UNIT IV - data structure notes r18 jntuh

Computer Science and Engineering (Jawaharlal Nehru Technological University, Hyderabad)

UNIT IV
Graphs: Graph Implementation Methods. Graph Traversal Methods. (DFS,BFS)
Sorting: Heap Sort, External Sorting- Model for external sorting, Merge Sort

Introduction to Graphs

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

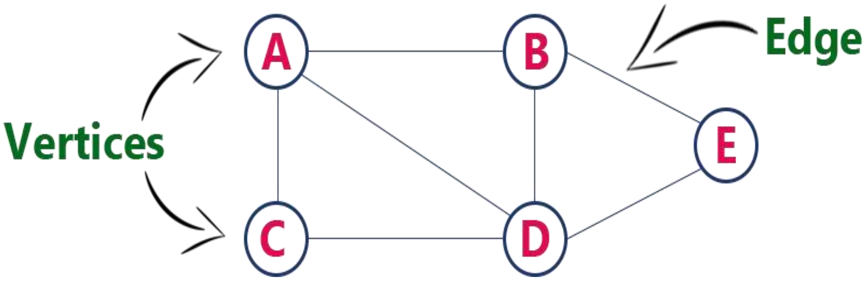
Graph is a collection of vertices and arcs in which vertices are connected with arcs

Graph is a collection of nodes and edges in which nodes are connected with edges

Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example

The following is a graph with 5 vertices and 6 edges.
This graph G can be defined as G = (V , E)
Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



DFS (A-

BFS

Graph Terminology

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

Vertex

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

Edge

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

Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

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

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with both undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by edge are called end vertices (or endpoints) of that edge.

Origin

If a edge is directed, its first endpoint is said to be the origin of it.

Destination

If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

Incident

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

Outgoing Edge

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

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

Self-loop

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

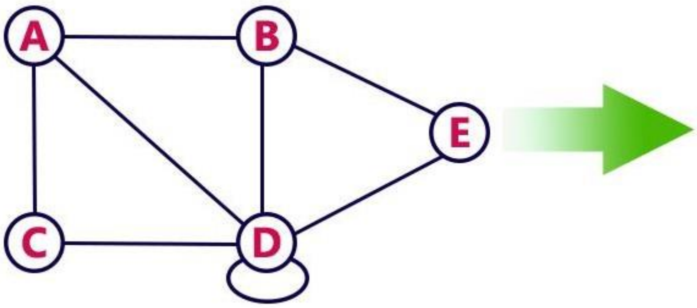
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

Adjacency Matrix

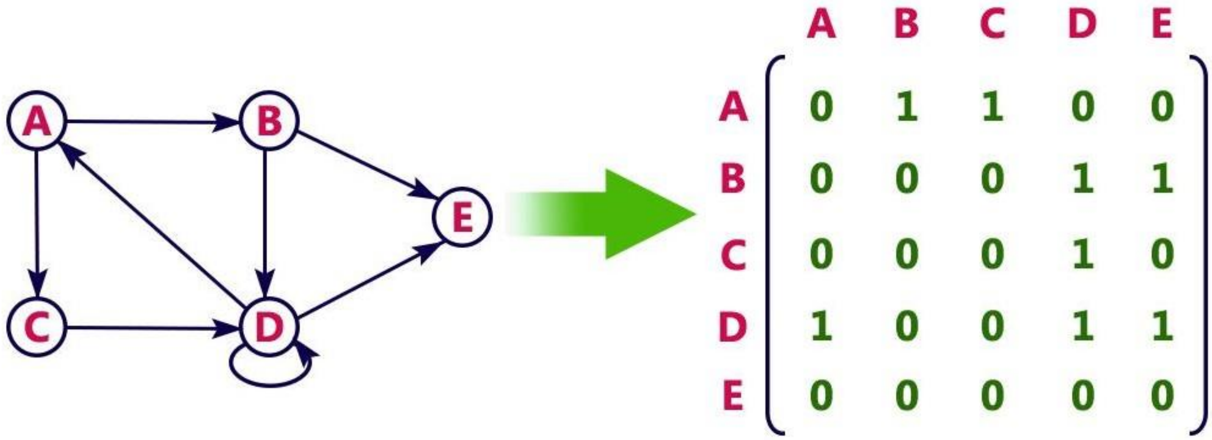
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph representation...

