UNIT – V: Searching and sorting: Sorting – An Introduction, Bubble Sort, Insertion Sort, Merge Sort, Searching – An Introduction, Linear or Sequential Search, Binary Search, Indexed Sequential Search

Graphs: Introduction to Graphs, Terms Associated with Graphs, Sequential Representation of Graphs, Linked Representation of Graphs, Traversal of Graphs, Spanning Trees, Shortest Path, Application of Graphs.

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, dictionaries in which words are arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort
4. Insertion sort and
5. Heap sort

There are two types of sorting techniques:

1. Internal sorting
2. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

5.1. LINEAR SEARCH

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is O(n).

Algorithm:

Let array a[n] stores n elements. Determine whether element 'x' is present or not.

```
linsrch(a[n], x)
{
        index = 0;
        flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
                }
                index ++;
        }
        if(flag == 1)
                printf("Data found  at %d position", index);
        else
                printf("data not found");

}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for:	45, we'll look at 1 element before success
	39, we'll look at 2 elements before success
	8, we'll look at 3 elements before success
	54, we'll look at 4 elements before success
	77, we'll look at 5 elements before success
	38 we'll look at 6 elements before success
	24, we'll look at 7 elements before success
	16, we'll look at 8 elements before success
	4, we'll look at 9 elements before success
	7, we'll look at 10 elements before success
	9, we'll look at 11 elements before success
	20, we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

1. Searching for x = 7	Search successful, data found at 3rd position
2. Searching for x = 82	Search successful, data found at 7th position
3. Searching for x = 42	Search un-successful, data not found

5.2. BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid]. Similarly, if a[mid] > x, then further search is only necessary in that part of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

Algorithm:

Let array a[n] of elements in increasing order, $n \geq 0$, determine whether 'x' is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)
{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid = ⌊ (low + high)/2 ⌋
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid + 1;
                else return mid;
        }
        return 0;
}
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, ***found***

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, ***found***

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8, ***found***

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, ***found***

If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20

low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, ***found***

If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, ***found***

If we are searching for x = 24: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 10, mid = 17/2 = 8, check 38
low = 7, high = 7, mid = 14/2 = 7, check 24, ***found***

If we are searching for x = 38: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 10, mid = 17/2 = 8, check 38, ***found***

If we are searching for x = 39: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, ***found***

If we are searching for x = 45: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 10, high = 10, mid = 20/2 = 10, check 45, ***found***

If we are searching for x = 54: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54, ***found***

If we are searching for x = 77: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, ***found***
The number of comparisons necessary by search element:

> 20 – requires 1 comparison; 8 and 39 – requires 2 comparisons;
> 4, 9, 38, 54 – requires 3 comparisons; and 7, 16, 24, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

5.3. Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements)

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared)

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared)

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared)

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks

11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared)
We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.
For an array of size n, we required (n-1) passes.

Algorithm:

```
void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    // print the sorted array
    for(i = 0; i < n; i++)
    {
        Display arr[i];
    }
}
```

Time Complexity:
The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is (n-1) * (n-1) = $n^2 - 2n + 1$, which is $O(n^2)$. Therefore bubble sort is very inefficient when there are more elements to sorting.

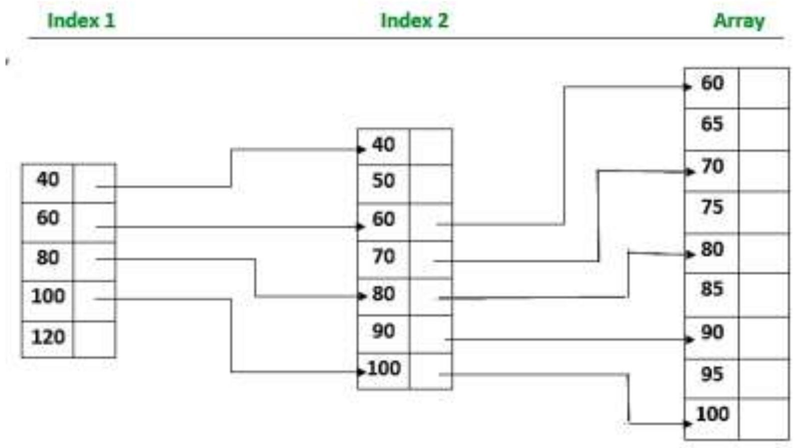
5.4 Indexed Sequential Search
In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time cause it is located in a specified group.
Note: When the user makes a request for specific records it will find that index group first where that specific record is recorded.

Characteristics of Indexed Sequential Search:

- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- The index is searched 1st then the array and guides the search in the array.

Note: Indexed Sequential Search actually does the indexing multiple time, like creating the index of an index.

Explanation by diagram “Indexed Sequential Search”:



CODE:

