 0 =7

$hash_2(58)$=7-(58%7)= 7 – 2 =5

$hash_2(69)$=7-(69%7)= 7 – 6 = 1

	Empty table	89	18	49	58	69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

2. <u>What is a graph? Write short notes on its basic terminologies.</u>

<u>DEFINING GRAPH</u>

- A graphs G consists of a set V of vertices (nodes) and a set E of edges (arcs).
- We write G=(V,E). V is a finite and non-empty set of vertices.
- E is a set of pair of vertices; these pairs are called as edges .
- Therefore (G).read as V of G, is a set of vertices and E(G),read as E of G is a set of edges.
- An edge e= (v, w) is a pair of vertices v and w, and to be incident with v and w.

- **A graph can be pictorially represented as follows,**

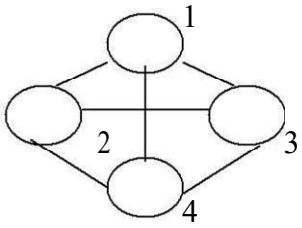


FIG: Graph G

- We have numbered the graph as 1,2,3,4.
- Therefore, V(G)=(1,2,3,4) and E(G) = {(1,2),(1,3),(1,4),(2,3),(2,4)}.

BASIC TERMINOLGIES OF GRAPH UNDIRECTED GRAPH

An undirected graph is that in which, the pair of vertices representing the edges is unordered.

DIRECTED GRAPH

- ***A directed graph is that in which, each edge is an ordered pair of vertices, (i.e.) each edge is represented by a directed pair.***
- ***It is also referred to as digraph.***

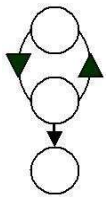


Fig.: Directed Graph

COMPLETE GRAPH

A vertex undirected graph with exactly n(n-1)/2 edges is said to be complete graph.

SUBGRAPH

A sub-graph of G is a graph G‘ such that V(G’) follow,

V(G) and E(G ‘) E(G). Some of the sub-graphs are as

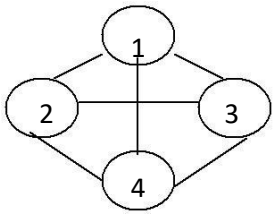
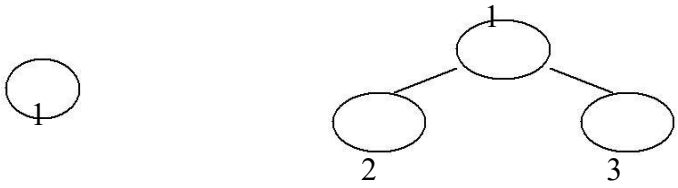
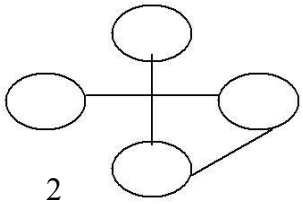


FIG: Graph G



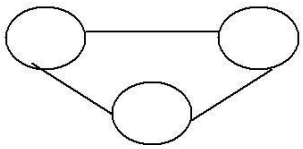
(a) (b)



2 3

FIG: Sub graphs of G (c) 4 (d)

4



ADJACENT VERTIC

A vertex v1 is said to be a adjacent vertex if there exist an edge (v1,v2) or (v2,v1).

PATH:

A path from vertex V to the vertex W is a sequence of vertices, each adjacent to the next. The length of the graph is the number of edges in it.

CONNECTED GRAPH

A graph is said to be connected if there exist a path from any vertex to another vertex.

UNCONNECTED GRAPH

A graph is said to be an unconnected graph if there exist any two unconnected components.

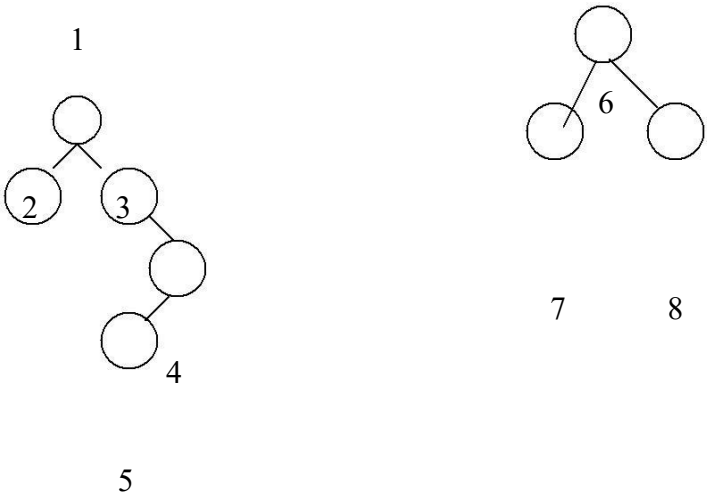
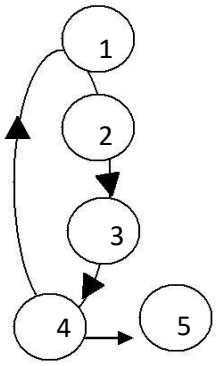


FIG: Unconnected Graph

STRONGLY CONNECTED GRAPH

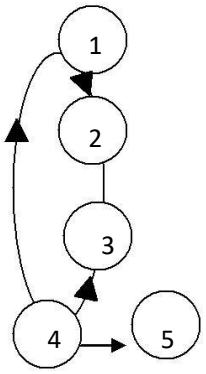
A digraph is said to be strongly connected if there is a directed path from any vertex to any other vertex

.



WEAKLY CONNECTED GRAPH

If there does not exist a directed path from one vertex to another vertex then it is said to be a weakly connected graph.



CYCLE

A cycle is a path in which the first and the last vertices are the same.

Eg. 1,3,4,7,2,1

DEGREE:

The number of edges incident on a vertex determines its degree. There are two types of degrees In-degree and out-degree.

- IN-DEGREE of the vertex V is the number of edges for which vertex V is a head.
- OUT-DEGREE is the number of edges for which vertex is a tail.

A GRAPH IS SAID TO BE A TREE, IF IT SATISFIES THE TWO PROPERTIES:

1. It is connected
2. There are no cycles in the graph.

GRAPH REPRESENTATION:

The graphs can be represented by the follow three methods,

1. Adjacency matrix.
2. Adjacency list.
3. Adjacency multi-list.

ADJACENCY MATRIX:

The adjacency matrix A for a graph G = (V,E) with n vertices, is an n* n matrix of bits ,such that A Ij = 1 , if there is an edge from vi to vj and

Aij = 0, if there is no such edge. The adjacency matrix for the graph G is,

$$\begin{matrix} 0\ 1\ 1\ 1 \\ \\ 1\ 0\ 1\ 1 \\ \\ 1\ 1\ 0\ 1 \\ \\ 1\ 1\ 1\ 0 \end{matrix}$$

The space required to represent a graph using its adjacency matrix is n* n bits. About half this space can be saved in case of an undirected graph, by storing only the upper or lower triangle of the matrix. From the adjacency matrix, one may readily determine if there an edge connecting any two vertices I and j. for an undirected graph the degree of any vertices I is its row $\sum^{n}$ (j=1) A (I, j). For a directed graph the row is the out-degree and column sum is the in-degree.

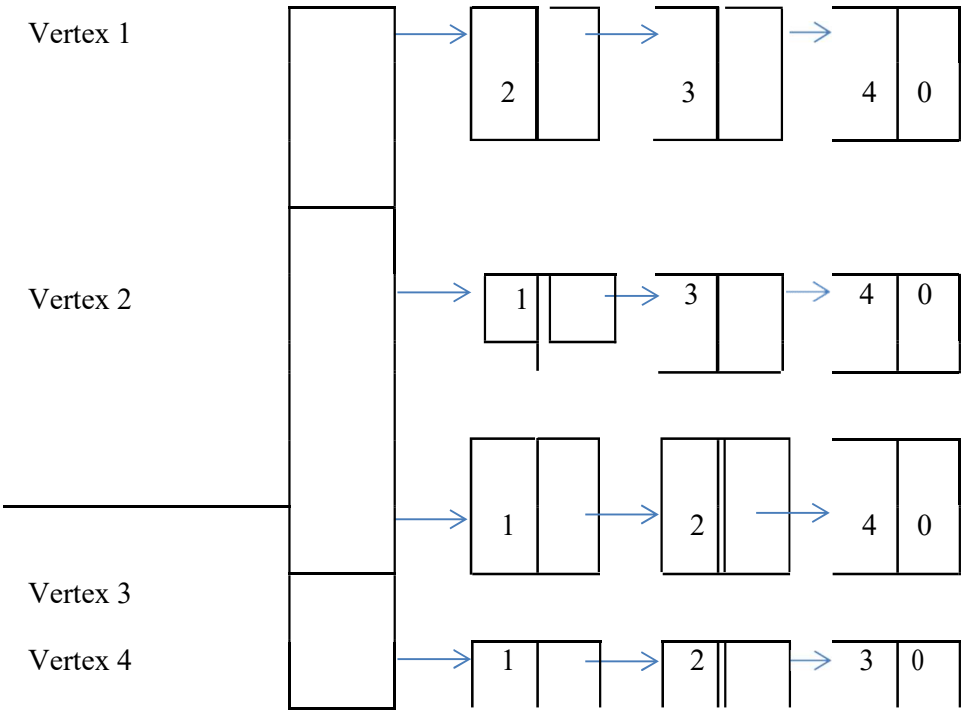
The adjacency matrix is a simple way to represent a graph , but it has 2 disadvantages,

- It takes O(n*n) space to represent a graph with n vertices ; even for sparse graphs and
- It takes… (n*n) times to solve most of the graph problems.

ADJACENCY LIST

In the representation of the n rows of the adjacency matrix are represented as n linked lists. There is one list for each vertex in G. the nodes in list I represent the vertices that are adjacent from vertex i. Each node has at least 2 fields: VERTEX and LINK. The VERTEX fields contain the indices of

the vertices adjacent to vertex i. In the case of an undirected graph with n vertices and E edges, this representation requires n head nodes and 2e list nodes.



The degree of the vertex in an undirected graph may be determined by just counting the number of nodes in its adjacency list. The total number of edges in G may , therefore be determined in time O(n+e).in the case of digraph the number of list nodes is only e. the out-degree of any vertex can be determined by counting the number of nodes in an adjacency list. The total number of edges in G can, therefore be determined in O(n+e).