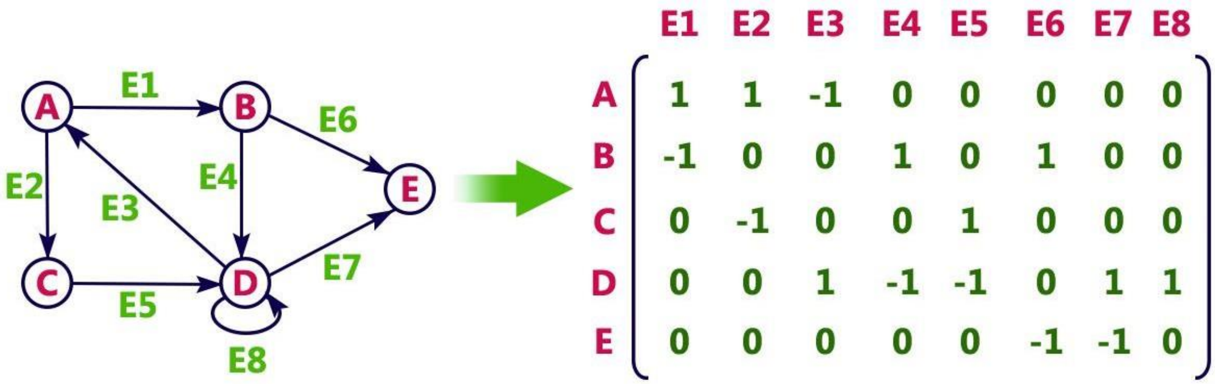


$$\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 1 & 0 & 0 \\
B & 0 & 0 & 0 & 1 & 1 \\
C & 0 & 0 & 0 & 1 & 0 \\
D & 1 & 0 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0
\end{array}$$

Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

For example, consider the following directed graph representation...

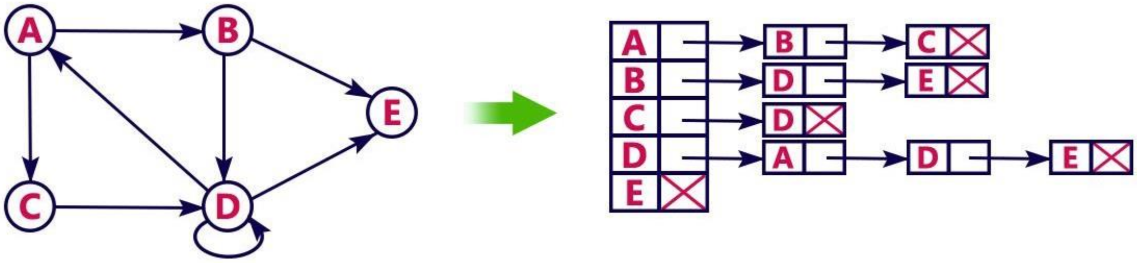


$$\begin{array}{c|cccccccc}
 & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\
\hline
A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\
D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0
\end{array}$$

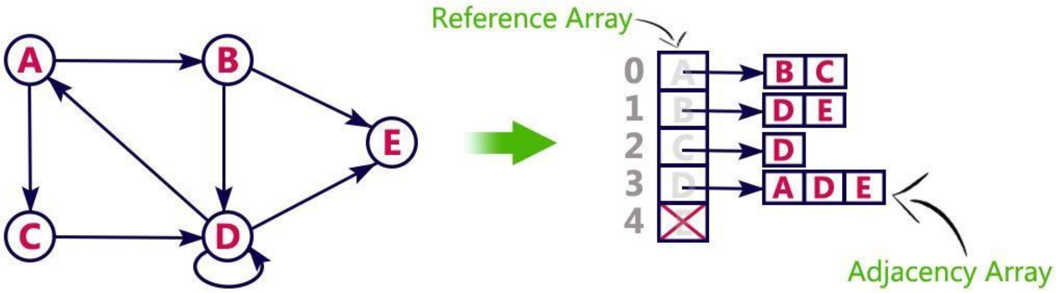
Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



Graph Traversal

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

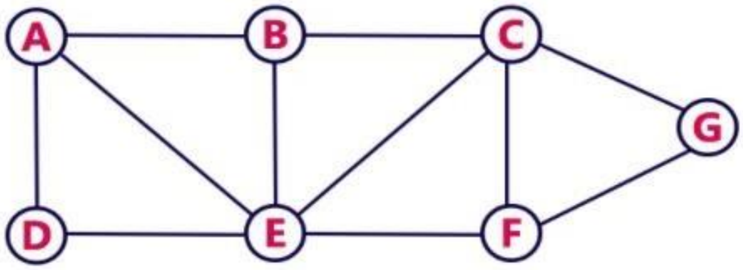
- Step 1 - Define a Stack of size total number of vertices in the graph.
- Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
- Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.
- Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

- Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.
- Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.
- Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we reached the current vertex.