```
// C program for Indexed Sequential Search
#include <stdio.h>
#include <stdlib.h>
void indexedSequentialSearch(int arr[], int n, int k)
{
    int elements[20], indices[20], temp, i, set = 0;
    int j = 0, ind = 0, start, end;
    for (i = 0; i < n; i += 3) {
                        // Storing element
        elements[ind] = arr[i];
                        // Storing the index
        indices[ind] = i;
        ind++;
    }
    if (k < elements[0]) {
        printf("Not found");
        exit(0);
    }
    else {
        for (i = 1; i <= ind; i++)
            if (k <= elements[i]) {
                start = indices[i - 1];
                end = indices[i];
                set = 1;
                break;
            }
    }
    if (set == 0) {
        start = indices[i - 1];
        end = n;
    }
    for (i = start; i <= end; i++) {
        if (k == arr[i]) {
            j = 1;
            break;
        }
    }
    if (j == 1)
        printf("Found at index %d", i);
    else
        printf("Not found");
}
                              // Driver code
void main()
{

    int arr[] = { 6, 7, 8, 9, 10 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Element to search
    int k = 8;
    indexedSequentialSearch(arr, n, k);
}
```
Output:
```
Found at index 2
```

5.6 Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements)
We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared)
We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared)
We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared)
We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared)
We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.
For an array of size n, we required (n-1) passes.
Algorithm:
```
void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    // print the sorted array
    for(i = 0; i < n; i++)
    {
        Display arr[i];
    }
}
```

Time Complexity:
The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is (n-1) * (n-1) = n^2 – 2n + 1, which is O(n^2). Therefore bubble sort is very inefficient when there are more elements to sorting.

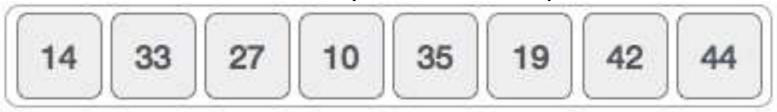
5.7 Insertion Sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort.**

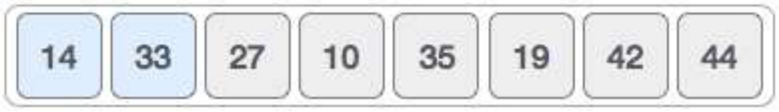
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where n is the number of items.

How Insertion Sort Works?

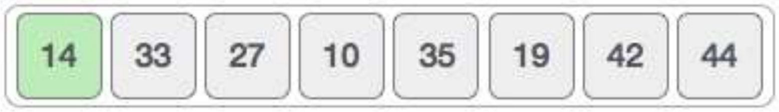
We take an unsorted array for our example.



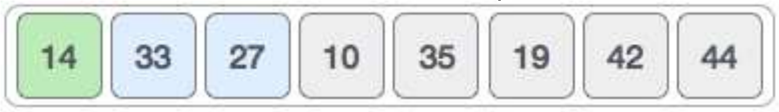
Insertion sort compares the first two elements.



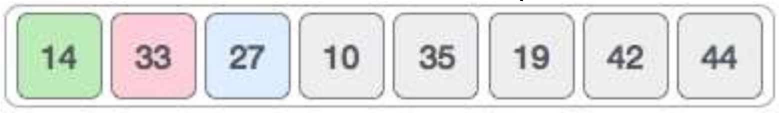
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



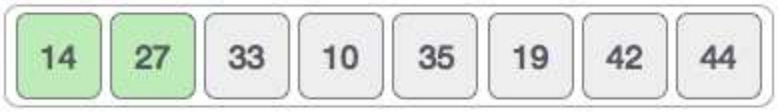
Insertion sort moves ahead and compares 33 with 27.



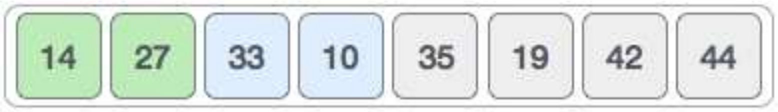
And finds that 33 is not in the correct position.



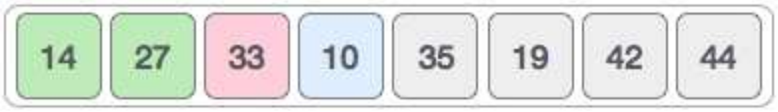
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



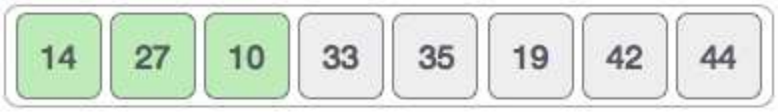
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



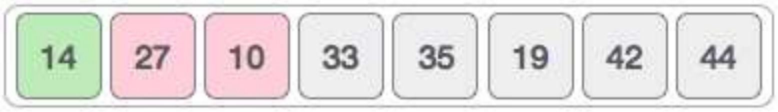
These values are not in a sorted order.



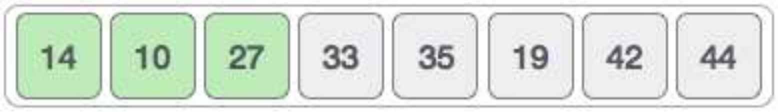
So we swap them.



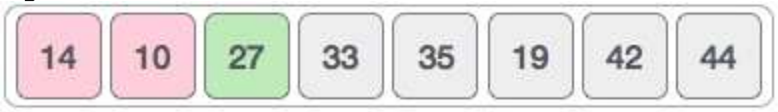
However, swapping makes 27 and 10 unsorted.



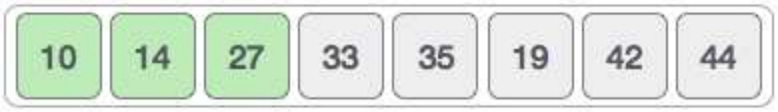
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted
```