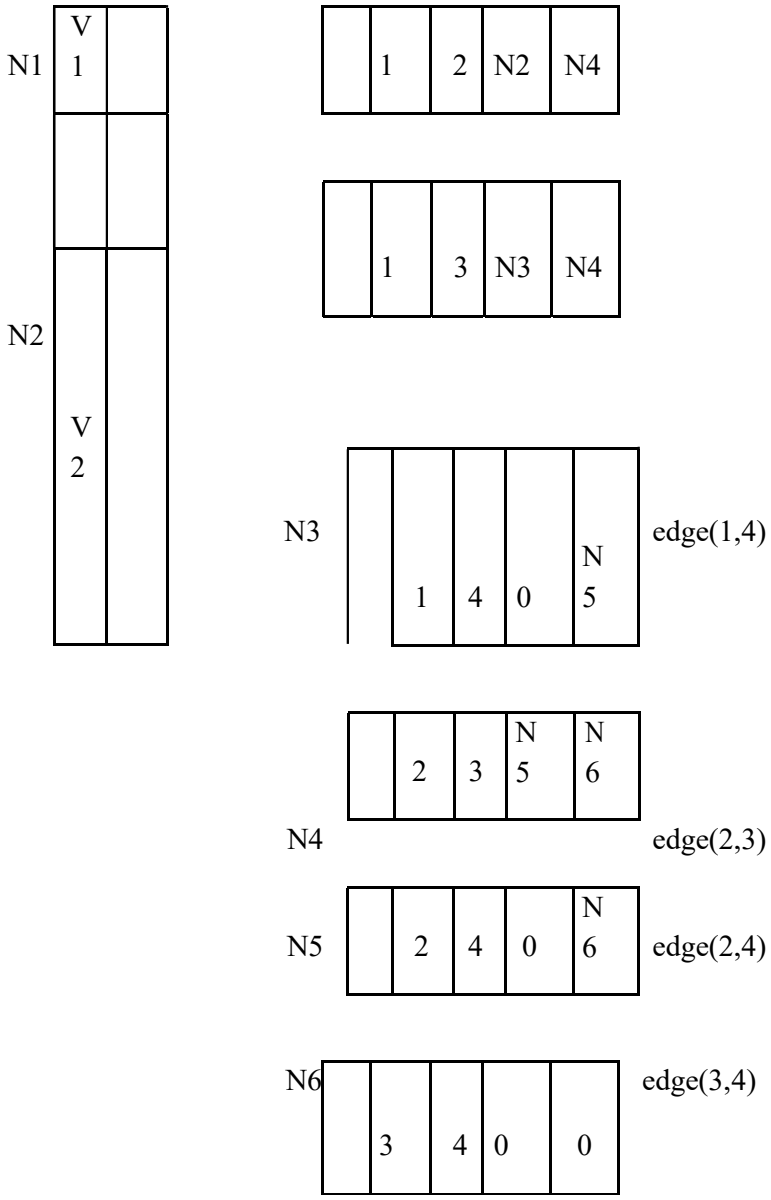
ADJACENCY MULTILIST:

In the adjacency list representation of an undirected graph each edge (vi, vj) is represented by two entries ,one on the list for Vi and the other on the list for Vj.

For each edge there will be exactly one node , but this node will be in two list , (i.e) the adjacency list for each of the two nodes it is incident to .the node structure now becomes

M	V_1	V_2	Link for V1	Link for V_2

Where M is a one bit mark field that may be used to indicate whether or not the edge has been examined. The adjacency multi-list diagram is as follow,



The lists are : v1: N1 N2 N3

V2: N1
N4 N5

V3: N2
N4 N6

V4: N3 N5 N6

FIG: Adjacency Multilists for G

3. What is graph traversal? What are its types? Explain (Nov 13, Nov 14, May 15)

GRAPH TRAVERSAL

Given an undirected graph G = (V.E) and a vertex v in V(G) we are interested in visiting all vertices in G that are reachable from v (that is all vertices connected to v). We have two ways to do the traversal. They are

DEPTH FIRST SEARCH

BREADTH FIRST SEARCH.

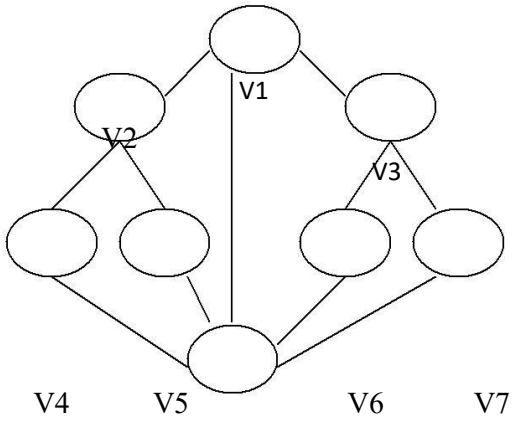
DEPTH FIRST SEARCH

In graphs, we do not have any start vertex or any special vertex singled out to start traversal from. Therefore the traversal may start from any arbitrary vertex.

We start with say, vertex v. An adjacent vertex is selected and a Depth First search is intimated from it, i.e. let V1, V2.. Vk are adjacent vertices to vertex v. We may select any vertex from this list. Say, we select V1. Now all the adjacent vertices to v1 are identified and all of those are visited; next V2 is selected and all its adjacent vertices visited and so on.

This process continues till are the vertices are visited. It is very much possible that we reach a traversed vertex second time. Therefore we have to set a flag somewhere to check if the vertex is already visited.

Let us see an example, consider the following graph.



Let us start with V1,

V8

1. Its adjacent vertices are V2, V8, and V3. Let us pick on v2.
2. Its adjacent vertices are V1, V4, V5, V1 is already visited.
3. Let us pick on V4.
4. Its adjacent vertices are V2, V8.

5. V2 is already visited .let us visit V8.
6. Its adjacent vertices are V4, V5, V1, V6, V7.
7. V4 and V1 are visited. Let us traverse V5.
8. Its adjacent vertices are V2, V8. Both are already visited therefore, we back track.
9. We had V6 and V7 unvisited in the list of V8. We may visit any. We may visit any. We visit V6.
10. Its adjacent is V8 and V3. Obviously the choice is V3.
11. Its adjacent vertices are V1, V7. We visit V7.
12. All the adjacent vertices of V7 are already visited, we back track and find that we have visited all the vertices.

Therefore the sequence of traversal is

V1, V2, V4, V5, V6, V3,
V7.

This is not a unique or the only sequence possible using this traversal method.

We may implement the Depth First search by using a stack, pushing all unvisited vertices to the one just visited and popping the stack to find the next vertex to visit

This procedure is best described recursively as in,

Procedure DFS(v)

-

// Given an undirected graph G = (V.E) with n vertices and an array visited (n) initially set to zero . This algorithm visits all vertices reachable from v .G and VISITED are global > //VISITED (v) 1

for each vertex w adjacent to v do *if*
VISITED (w) =0 then call DFS (w)
end

end DFS

COMPUTING TIME

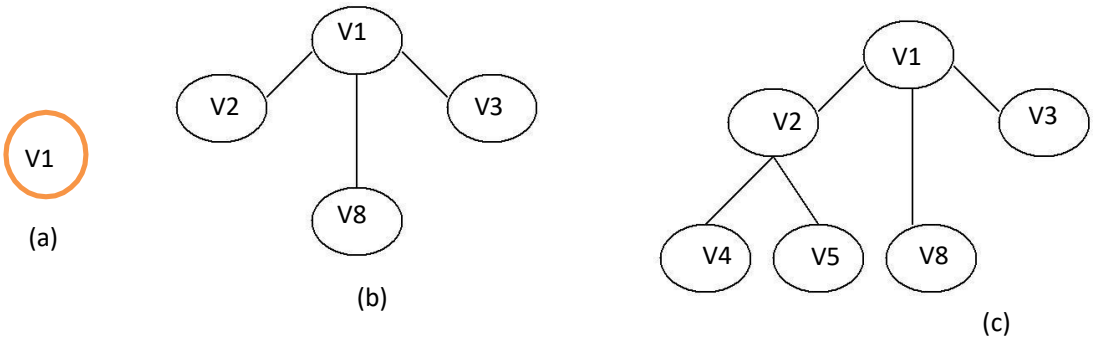
1. In case G is represented by adjacency lists then the vertices w adjacent to v can be determined by following a chain of links. Since the algorithm DFS would examine each node in the adjacency lists at most once and there are 2e list nodes. The time to complete the search is O (e).

2. If G is represented by its adjacency matrix, then the time to determine all vertices adjacent to v is O(n).

   Since at most n vertices are visited. The total time is $O(n^2)$.

BREADTH FIRST SEARCH

- In DFS we pick on one of the adjacent vertices; visit all of its adjacent vertices and back track to visit the unvisited adjacent vertices.
- In BFS , we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.
- Let us consider the same example, given in figure. We start say, with V1. Its adjacent vertices are V2, V8 , V3.
- We visit all one by one. We pick on one of these, say V2. The unvisited adjacent vertices to V2 are V4, V5 . we visit both.
- We go back to the remaining visited vertices of V1 and pick on one of this, say V3. T The unvisited adjacent vertices to V3 are V6,V7. There are no more unvisited adjacent vertices of V8, V4, V5, V6 and V7.



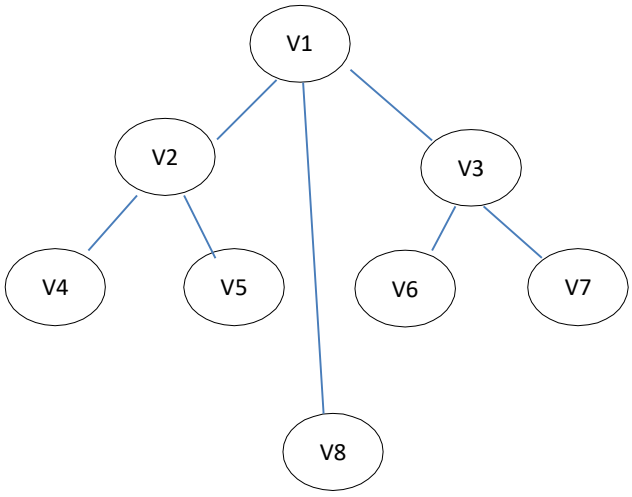


FIG: Breadth First Search

- Thus the sequence so generated is V1,V2, V8, V3,V4, V5,V6, V7. Here we need a queue instead of a stack to implement it.

- We add unvisited vertices adjacent to the one just visited at the rear and read at from to find the next vertex to visit.

Algorithm BFS gives the details.

Procedure BFS(v)

//A breadth first search of G is carried out beginning at vertex v. All vertices visited are marked as VISITED(I) = 1. The graph G and array VISITED are global and VISITED is initialised to 0.//

1. VISITED(v) 1
2. *Initialise Q to be empty //Q is a queue//*
3. loop
4. for all vertices w adjacent to v do
5. if VISITED(w) = 0 //add w to queue//
6. then [call ADDQ(w, Q); VISITED(w) 1] //mark w as VISITED//
7. end
8. if Q is empty then return
9. call DELETEQ(v,Q)
10. forever
11. end BFS

COMPUTING TIME

1. Each vertex visited gets into the queue exactly once, so the loop forever is iterated at most n times.
2. If an adjacency matrix is used, then the for loop takes O(n) time for each vertex visited. The total time is, therefore, $O(n^2)$.
3. In case adjacency lists are used the for loop as a total cost of d1+.....+dn = O(e) where di = degree(vi). Again, all vertices visited. Together with all edges incident to from a connected component of G.