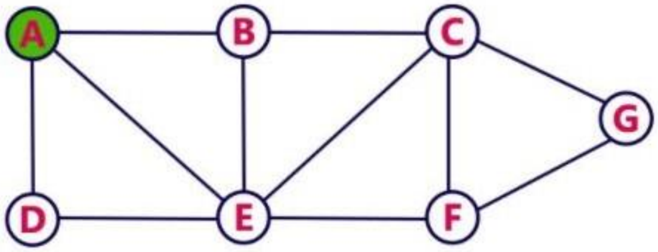
Example

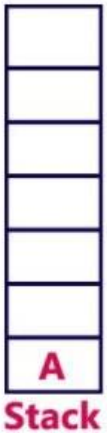
Consider the following example graph to perform DFS traversal



Step 1:

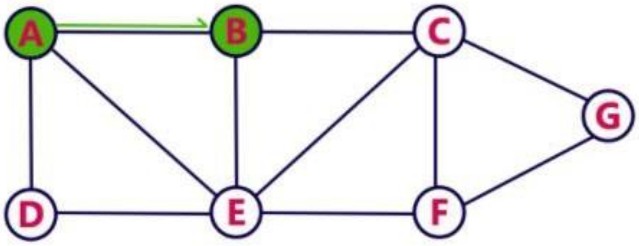
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

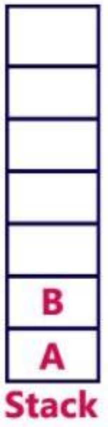




Step 2:

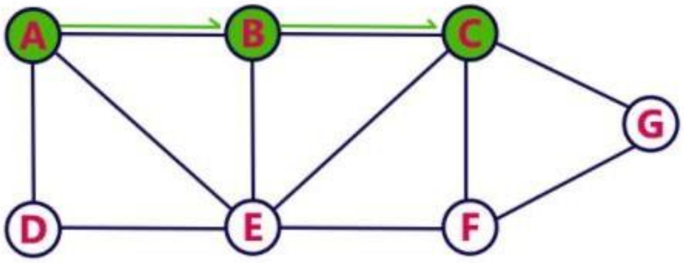
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

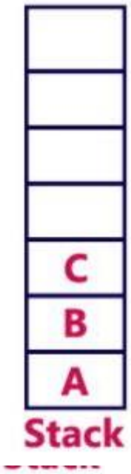




Step 3:

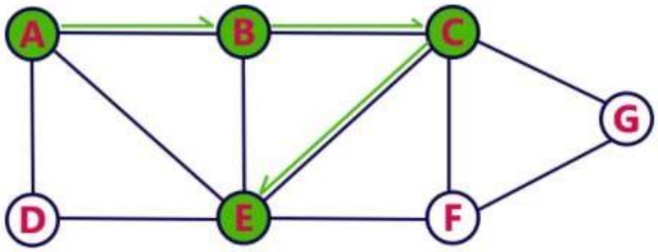
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

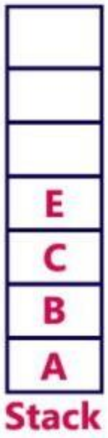




Step 4:

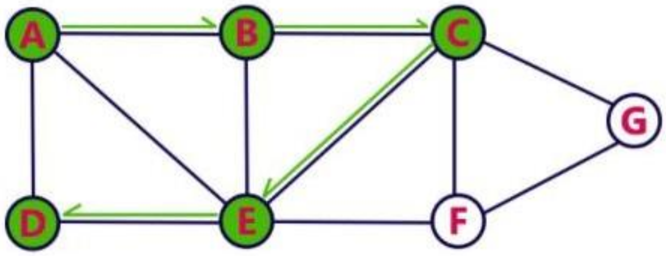
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