Pseudocode

procedure insertionSort(A[n])

```
{
   int holePosition, valueToInsert

   for i = 1 to length(A)
{
     /* select value to be inserted */
      valueToInsert = A[i]
      holePosition = i
       /*locate hole position for the element to be inserted */
      while (holePosition > 0) and A[holePosition-1] > valueToInsert)
     {
         A[holePosition] = A[holePosition-1]
         holePosition = holePosition -1
      }
      /* insert the number at hole position */
      A[holePosition] = valueToInsert
    }
   }
```

5.8 Merge Sort Algorithm

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

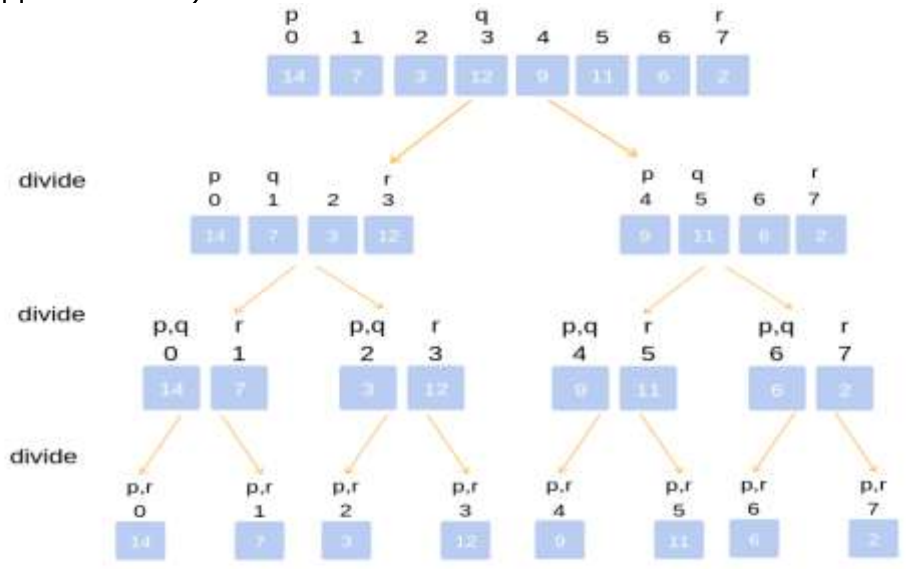
A merge sort works as follows:

Top-down Merge Sort Implementation:

The top-down merge sort approach is the methodology which uses recursion mechanism. It starts at the Top and proceeds downwards, with each recursive turn asking the same question such as "What is required to be done to sort the array?" and having the answer as, "split the array into two, make a recursive call, and merge the results.", until one gets to the bottom of the array-tree.

Example: Let us consider an example to understand the approach better.

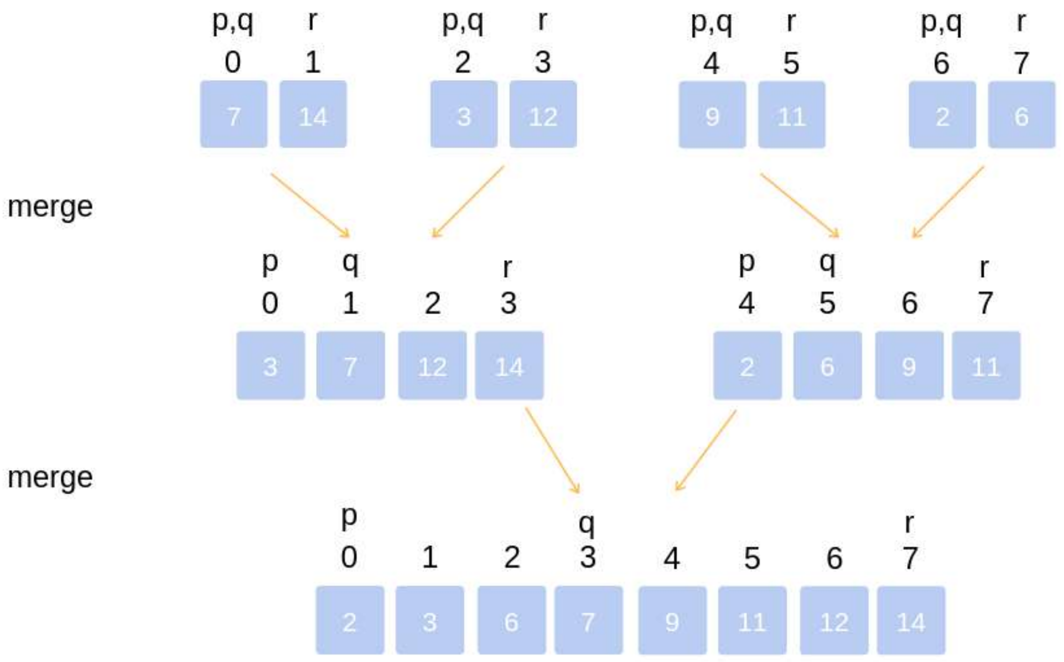
Divide the unsorted list into n sublists, each comprising 1 element (a list of 1 element is supposed sorted).