3. What isgraphtraversal? Whatareitstypes? Explain(April 2011, April 2012, April 2013) Graph Traversal

Some applications require the graph to be traversed . This means that starting from some designated vertex the graph is walked around in a symmetric way such that vertex is visited only once. Two algorithms are commonly used:

1. Depth-First Search

2. Breadth-First Search

DEPTH FIRST SEARCH

1. DFS stands for "Depth First Search".
2. DFS in not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible.
3. DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise.
4. Depth First Search can be done with the help of Stack i.e. LIFO implementations.
5. This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped- off.
6. DFS is more faster than BFS.

Applications of DFS

1. Useful in Cycle detection
2. In Connectivity testing
3. Finding a path between V and W in the graph
4. useful in finding spanning trees & forest.

Algorithm: Depth First Search

1. PUSH the starting node in the Stack.

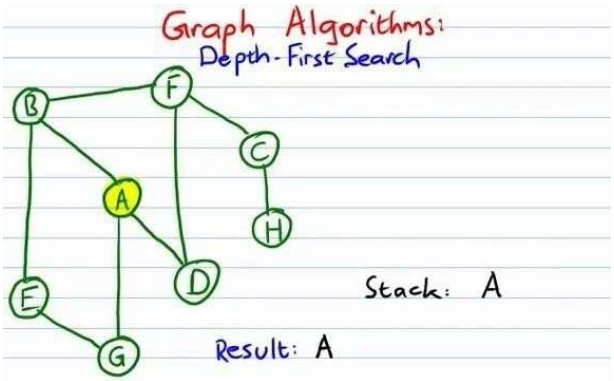
2. If the Stack is empty, return failure and stop.

3. If the first element on the Stack is a goal node g , return succeed and stop otherwise.

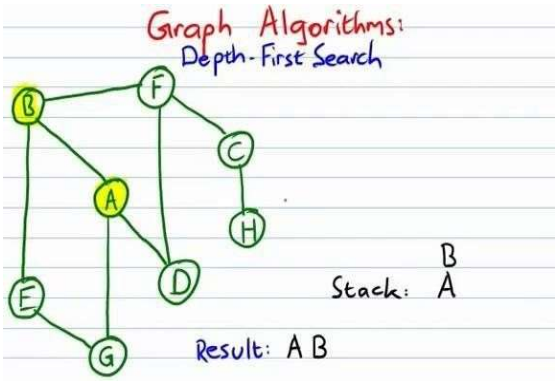
4. POP and expand the first element and place the children at the front of the Stack.

5. Go back to step 2

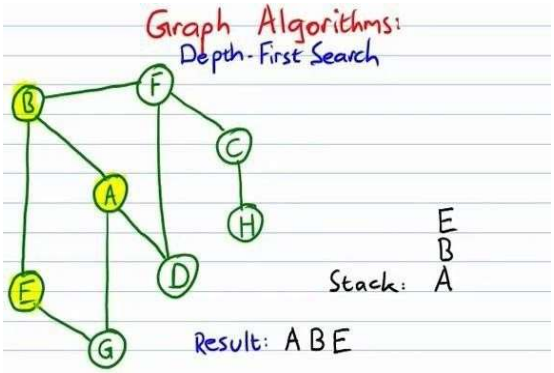
Step 1



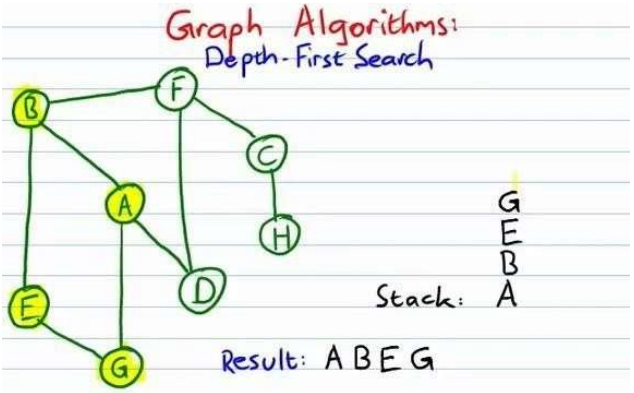
Step 2



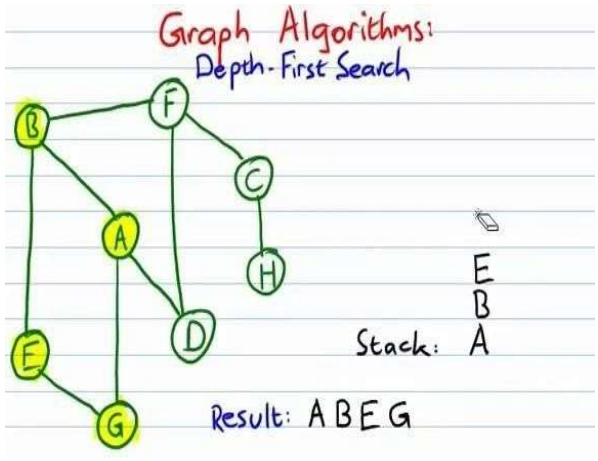
Step 3



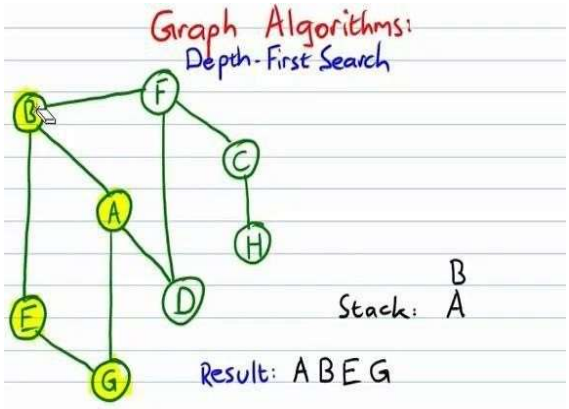
Step 4



Step 5

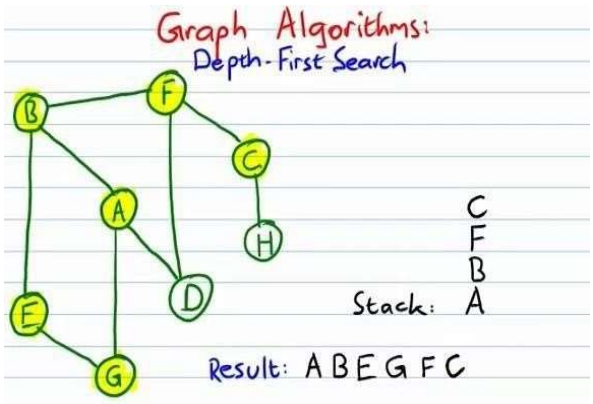


Step 6

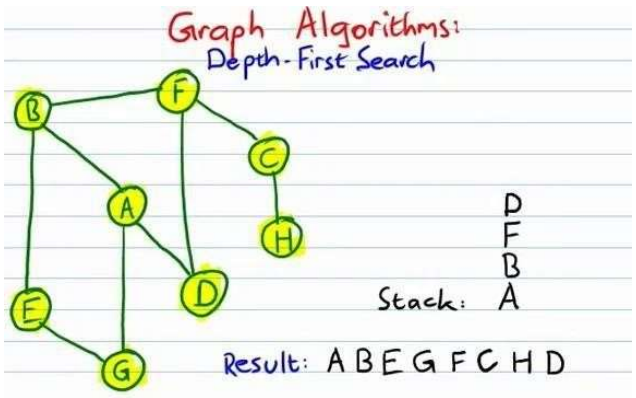


Step 7

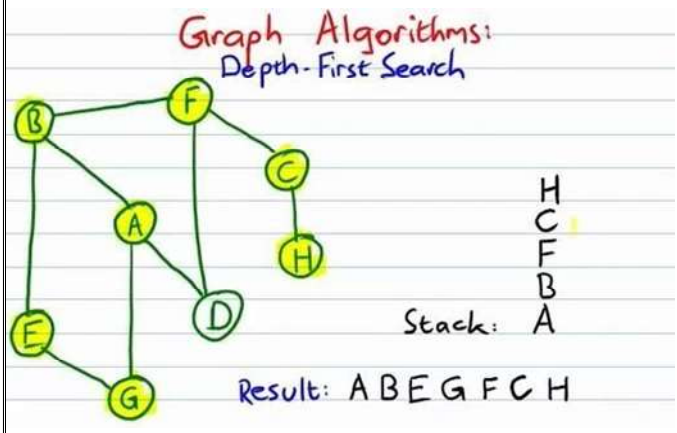
Step 8

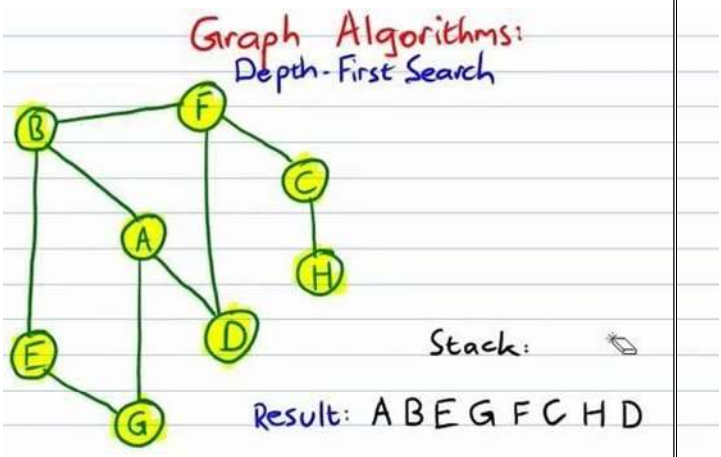


Step 9



Step 10





2. BREADTH FIRST SEARCH:-

1. BFS Stands for “Breadth First Search”.
2. BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.
3. BFS is useful in finding shortest path.BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph.
4. Breadth First Search can be done with the help of queue i.e. FIFO implementation.
5. This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.
6. BFS is slower than DFS.
7. BFS requires more memory compare to DFS.

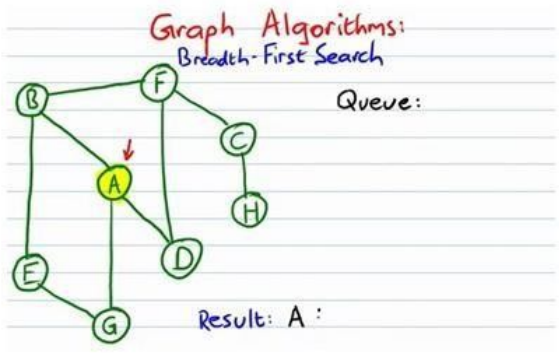
Applications of BFS

1. To find Shortest path
2. Single Source & All pairs shortest paths
3. In Spanning tree
4. In Connectivity

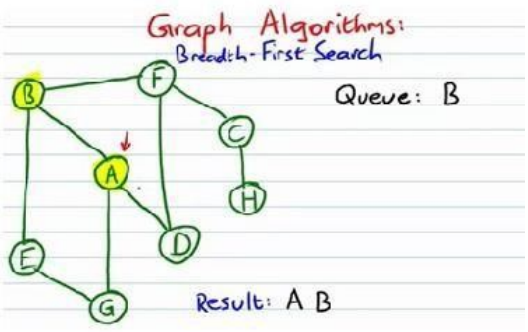
ALGORITHM: BREADTH - FIRST SEARCH

1. Place the starting node on the queue.
2. If the queue is empty return failure and stop.
3. If the first element on the queue is a goal node, return success an stop otherwise.
4. Remove and expand the first element from the queue and place all children at the end of the queue in any order.
5. Go back to step 1.