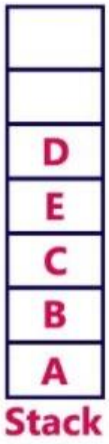




Step 5:

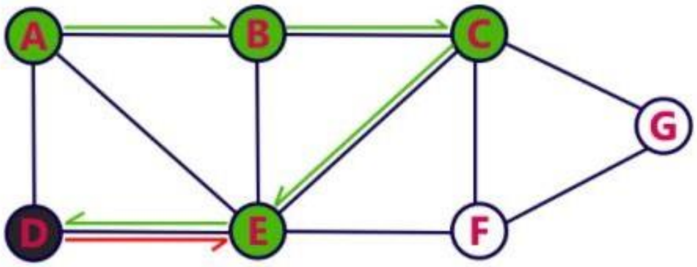
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

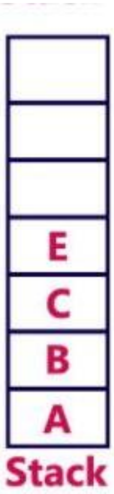




Step 6:

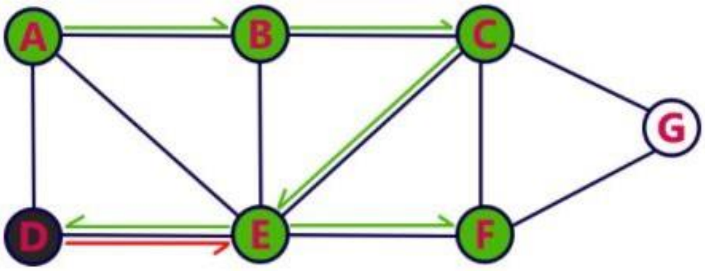
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

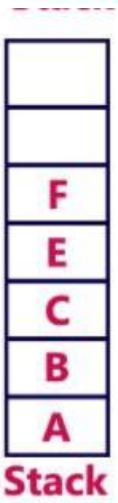




Step 7:

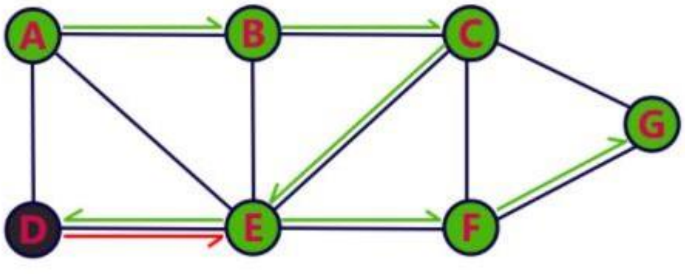
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

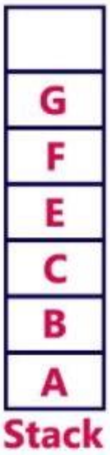




Step 8:

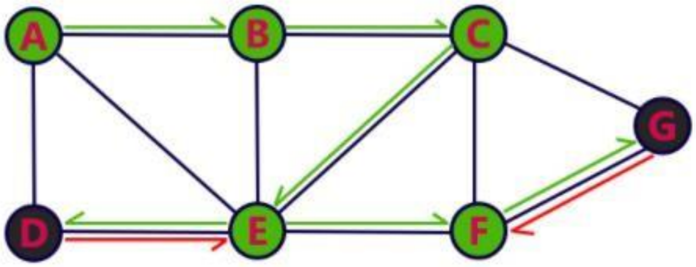
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

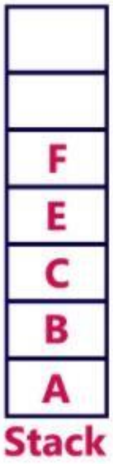




Step 9:

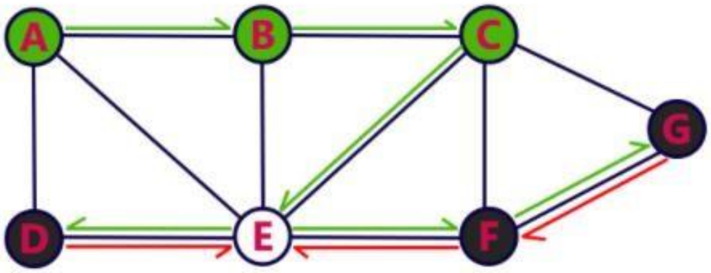
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