Working of Merge Sort

Top-down Implementation

Repeatedly merge sublists to produce newly sorted sublists until there is only 1 sublist remaining. This will be the sorted list.
Merging of two lists done as follows:
The first element of both lists is compared. If sorting in ascending order, the smaller element among two becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the newly combined sublist covers all the elements of both the sublists.



Implementation Of Merge Sort

```
void merge(int Arr[], int start, int mid, int end)
        {
                int temp[end - start + 1];
                int i = start, j = mid+1, k = 0;
                while(i <= mid && j <= end)
                {
                        if(Arr[i] <= Arr[j])
                        {
                                temp[k] = Arr[i];
                                k += 1;
                                i += 1;
                        }
                        else
                        {
                                temp[k] = Arr[j];
                                k += 1;
                                j += 1;
                }
        }
        while(i <= mid)
        {
                temp[k] = Arr[i];
                k += 1; i += 1;
        }

                while(j <= end)
        {
                temp[k] = Arr[j];
                k += 1; j += 1;
        }
                for(i = start; i <= end; i += 1)
        {
                Arr[i] = temp[i - start]
        }
}

// Arr is an array of integer type
// start and end are the starting and ending index of current interval of Arr
void mergeSort(int *Arr, int start, int end)
{
        if(start < end)
       {
                int mid = (start + end) / 2;
                mergeSort(Arr, start, mid);
                mergeSort(Arr, mid+1, end);
                merge(Arr, start, mid, end);
        }
}
```

Graphs: Introduction to Graphs, Terms Associated with Graphs, Sequential Representation of Graphs, Linked Representation of Graphs, Traversal of Graphs, Spanning Trees, Shortest Path, Application of Graphs.

Graphs:

Graph G is a pair (V, E), where V is a finite set of vertices and E is a finite set of edges. We will often denote $n = |V|$, $e = |E|$.

A graph is generally displayed as figure 7.5.1, in which the vertices are represented by circles and the edges by lines.

An edge with an orientation (i.e., arrow head) is a directed edge, while an edge with no orientation is our undirected edge.

If all the edges in a graph are undirected, then the graph is an undirected graph. The graph of figures 7.5.1(a) is undirected graphs. If all the edges are directed; then the graph is a directed graph. The graph of figure 7.5.1(b) is a directed graph. A directed graph is also called as digraph.

A graph G is connected if and only if there is a simple path between any two nodes in G.

A graph G is said to be complete if every node a in G is adjacent to every other node v in G. A complete graph with n nodes will have $n(n-1)/2$ edges. For example, Figure 7.5.1.(a) and figure 7.5.1.(d) are complete graphs.

A directed graph G is said to be connected, or strongly connected, if for each pair u, v for nodes in G there is a path from u to v and there is a path from v to u. On the other hand, G is said to be unilaterally connected if for any pair u, v of nodes in G there is a path from u to v or a path from v to u. For example, the digraph shown in figure 7.5.1 (e) is strongly connected.