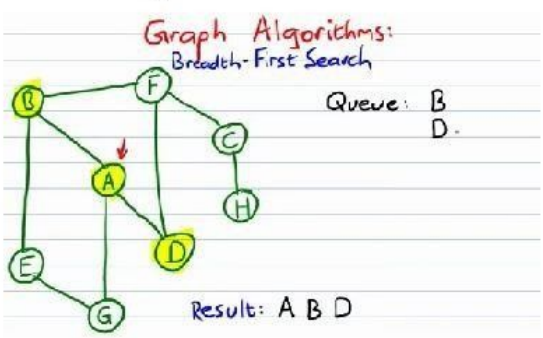
Step 1



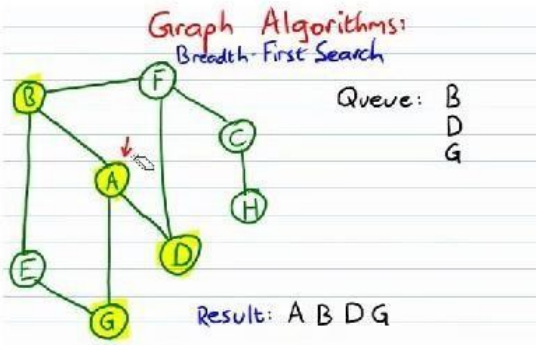
Step 2



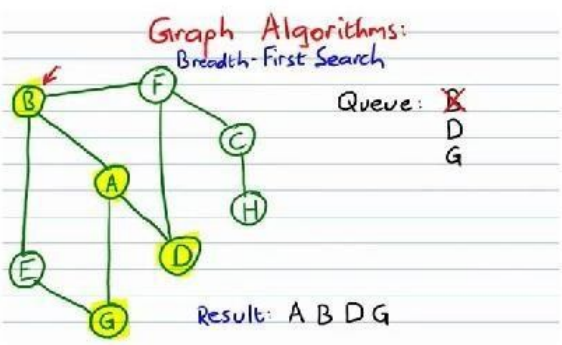
Step 3



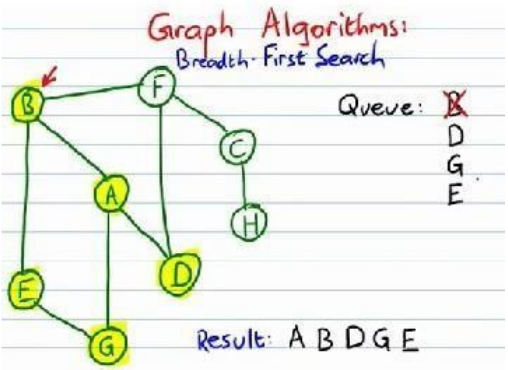
Step 4



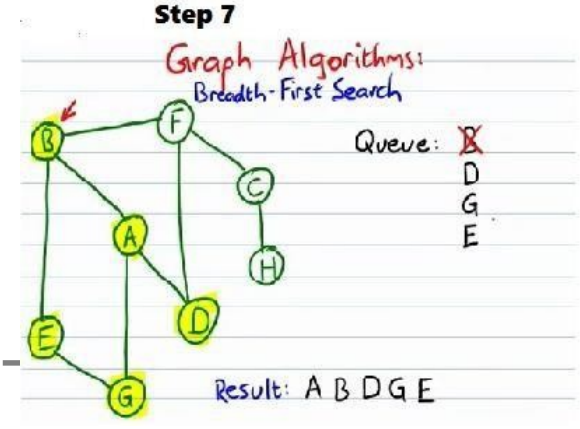
Step 5



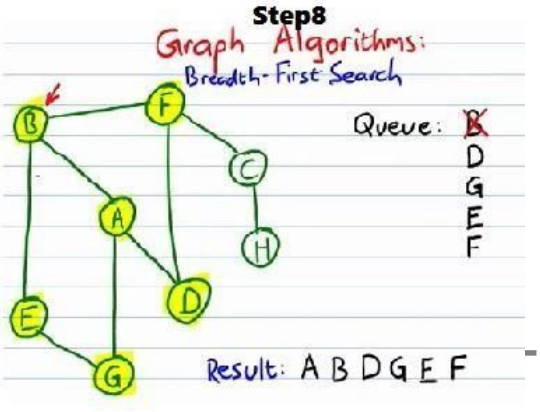
Step 6



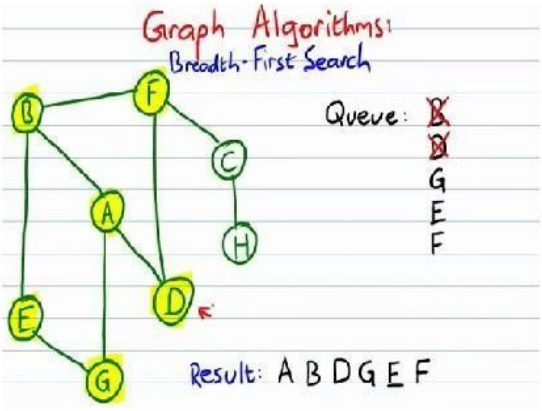
Step 7



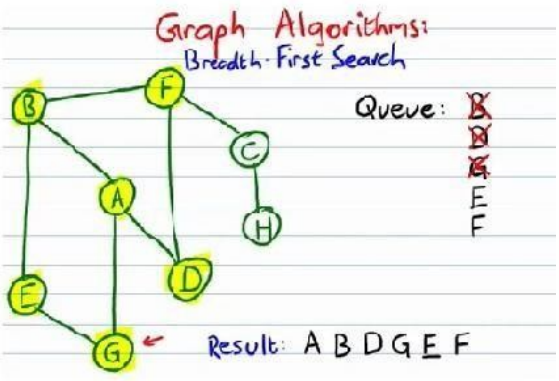
Step8



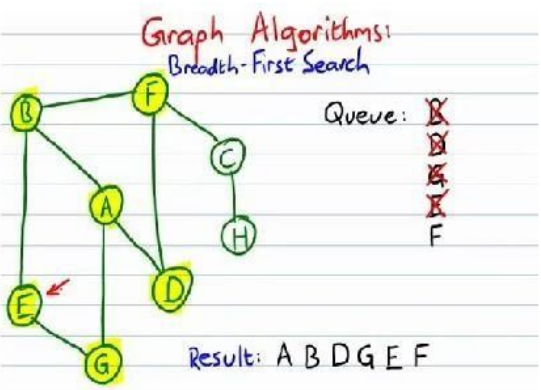
Step 9



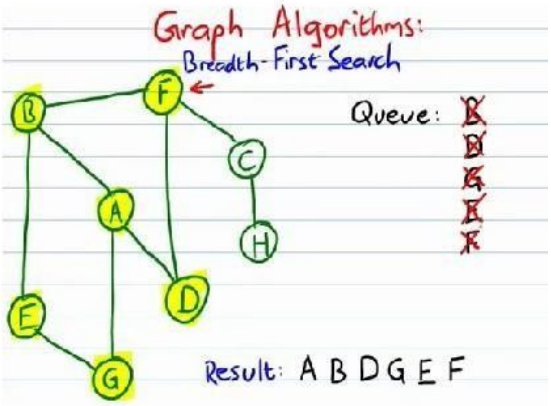
Step 10



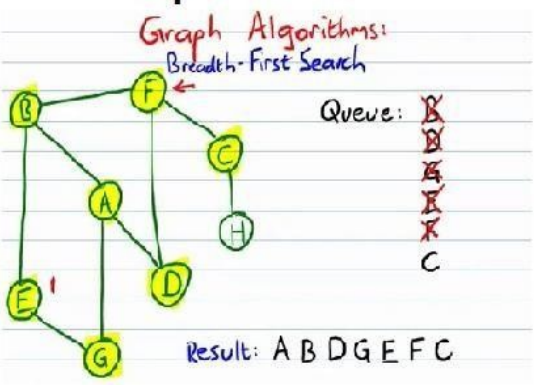
Step 11



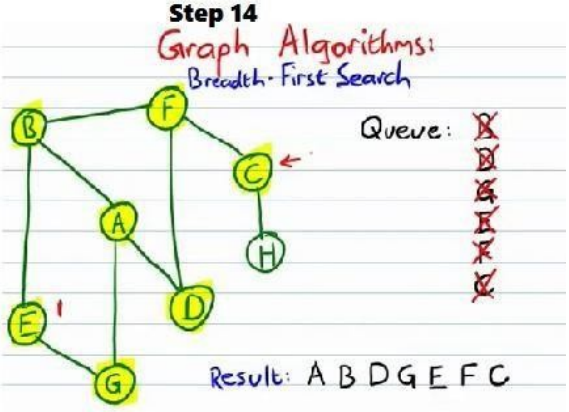
Step 12



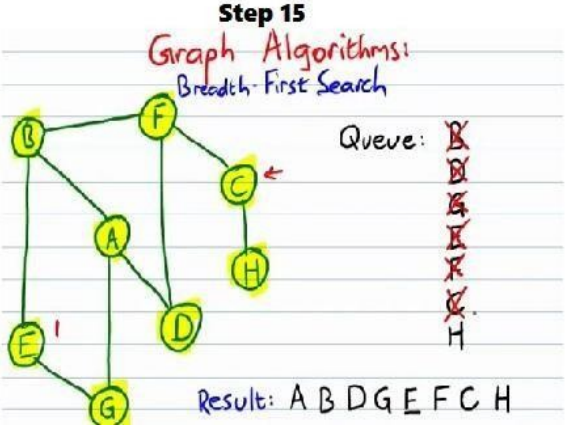
Step 13



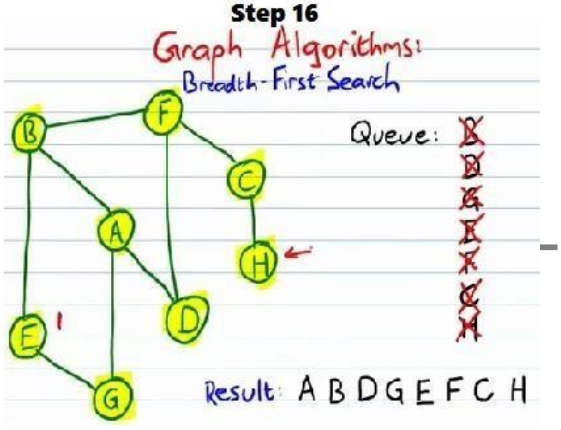
Step 14



Step 15



Step 16



BREADTHFIRSTSEARCH

- In DFS we pick on one of the adjacent vertices; visit all of its adjacent vertices and back track to visit the unvisited adjacent vertices.
- In BFS , we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.
- Let us consider the same example, given in figure. We start say, with V1. Its adjacent vertices are V2, V8, V3.
- We visit all one by one. We pick on one of these, say V2. The unvisited adjacent vertices to V2 are V4, V5 . we visit both.
- We go back to the remaining visited vertices of V1 and pick on one of this, say V3. T The unvisited adjacent vertices to V3 are V6,V7. There are no more unvisited adjacent vertices of
- V8, V4, V5, V6 and V7.