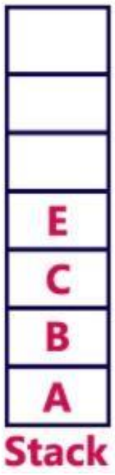




Step 10:

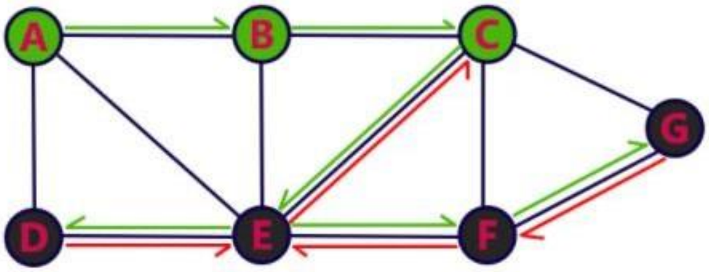
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

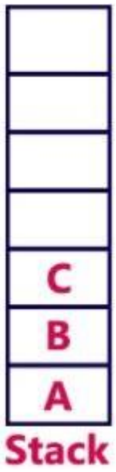




Step 11:

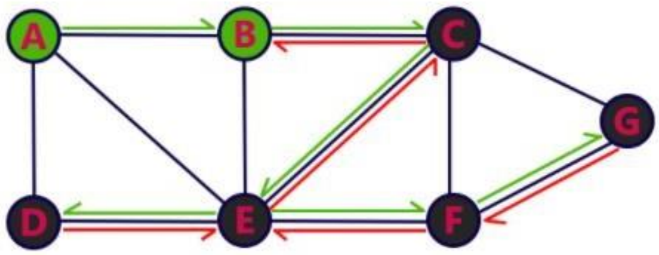
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

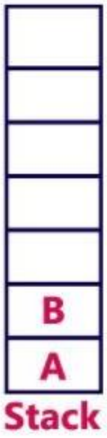




Step 12:

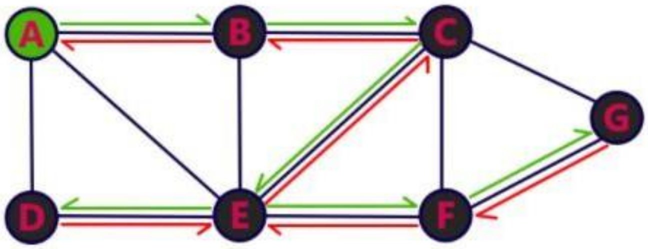
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

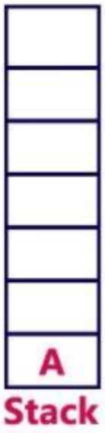




Step 13:

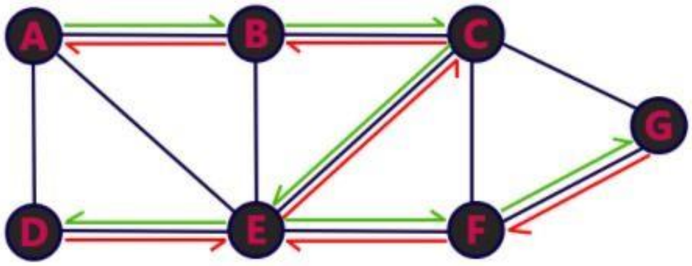
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

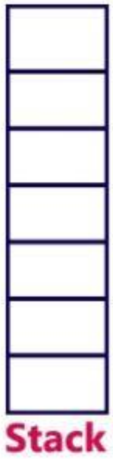




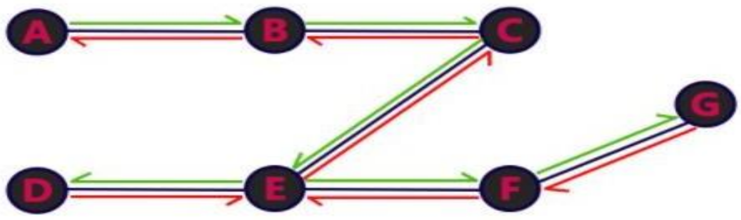
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