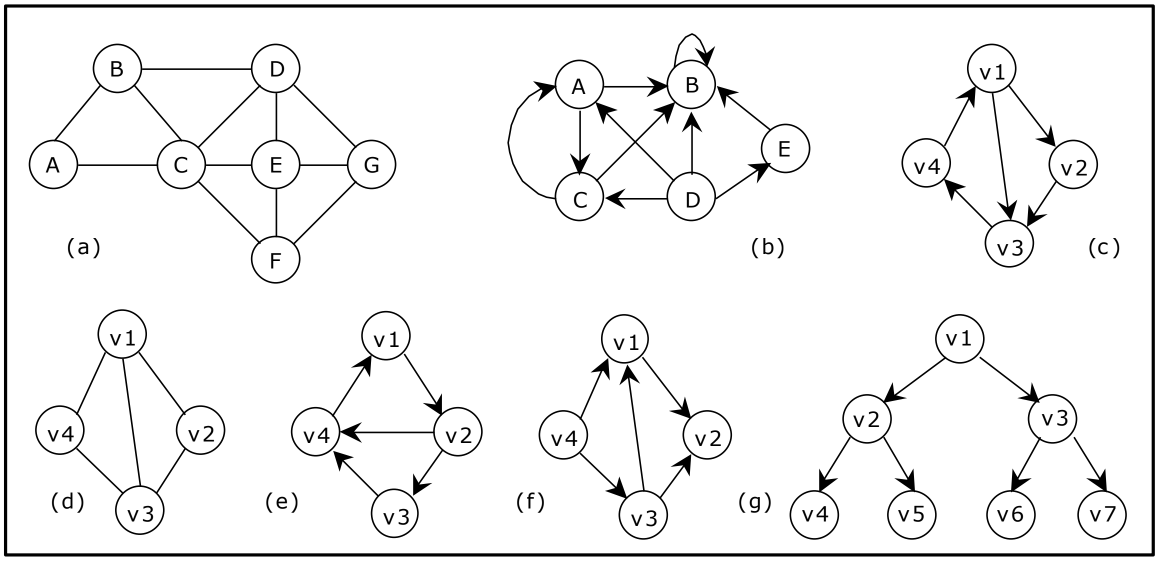


Figure 7.5.1 Various Graphs

We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called weighted graph.

The number of incoming edges to a vertex v is called in–degree of the vertex (denote indeg(v)). The number of outgoing edges from a vertex is called out-degree (denote outdeg(v)). For example, let us consider the digraph shown in figure 7.5.1(f),

$$\text{indegree}(v_1) = 2 \qquad \text{outdegree}(v_1) = 1$$

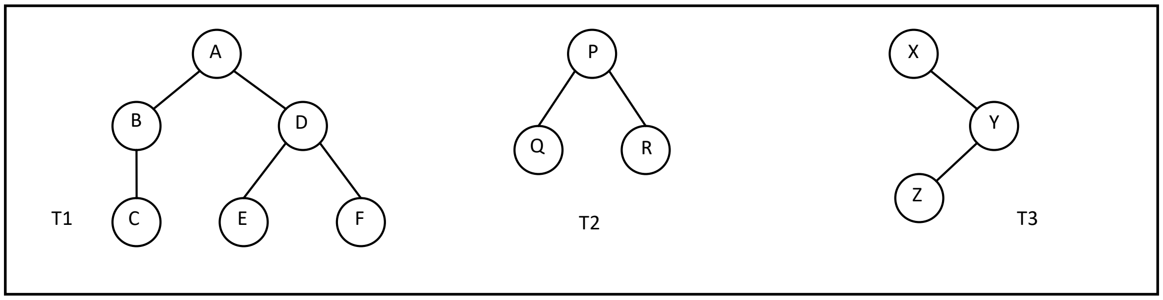
$$\text{indegree}(v_2) = 2 \qquad \text{outdegree}(v_2) = 0$$

A path is a sequence of vertices $(v_1, v_2, \ldots\ldots, v_k)$, where for all i, $(v_i, v_{i+1}) \varepsilon E$. A path is simple if all vertices in the path are distinct. If there a path containing one or more edges which starts from a vertex V_i and terminates into the same vertex then the path is known as a cycle. For example, there is a cycle in figure 7.5.1 (a), figure 7.5.1 (c) and figure 7.5.1 (d).

If a graph (digraph) does not have any cycle then it is called **acyclic graph**. For example, the graphs of figure 7.5.1 (f) and figure 7.5.1 (g) are acyclic graphs.

A graph $G' = (V', E')$ is a sub-graph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

A **Forest** is a set of disjoint trees. If we remove the root node of a given tree then it becomes forest. The following figure shows a forest F that consists of three trees T1, T2 and T3.



A Forest F

A graph that has either self loop or parallel edges or both is called **multi-graph**.

Tree is a connected acyclic graph (there aren't any sequences of edges that go around in a loop). A spanning tree of a graph G = (V, E) is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

Let T be a spanning tree of a graph G. Then

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T, then T becomes disconnected.*
3. *If we add any edge into T, then the new graph will contain a cycle.*
4. *Number of edges in T is n-1.*

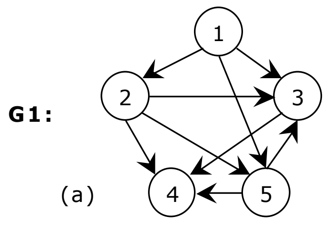
Representation of Graphs:
There are two ways of representing digraphs. They are:

- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

Adjacency matrix:
In this representation, the adjacency matrix of a graph G is a two dimensional n x n matrix, say A = ($a_{i,j}$), where

$$a_{i,j} = \begin{cases} 1 & \textit{if there is an edge from } v_i \textit{ to } v_j \\ 0 & \textit{otherwise} \end{cases}$$

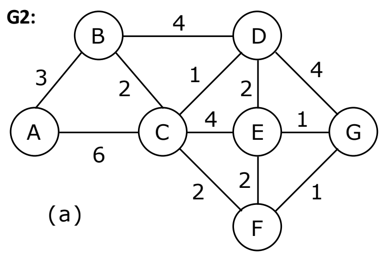
The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed. This matrix is also called as Boolean matrix or bit matrix.



	1	**2**	**3**	**4**	**5**
1	0	1	1	0	1
2	0	0	1	1	1

Figure 7.5.2. A graph and its Adjacency matrix

Figure 7.5.2(b) shows the adjacency matrix representation of the graph G1 shown in figure 7.5.2(a). The adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 0 or 1, the entry will be between number of edges between two vertices.
In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix. Figure 7.5.3(b) shows the cost adjacency matrix representation of the graph G2 shown in figure 7.5.3(a).