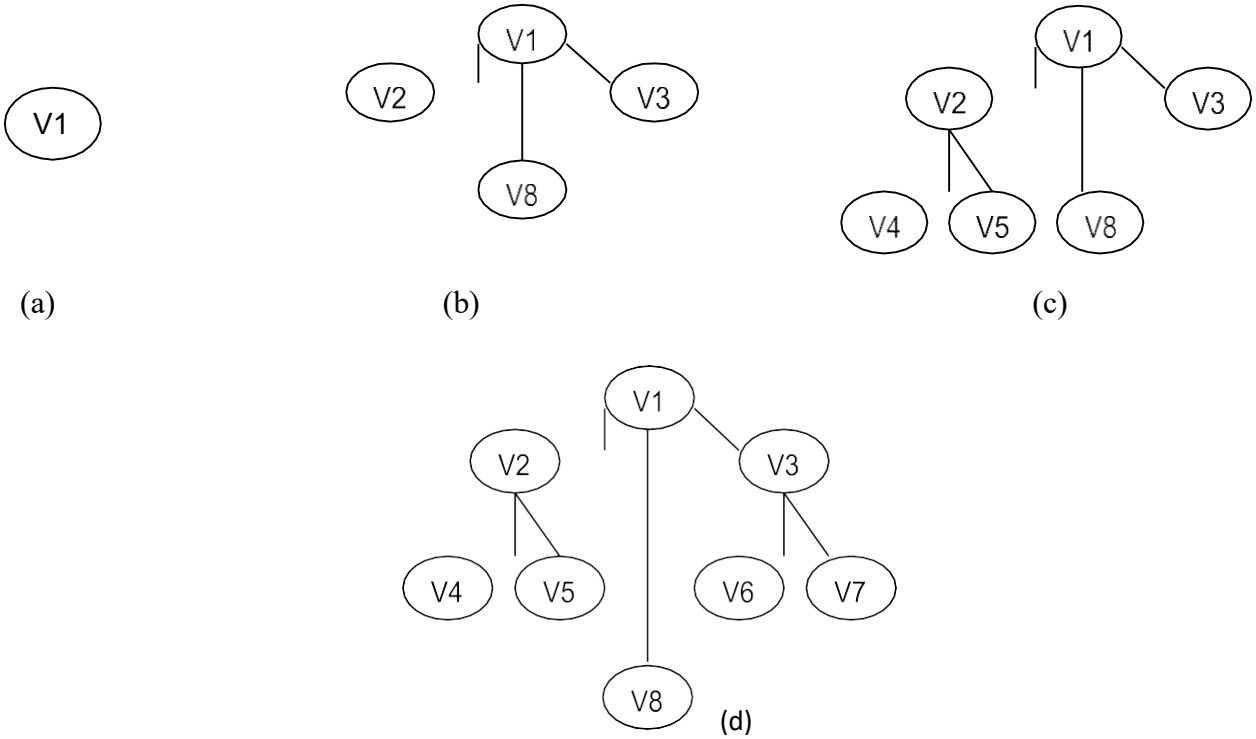


FIG: Breadth First Search

- Thus the sequence so generated is V1,V2, V8, V3,V4, V5,V6, V7. Here we need a queue instead of a stack to implement it.
- We add unvisited vertices adjacent to the one just visited at the rear and read at from to find the next vertex to visit.

Algorithm BFS gives the details.

Procedure BFS(v)

//A breadth first search of G is carried out beginning at vertex v. All vertices visited are marked as VISITED(I)

= 1. The graph G and array VISITED are global and VISITED is initialised to 0.//

1. VISITED(v) □ 1
2. Initialise Q to be empty //Q is a queue//
3. loop
4. for all vertices w adjacent to v do
5. if VISITED(w) = 0 //add w to queue//
6. then [call ADDQ(w, Q); VISITED(w) □ 1] //mark w as VISITED//
7. end
8. if Q is empty then return
9. call DELETEQ(v,Q)
10. forever
11. end BFS

Computing Time

1. Each vertex visited gets into the queue exactly once, so the loop forever is iterated at most n times.
2. If an adjacency matrix is used, then the for loop takes O(n) time for each vertex visited. The Total time is, therefore, $O(n^2)$.
3. In case adjacency lists are used the for loop as a total cost of $d1+ \ldots\ldots\ldots +d_n = O(e)$ where d_i = degree(v_i). Again, all vertices visited. Together with all edges incident to from a connected component of G.

4. What is minimum Spanning tree? Explain Kruskal's algorithm with the graph.(Nov 2012) Spanning Tree

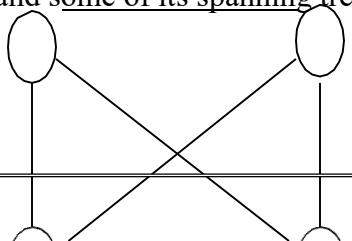
- When the graph G is connected, a depth first search starting at any vertex, visits all the vertices in G.
- In this case the edges of G are partitioned into two sets T (for tree edges) and B (for back edges), where T is the set of edges used or traversed during the search and B the set of remaining edges.
- The set T may be determined by inserting the statement

$$T \longleftarrow T \cup \{(v,w)\}$$

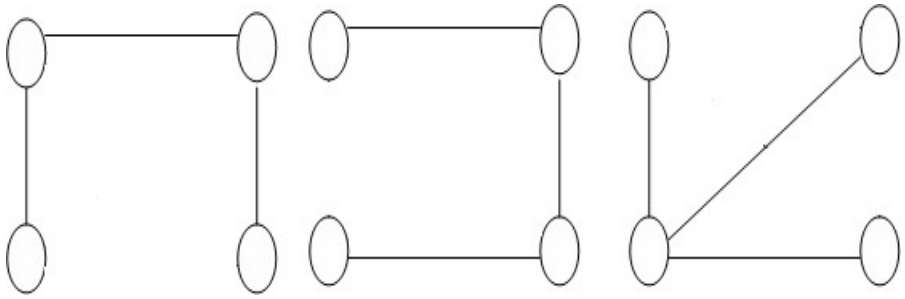
In the then clauses of DFS and BFS.

- The edges in T form a tree which includes all the vertices of G
- Any tree consisting solely of edges in G and including all vertices in G is called **spanning tree.**

The below diagram shows the graph G and some of its spanning trees.



Graph G



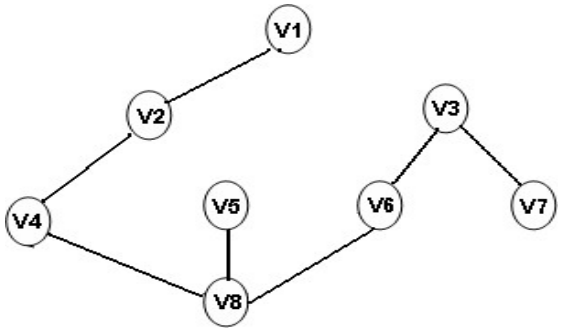
Its spanning trees

➢ **DEPTH FIRST SPANNING TREE**

When DFS are used the edges of T form a spanning tree

○ The spanning tree resulting from a call to DFS is known as a depth first spanning tree.

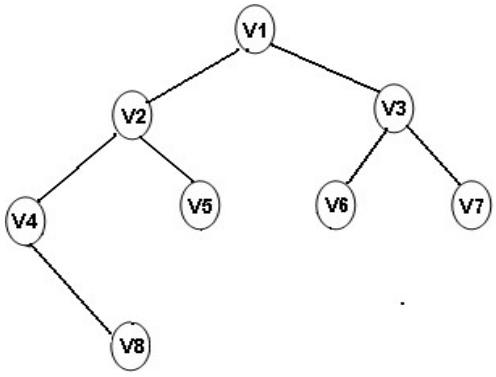
The below diagram shows the Depth first spanning tree



➢ BREADTH FIRST SPANNING TREE

- When BFS are used the edges of T form a spanning tree.
- The spanning tree resulting from a call to BFS is known as a breadth first spanning tree .

The below diagram shows the breadth first spanning tree



MINIMUM COST SPANNING TREES (APPLICATIONS OF SPANNING TREE):

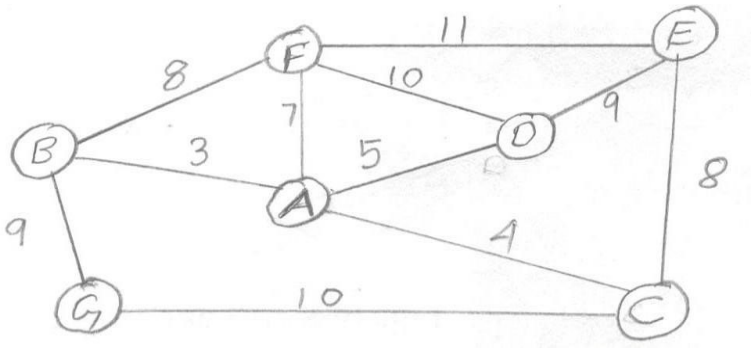
BUILDING A MINIMUM SPANNING TREE

- If the nodes of G represent cities and the edges represent possible communication links connecting two cities then the minimum number of links needed to connect the n cities is n-1. The spanning tree of G will represent all feasible choices.
- This application of spanning tree arises from the property that a spanning tree is a minimal sub-graph G' of G such that V (G') =V (G) and G' is connected where minimal sub-graph is one with fewest number of edges.
- The edges will have weights associated to there i.e. cost of communication. Given the weighted graph, one has to construct a communication links that would connect all the cities with minimum total cost.
- Since, the links selected will form a tree. We are going to find the spanning tree of a given graph withminimum cost. The cost of a spanning tree is the sum of the costs of the edges in the tree.

KRUSKAL'S METHOD FOR CREATING A SPANNING TREE

One approach to determine a minimum cost spanning of a graph has been given by kruskal. In this approach we need to select (n-1) edges in G, edge by edge, such that these form an MST or G. We can select another least cost edge and so on. An edge is included in the tree if it does not form a cycle with the edges already in T.

For example, consider the following graph,



The minimum cost edge is that connects vertex A and B. let us choose that Fig.(a)

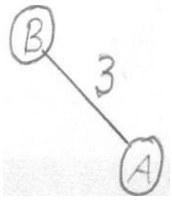


Fig. (a)

Now we have, Vertices that may be

added,

	Edge	cost
F	BF, AF	8, 7
G	BG, CG	9, 10
D	AD	5
E	CE	8
C	AC	4

The least cost edge is AC therefore we choose AC. Now we have Fig. (b),

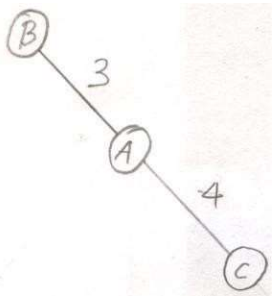
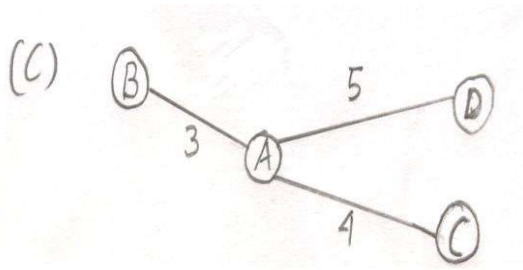


Fig. b

	EDGE	COST
F	BF	8
	AF	7
G	BG	9
	CG	10
D	AD	5
E	CE	8
C	AC	4

The least cost edge is Ad therefore we have



Vertices that may be added,

	EDGE	COST
F	BF	8
	AF	7
	DF	10
G	BG	9
	CG	10
E	CE	8
	DE	9

AF is the minimum cost edge therefore we add it to the partial tree we have

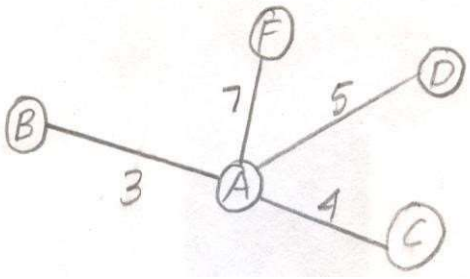
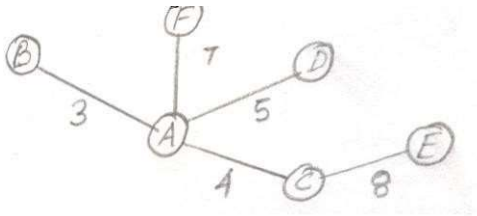


Fig.: (d)

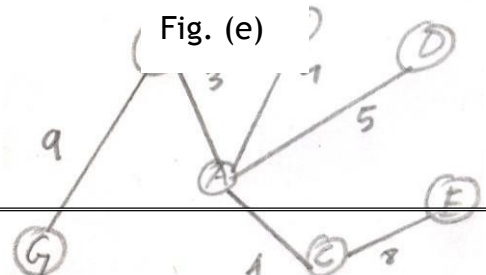
	edge	cost
G	BG	9
	CG	10
E	CE	8
	DE	9
	FE	11

Obvious choice is CE, shown in figure (e),



The only vertex left is G and we have the minimum cost edge that connects it to the tree constructed so for is BG. Therefore add it and the costs 9+3+7+5+4+8+36 and is given by the following figure,

Fig. (e)



ALGORITHM

1. T<-0

2. While T contains less than(n-1) edges and E not empty do

3. Choose an edge(V,W) from E of lowest cost

4. Delete (V,W) from E

5. if (V,W) does not create a cycle in T

6. then add(V,W) to T

7. else discard (V,W)

8. end

9. if T contains fewer than (n-1) edges then print (no spanning tree)

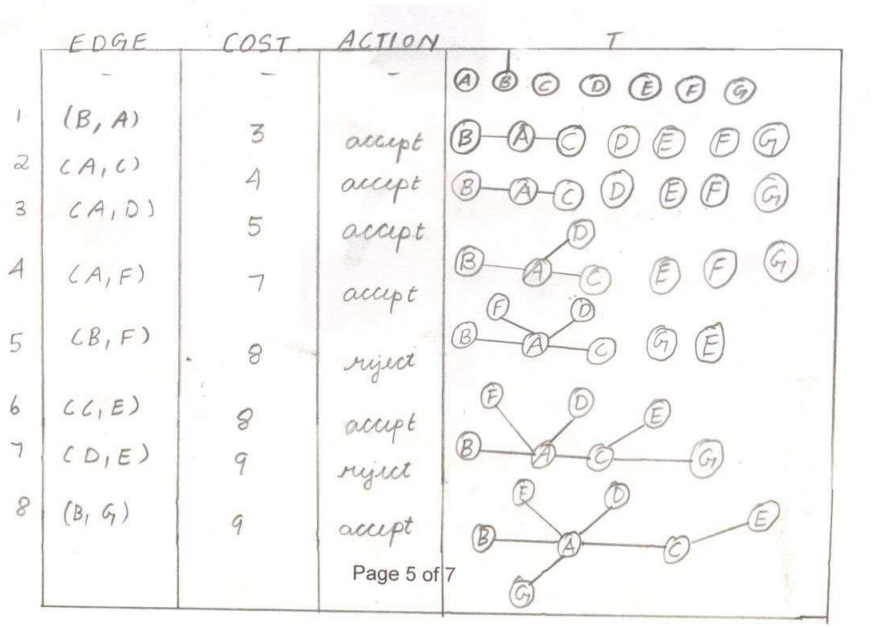
Initially, E is the set of all edges in G. In this set, we are going to (I) determining an edge with minimum cost (in line 3)

(ii) Delete that edge (line 4)

This can be done efficiently if the edges in E are maintained as a sorted sequential list.

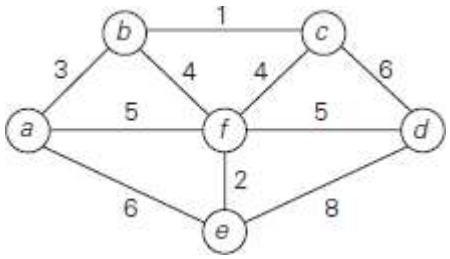
In order to be able to perform steps 5&6 efficiently, the vertices in G should be grouped together in such way that one may easily determine of the vertices V and W are already connected by the earlier selection of edges.

In case they are, then the edge (V,W) is to be discarded. If they are not then (V,W) is to be added to T. Our possible grouping is to place all vertices in the same connected components of T into the set. Then two vertices V,W are connected in t if they are in the same set.



Here for example when the edge (B,F) is to be considered, the sets would be {A,B,C,D,F},{E},{F}.vertices B and F are in the same set and so the edge (B,F) is rejected. The next edge to be considered is (C, E).since vertices C and E are in different sets the edge is accepted. This edge connects the two components {A, B, C, D, F} and {E} together and so these two sets should be joined to obtain the set representing the new component.

5. Explain Prim's Algorithmindetail and findthe Minimum Spanning Tree forthebelow graph.



Spanning tree:

In a Graph G(V,E), Spanning tree of 'G' is a selection of edges of G that form a tree spanning every vertex.(i.e) every vertex lies in the tree, but, no cycle are formed

Minimum Spanning Tree:

Minimum spanning tree is a spanning tree with weight less or equal to weight of every other spanning tree.

Algorithms used to find minimum spanning tree:

- Prim's Algorithm
- kruskal's algorithm

Algorithm:

To find minimum spanning tree T.

Step 1:

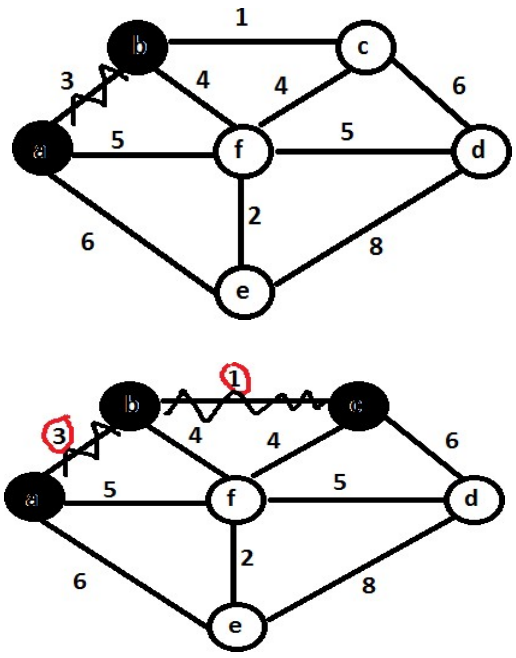
Select any node to be the first of T.

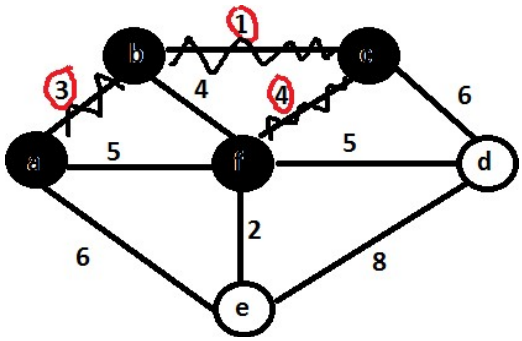
Step 2:

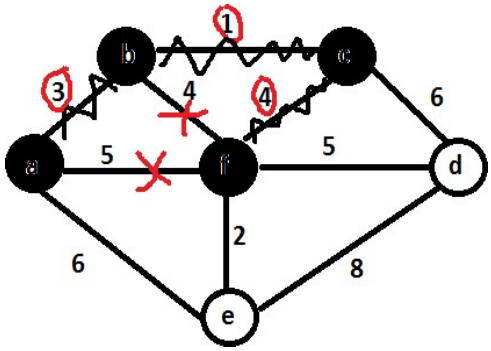
Consider which arcs connects nodes in T to nodes outside T. Pick one with minimum weight(if more than one, choose any). Add this arc and node to T.

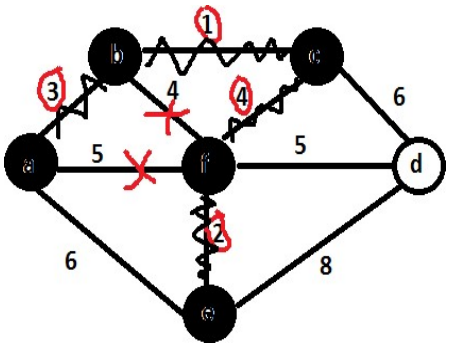
Step 3:

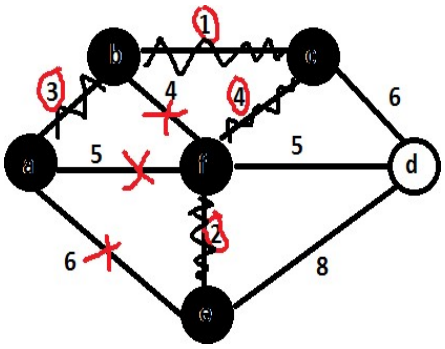
Repeat step 2 until T contains every node of the graph.