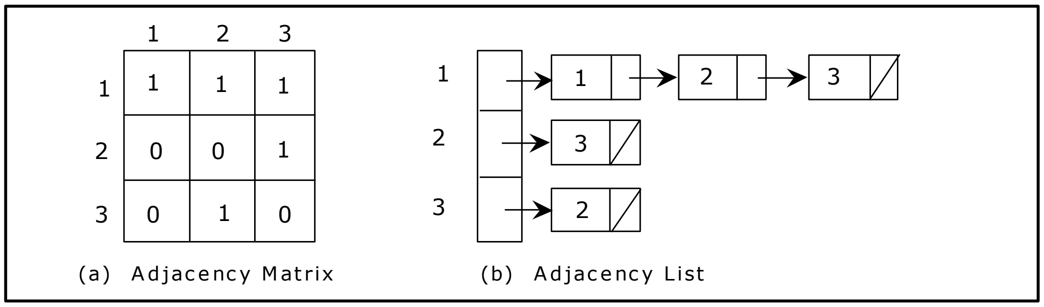


(b)

	A	**B**	**C**	**D**	**E**	**F**	**G**
A	0	3	6	∝	∝	∝	∝
B	3	0	2	4	∝	∝	∝
C	6	2	0	1	4	2	∝

Figure 7.5.3 Weighted graph and its Cost adjacency matrix

Adjacency List:
In this representation, the n rows of the adjacency matrix are represented as n linked lists. An array Adj[1, 2, ... n] of pointers where for 1 $\le$ v $\le$ n, Adj[v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements. For the graph G in figure 7.5.4(a), the adjacency list in shown in figure 7.5.4 (b).



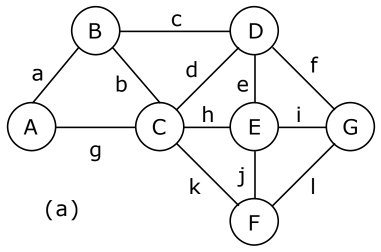
$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 \end{array}$$

Figure 7.5.4 Adjacency matrix and adjacency list

Incidence Matrix:
In this representation, if G is a graph with n vertices, e edges and no self loops, then incidence matrix A is defined as an n by e matrix, say A = ($a_{i,j}$), where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge } j \text{ incident to } v_i \\ 0 & \text{otherwise} \end{cases}$$

Here, n rows correspond to n vertices and e columns correspond to e edges. Such a matrix is called as vertex-edge incidence matrix or simply incidence matrix.



(b)

	a	b	c	d	e	f	g	h	i	j	k	l
A	1	0	0	0	0	0	1	0	0	0	0	0
B	1	1	1	0	0	0	0	0	0	0	0	0
C	0	1	0	1	0	0	1	1	0	0	1	0

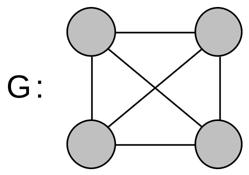
Figure 7.5.4 Graph and its incidence matrix

Figure 7.5.4(b) shows the incidence matrix representation of the graph G1 shown in figure 7.5.4(a).

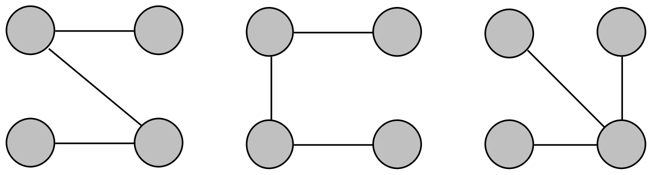
Minimum Spanning Tree (MST):
A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree w(T) is the sum of weights of all edges in T. Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

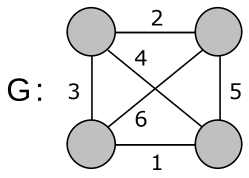
Example:



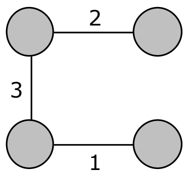
A graph G:



Three (of many possible) spanning trees from graph G:



A weighted graph G:



The minimal spanning tree from weighted graph G:

Let's consider a couple of real-world examples on minimum spanning tree:

- One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
- Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

Minimum spanning tree, can be constructed using any of the following two algorithms:

1. Kruskal's algorithm and
2. Prim algorithm.

Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost.

The order in which the edges are chosen, in this case, does not matter. Different MST's may result, but they will all have the same total cost, which will always be the minimum cost.

Kruskal's Algorithm for minimal spanning tree is as follows:

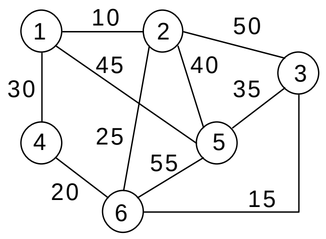
1. Make the tree T empty.
2. Repeat the steps 3, 4 and 5 as long as T contains less than n - 1 edges and E is not empty otherwise, proceed to step 6.