3 + 1 + 4 + 6 + 5 + 2 = 21.

Running time Performance:

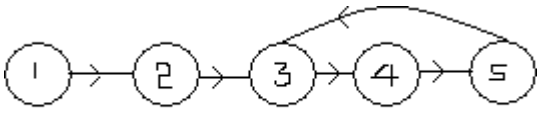
Adjacency matrix: O $|V^2|$

Binary heap: O (E log V)

Fibonacci heap: O (E + V log V)

6. Explain transitive closure with examples.

A problem related to the all pairs shortest path problem is that of determining for every pair of vertices *i,j* in *G* the existence of a path from *i* to *j*. Two cases are of interest, one when all path lengths (i.e., the number of edges on the path) are required to be positive and the other when path lengths are to be nonnegative. If *A* is the adjacency matrix of *G*, then the matrix A^+ having the property $A^+(i,j) = 1$ if there is a path of length > 0 from *i* to *j* and 0 otherwise is called the *transitive closure* matrix of *G*. The matrix A^* with the property $A^*(i,j) = 1$ if there is a path of length 0 from *i* to *j* and 0 otherwise is the *reflexive transitive closure* matrix of *G*.



a Digraph G

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

b Adjacency matrix A for G

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

c A^+

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

d A^*

Figure Graph G and Its Adjacency Matrix A, A^+ and A^*

Figure shows A^+ and A^* for a digraph. Clearly, the only difference between A^* and A^+ is in the terms on the diagonal. $A^+(i,i) = 1$ iff there a cycle of length > 1 containing vertex i while $A^*(i,i)$ is always one as there is a path of length 0 from i to i. If we use algorithm ALL_COSTS with COST$(i,j) = 1$ if $<i,j>$ is an edge in G and

COST$(i,j) = +$ if $<i,j>$ is not in G, then we can easily obtain A^+ from the final matrix A by letting $A^+ (i,j) = 1$ iff $A (i,j) < +\infty$. A^* can be obtained from A^+ by setting all diagonal elements equal 1. The total time is $O(n^3)$. Some simplification is achieved by slightly modifying the algorithm. In this modification the computation of line 9 of ALL_COSTS becomes $A(i,j)$ ☐ $A (i,j)$ **or** ($A(i,k)$ **and** $A(k,)$) and COST(i,j) is just the adjacencymatrix of G. With this modification, A need only be a bit matrix and then the final matrix A will be A^+.

7. Describe the applications of graph with example.

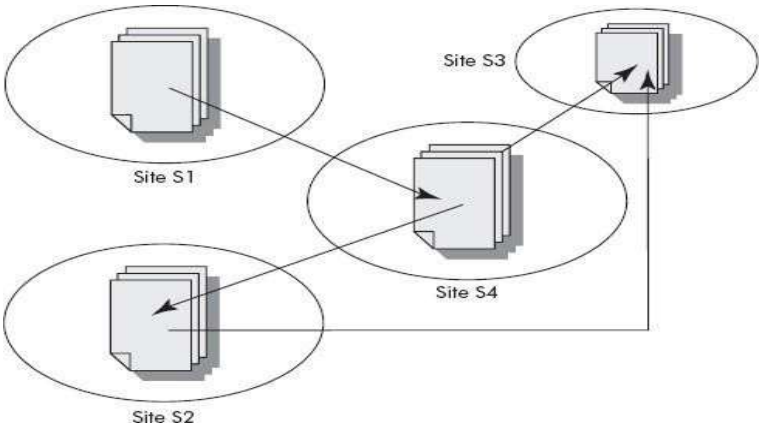
Structures that can be represented as graphs are ubiquitous, and many problems of practical interest can be represented by graphs. The link structure of a website could be represented by a directed graph: the vertices are the web pages available at the website and a directed edge from page A to page B exists if and only if A contains a link to B. A similar approach can be taken to problems in travel, biology, computer chip design, and many other fields. The development of algorithms to handle graphs is therefore of major interest in computer science. There, the transformation of graphs is often formalized and represented by graph rewrite systems. They are either directly used or properties of the rewrite systems(e.g. confluence) are studied.

A graph structure can be extended by assigning a weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pairwise connections have some numerical values. For example if a graph represents a road network, the weights could represent the length of each road. A digraph with weighted edges in the context of graph theory is called a network.

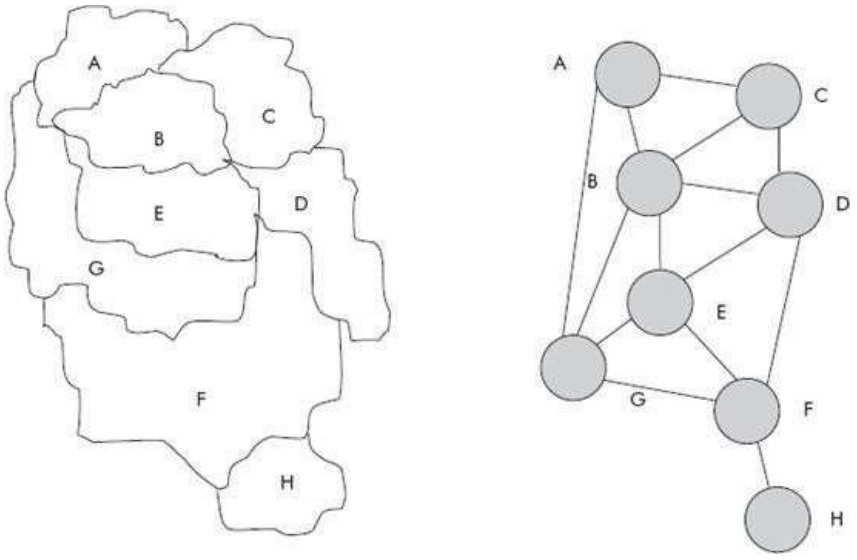
Networks have many uses in the practical side of graph theory, network analysis (for example, to model and analyze traffic networks). Within network analysis, the definition of the term "network" varies, and may often refer to a simple graph.

Model of www: The model of world wide web (www) can be represented by a graph (directed) wherein nodes denote the documents, papers, articles, etc. and the edges represent the outgoing hyperlinks between them as shown in

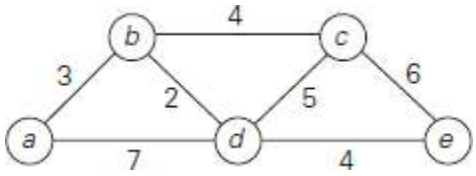
Figure



Coloring of maps: Coloring of maps is an interesting problem wherein it is desired that a map has to be colored in such a fashion that no two adjacent countries or regions have the same color. The constraint is to use minimum number of colors. A map can be represented as a graph wherein a node represents a region and an edge between two regions denote that the two regions are adjacent.



8. <u>Discuss in detail Dijikstra's algorithm and Find the shortest path for the below graph using the same Algorithm.</u>



Shortest Path: Theshortestpathproblemistheproblemoffindinga PATHbetween 2 nodes(vertices), such that the sum of weight of its constituent edge is minimized.

Eg: finding quickest way to go to one location to another.

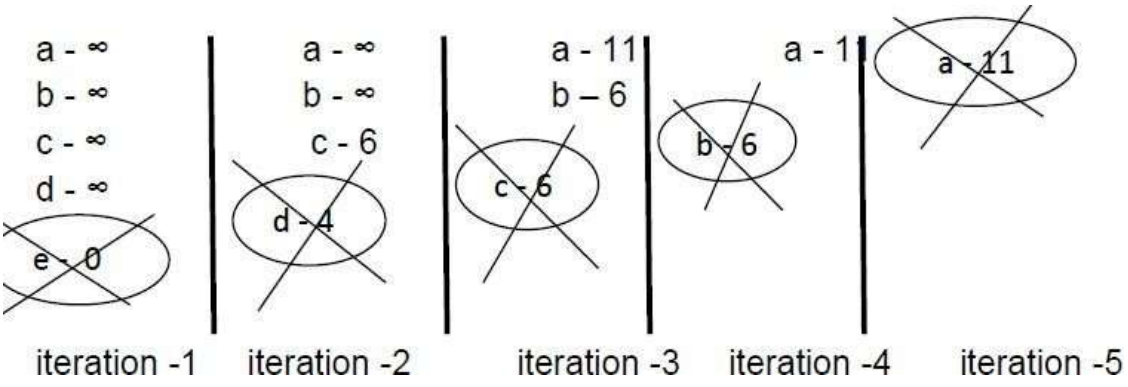
Single source shortest path: Finding shortest path from source vertex to all other vertex in the graph.

Dijkstra Algorithm :

```
1 function Dijkstra(Graph, source):
2for each vertex v in Graph: // Initializations
3dist[v] :=infinity; // Unknowndistance functionfrom
4 // source to v
5 previous[v] := undefined ; // Previous node in optimal path
6 end for // from source
7
8 dist[source] := 0 ; // Distance from source to source
9 Q := the set of all nodes in Graph ; // All nodes in the graph are
10 // unoptimized – thus are in Q
11 while Q is not empty: // The main loop
12u := vertex in Q with smallest distance in dist[] ; // Source node in first case
13 remove u from Q ;
14 ifdist[u] = infinity:
15 break ; // all remaining vertices are
16 end if // inaccessible from source
17
18 for each neighbor v of u: // where v has not yet been
19 // removed from Q.
20 alt :=dist[u] +dist_between(u, v);
21 if alt < dist[v]: // Relax (u,v,a)
22 dist[v] := alt ;
23 previous[v] := u ;
24decrease-key v in Q; // Reorder v in the Queue
25 end if
26 end for
27 end while
28 return dist;
29 endfunction
```

From the graph..‘e’ is selected as Source.