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 *then* Add (v, w) to T

   else discard (v, w)

6. If T contains fewer than n - 1 edges then print no spanning tree.

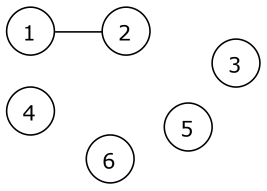
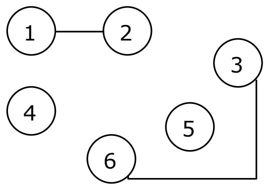
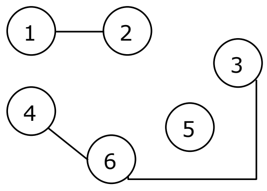
Example 1:
Construct the minimal spanning tree for the graph shown below:

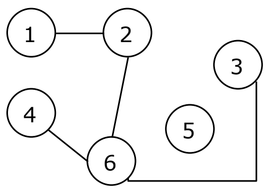
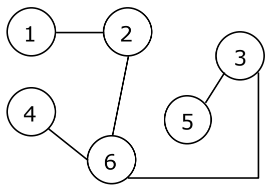


Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal's algorithm for minimal spanning tree is as follows:

Edge	Cost	Stages in Kruskal's algorithm	Remarks
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.

(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is **105**.

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Prim’s Algorithm:

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

The algorithm is given as follows.

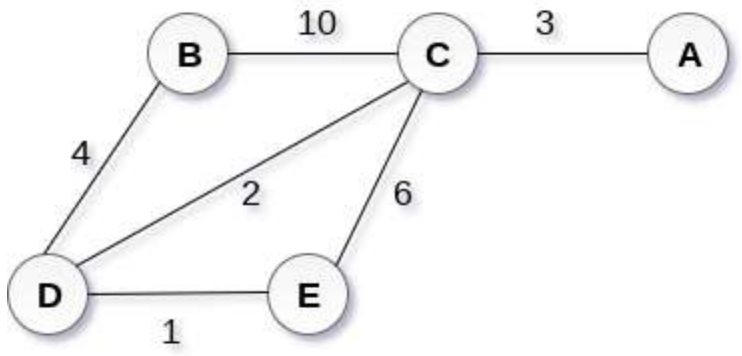
Algorithm

- **Step 1:** Select a starting vertex
- **Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
- **Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
- **Step 4:** Add the selected edge and the vertex to the minimum spanning tree T [END OF LOOP]

- **Step 5:** EXIT

Example :

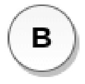
Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.



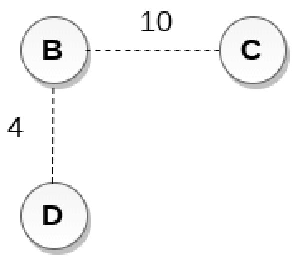
Solution

- **Step 1 :** Choose a starting vertex B.
- **Step 2:** Add the vertices that are adjacent to A. the edges that connecting the vertices are shown by dotted lines.
- **Step 3:** Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.
- **Step 3:** Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e. E and A.
- **Step 4:** Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

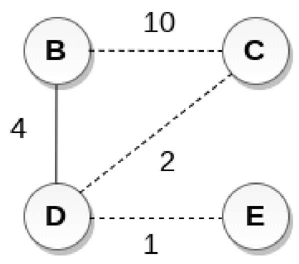
The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.
The cost of MST will be calculated as;
cost(MST) = 4 + 2 + 1 + 3 = 10 units.



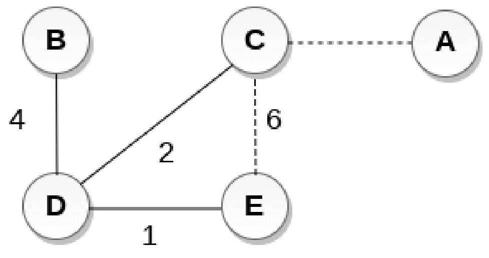
Step 1



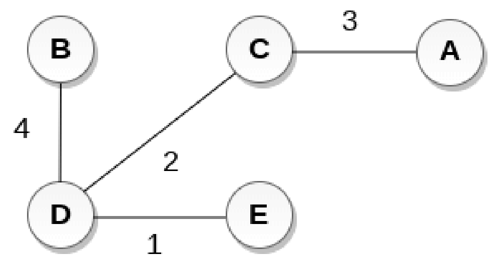
Step 2



Step 3



Step 4



Step 5

7.6.3. Traversing a Graph:

Many graph algorithms require one to systematically examine the nodes and edges of a graph G. There are two standard ways to do this. They are:

- Breadth first traversal (BFT)
- Depth first traversal (DFT)

The BFT will use a queue as an auxiliary structure to hold nodes for future processing and the DFT will use a STACK.
During the execution of these algorithms, each node N of G will be in one of three states, called the *status* of N, as follows:

1. STATUS = 1 (Ready state): The initial state of the node N.
2. STATUS = 2 (Waiting state): The node N is on the QUEUE or STACK, waiting to be processed.
3. STATUS = 3 (Processed state): The node N has been processed.

Both BFS and DFS impose a tree (the BFS/DFS tree) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. The spanning trees obtained using depth first searches are called depth first spanning trees. The spanning trees obtained using breadth first searches are called Breadth first spanning trees.

Breadth first search and traversal:
The general idea behind a breadth first traversal beginning at a starting node A is as follows. First we examine the starting node A. Then we examine all the neighbors of A. Then we examine all the neighbors of neighbors of A. And so on. We need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a QUEUE to hold nodes that are waiting to be processed, and by using a field STATUS that tells us the current status of any node. The spanning trees obtained using BFS are called Breadth first spanning trees.

Breadth first traversal algorithm on graph G is as follows:
This algorithm executes a BFT on graph G beginning at a starting node A.
Initialize all nodes to the ready state (STATUS = 1).

1. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).

2. Repeat the following steps until QUEUE is empty:

    a. Remove the front node N of QUEUE. Process N and change the status of N to the processed state (STATUS = 3).

b. Add to the rear of QUEUE all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

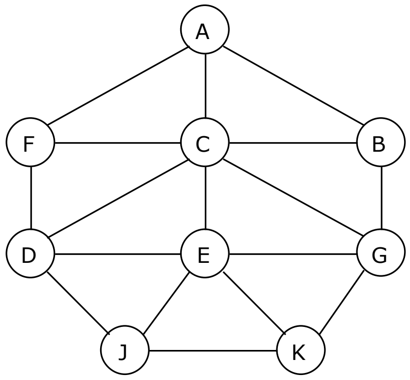
3. Exit.

Depth first search and traversal:
Depth first search of undirected graph proceeds as follows: First we examine the starting node V. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'U' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.
This algorithm is similar to the inorder traversal of binary tree. DFT algorithm is similar to BFT except now use a STACK instead of the QUEUE. Again field STATUS is used to tell us the current status of a node.
The algorithm for depth first traversal on a graph G is as follows.
This algorithm executes a DFT on graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).
2. Push the starting node A into STACK and change its status to the waiting state (STATUS = 2).
3. Repeat the following steps until STACK is empty:
    a. Pop the top node N from STACK. Process N and change the status of N to the processed state (STATUS = 3).
    b. Push all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
4. Exit.

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



A Graph G

Node	***Adjacency List***
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	*B*	*C*	*D*	*E*	*F*	*G*	*J*	*K*
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: ***A F C B D E G J K***.

Depth-first search and traversal:

The steps involved in depth first traversal are as follows:

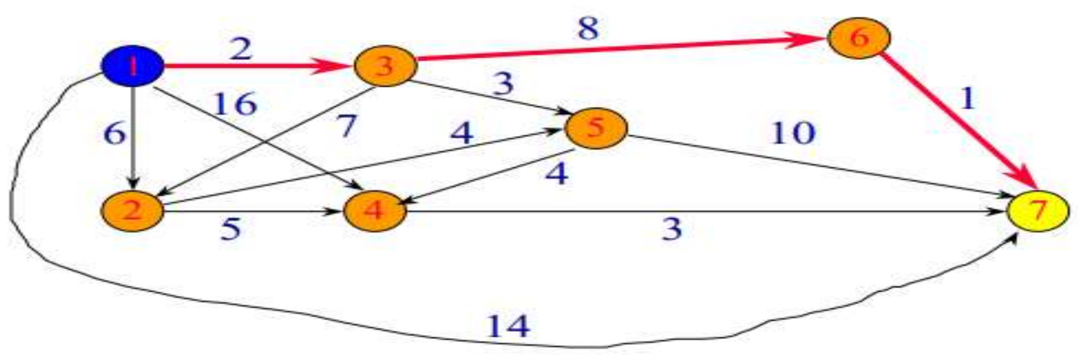
Current Node	Stack	Processed Nodes	Status								
			A	*B*	*C*	*D*	*E*	*F*	*G*	*J*	*K*
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: ***A F D J K G E C B***.

Shortest path:
We are given a graph **G** in which every edge has a weight attached, and our problem is to find a path from one vertex **V** to another **W** such that the sum of the weights on the path is small as possible. We call such a path a shortest path.
There are many different variations of the shortest path problem. They vary with respect to the specifications of the start vertex and end vertex. Some of the commonly known variants are listed below.

- The shortest path from a specified source vertex to a specified destination vertex.
- The shortest path from one specified vertex to all other vertices. This problem is also known as single source shortest path problem.
- The shortest path between all possible source and destination vertices. This problem is also known as all pairs shortest path problem.
- The shortest path can be determined using kruskal's and prim's algorithm.

Example



Another path from 1 to 7.
Path length is 11.

Applications of Graphs:

Social network graphs: Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

Transportation networks. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

Utility graphs. The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.

Document link graphs. The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

Protein-protein interactions graphs. Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process.

Graphs in epidemiology. Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

Graphs in compilers. Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.