Min heap:



iteration	S	d[a]	d[b]	d[c]	d[d]	p[a]	p[b]	p[c]	p[d]
0	Ø	∞	∞	∞	∞	-	-	-	-
1	e = 0	∞	∞	0+6=6	0+4=4	-	-	e	e
2	D = 4	4+7 = 11	4+2=6	4+5=9 9>6, So, **6**	**4**	d	d	**e**	**e**
3	C = 6	**11**	**6**	**6**	**4**	**d**	**d**	**e**	**e**
4	B = 6	6+3=9 9< 11 So, 9	**6**	**6**	**4**	b	**d**	**e**	**e**
5	A = 11	**9**	**6**	**6**	**4**	**b**	**d**	**e**	**e**

Running Performance:

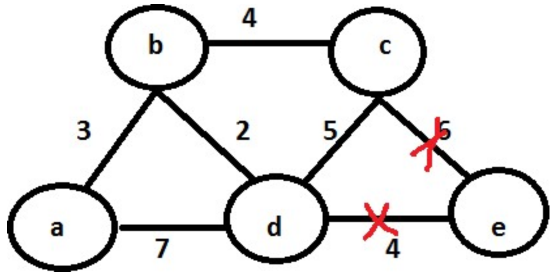
On array – O | V ² |

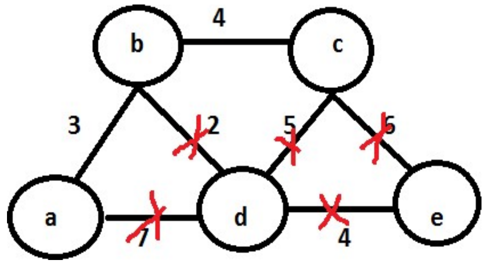
On binary heap – O (E + V log

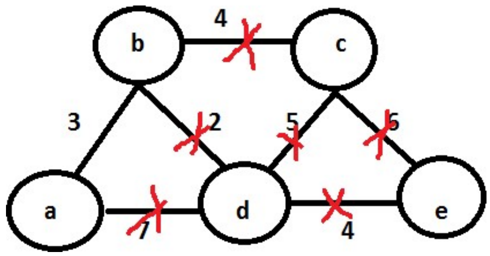
V) On Fibonacci heap – O (log

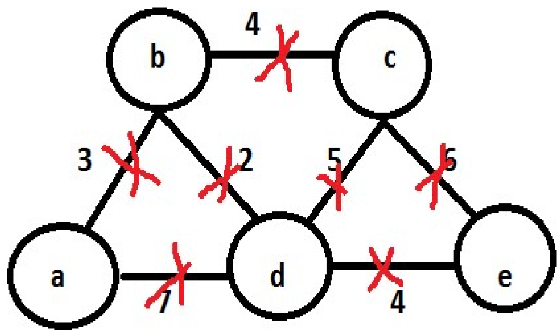
V) Where E – edges, V –

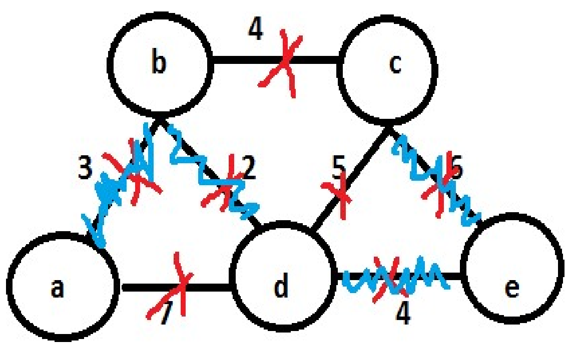
vertices











RESULT: e -> a = 9

e -> b = 6

e -> c = 6

e->d= 4

Important Program

1. SHORTEST PATH ALGORITHM - FLOYD'S ALGORITHM

```
#include<stdio.h>

void all_path();

int cost[20][20],n;

void main()

{

        int i,j;

        clrscr();

        printf("ALL PAIR SHORTEST PATH");

        printf("\nEnter the number of nodes");

        scanf("%d",&n);

        printf("\nEnter the cost matrix");

        for(i=1;i<=n;i++)

        {

        for(j=1;j<=n;j++)

             {

                    scanf("%d",&cost[i][j]);

              }

        }

        all_path();

        getch();

}

        void all_path()
```

```
{
        int i,j,k,a[20][20];
        for(i=1;i<=n;i++)
            {
               for(j=1;j<=n;j++)
                   {
                       a[i][j]=cost[i][j];
                   }
              }

         for(k=1;k<=n;k++)
                {
                     for(i=1;i<=n;i++)
                          {
                               for(j=1;j<=n;j++)
                                    {
                                         if(a[i][j]<(a[i][k]+a[k][j]))
                                              {
                                                   a[i][j]=a[i][j];
                                              }
                                         else
                                              {
                                                   a[i][j]=(a[i][k]+a[k][j]);
```

```
                                        }
                                    }
                                }
                            printf("STEP:%d\n",k);
                        for(i=1;i<=n;i++)
                            {
                                for(j=1;j<=n;j++)
                                    {
                                        printf(" %d",a[i][j]);
                                    }
                                printf("\n");
                            }
                    }
            }
```

2. SHORTEST PATH ALGORITHM - DIJKSTRA'S ALGORITHM

```
#include  <stdio.h>
#include <conio.h>
#define infinity 999


void dij(int n,int v,int cost[10][10],int dist[])
{
 int i,u,count,w,flag[10],min;
 for(i=1;i<=n;i++)
```

```
 flag[i]=0,dist[i]=cost[v][i];
 count=2;
 while(count<=n)
 {
      min=99;
          for(w=1;w<=n;w++)
               if(dist[w]<min && !flag[w])
                    min=dist[w],u=w;
                    flag[u]=1;
                    count++;
       for(w=1;w<=n;w++)
            if((dist[u]+cost[u][w]<dist[w]) && !flag[w])
                 dist[w]=dist[u]+cost[u][w];
  }
}

void main()
{
 int n,v,i,j,cost[10][10],dist[10];
 clrscr();
 printf("\n Enter the number of nodes:");
 scanf("%d",&n);
 printf("\n Enter the cost matrix:\n");
```

```
 for(i=1;i<=n;i++)


for(j=1;j<=n;j++)
  {
        scanf("%d",&cost[i][j]);
        if(cost[i][j]==0)
        cost[i][j]=infinity;
  }
 printf("\n Enter the source matrix:");
 scanf("%d",&v);
 dij(n,v,cost,dist);
 printf("\n Shortest path:\n");
 for(i=1;i<=n;i++)
         if(i!=v)
              printf("%d->%d,cost=%d\n",v,i,dist[i]);
              getch();
}
```

3. SPANNING TREE - KRUSKAL'S ALGORITHM

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
```

```
int find(int);
int uni(int,int);
void main()
{
 clrscr();
 printf("\n\n\tImplementation of Kruskal's algorithm\n\n");
 printf("\nEnter the no. of vertices\n");
 scanf("%d",&n);
 printf("\nEnter the cost adjacency matrix\n");
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=n;j++)
        {
             scanf("%d",&cost[i][j]);
             if(cost[i][j]==0)
             cost[i][j]=999;
        }
 }
 printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
 while(ne<n)
        {
              for(i=1,min=999;i<=n;i++)
                     {
```

```
                for(j=1;j<=n;j++)
                    {
                        if(cost[i][j]<min)
                            {

                                min=cost[i][j];
                                a=u=i;
                                b=v=j;
                            }
                    }
                }
            u=find(u);
            v=find(v);
            if(uni(u,v))
                {
                    printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
                    mincost +=min;
                }
            cost[a][b]=cost[b][a]=999;
    }
 printf("\n\tMinimum cost = %d\n",mincost);
 getch();
}
```

```
int find(int i)
{
        while(parent[i])
        i=parent[i];
        return i;
}
int uni(int i,int j)
{
 if(i!=j)
 {
  parent[j]=i;
  return 1;
 }return 0;
}
```

4. SPANNING TREE - PRIM'S ALGORITHM

```
#include<stdio.h>
#include<conio.h>
int a,b,u,v,n,i,j,ne=1;
int visited[10]={0},min,mincost=0,cost[10][10];
void main()
{
 clrscr();
 printf("\n Prim's  Algorithm");
 printf("\n Enter the number of nodes:");
```

```
scanf("%d",&n);
printf("\n Enter the adjacency matrix:\n");
for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
        {
             scanf("%d",&cost[i][j]);
             if(cost[i][j]==0)
             cost[i][j]=999;
         }
visited[1]=1;
printf("\n");
while(ne<n)
     {
         for(i=1,min=999;i<=n;i++)
             for(j=1;j<=n;j++)
               if(cost[i][j]<min)
                   if(visited[i]!=0)
                        {
                              min=cost[i][j];
                              a=u=i;
                              b=v=j;
                        }
                if(visited[u]==0 || visited[v]==0)
```

```
                {
                        printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
                        mincost+=min;
                        visited[b]=1;
                }
            cost[a][b]=cost[b][a]=999;
        }
 printf("\n Minimun cost=%d",mincost);
 getch();
}
```

5. GRAPH TRAVERSAL- BREATH FIRST SEARCH

```
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int v)
{
 for(i=1;i<=n;i++)
  if(a[v][i] && !visited[i])
         q[++r]=i;
 if(f<=r)
       {
          visited[q[f]]=1;
          bfs(q[f++]);
```

```
        }
    }
void main()
{
 int v;
 clrscr();
 printf("\nBreadth First Search");
 printf("\n Enter the number of vertices:");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
       {
              q[i]=0;
              visited[i]=0;
       }
 printf("\n Enter graph data in matrix form:\n");
 for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
         scanf("%d",&a[i][j]);
 printf("\n Enter the starting vertex:");
 scanf("%d",&v);
 bfs(v);
 printf("\n The node which are reachable are:\n");
 for(i=1;i<=n;i++)
```

```
        if(visited[i])
            printf("%d\t",i);
        else
            printf("\n Bfs is not possible");
        getch();
}
```

6. GRAPH TRAVERSAL- DEPTH FIRST SEARCH

```
#include<stdio.h>
#include<conio.h>
int a[20][20],reach[20],n;
void dfs(int v)
{
   int i;
   reach[v]=1;
        for(i=1;i<=n;i++)
           if(a[v][i] && !reach[i])
               {
                    printf("\n %d->%d",v,i);
                    dfs(i);
               }
}
void main()
{
 int i,j,count=0;
```

```
clrscr();
printf("Depth First Search");
printf("\n Enter number of vertices:");
scanf("%d",&n);
for(i=1;i<=n;i++)
      {
            reach[i]=0;
            for(j=1;j<=n;j++)
            a[i][j]=0;
     }
printf("\n Enter the adjacency matrix:\n");
for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
scanf("%d",&a[i][j]);
dfs(1);
printf("\n");
for(i=1;i<=n;i++)
{
 if(reach[i])
  count++;
}
if(count==n)
        printf("\n Graph is connected");
```

```
 else
        printf("\n Graph is not connected");
 getch();
}
```

7. BINARY TREE TRAVERSAL

```
struct Node
{
   int data;
   struct Node* left, *right;
   Node(int data)
   {
      this->data = data;
      left = right = NULL;
   }
}
void printPostorder(struct Node* node)
{
   if (node == NULL)
      return;

   // first recur on left subtree
   printPostorder(node->left);

   // then recur on right subtree
   printPostorder(node->right);
```

```
    // now deal with the node
    cout << node->data << " ";
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
    if (node == NULL)
        return;

    /* first print data of node */
```

```
    cout << node->data << " ";

    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    struct Node *root = new Node(1);
    root->left            = new Node(2);
    root->right         = new Node(3);
    root->left->left     = new Node(4);
    root->left->right = new Node(5);

    cout << "\nPreorder traversal of binary tree is \n";
    printPreorder(root);

    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);

    cout << "\nPostorder traversal of binary tree is \n";
    printPostorder(root);
```

```
    return 0;
}
```

Output:

```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```