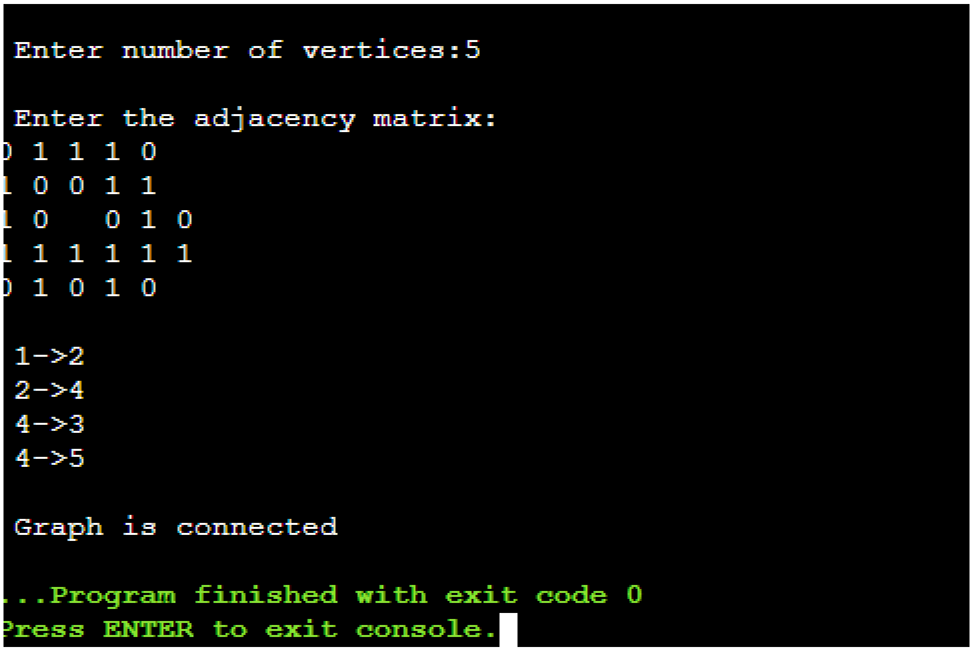
- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Program

```
#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;
            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");
}
```

OUTPUT:



```
 Enter number of vertices:5

 Enter the adjacency matrix:
0 1 1 1 0
1 0 0 1 1
1 0   0 1 0
1 1 1 1 1 1
0 1 0 1 0

 1->2
 2->4
 4->3
 4->5

 Graph is connected

...Program finished with exit code 0
Press ENTER to exit console.
```

BFS (Breadth First Search)

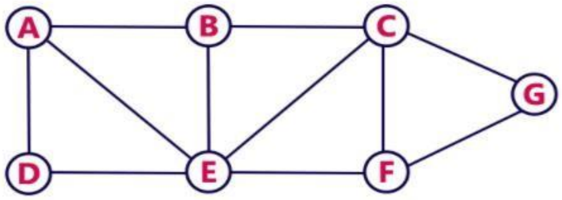
BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

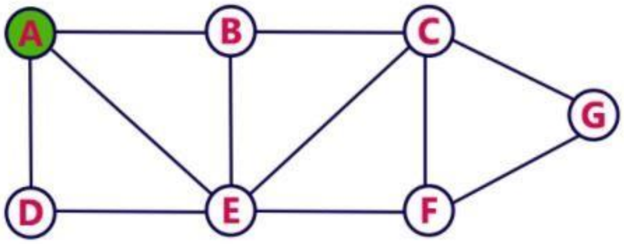
EXAMPLE

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

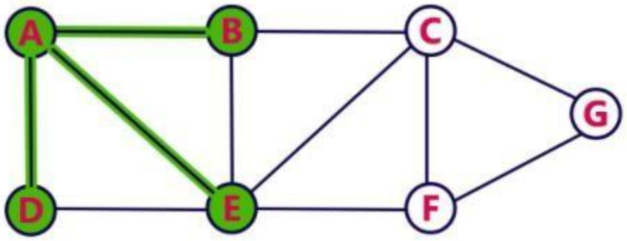


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

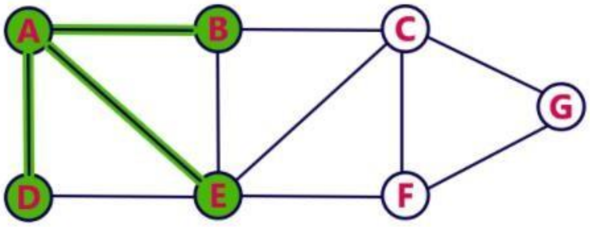


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

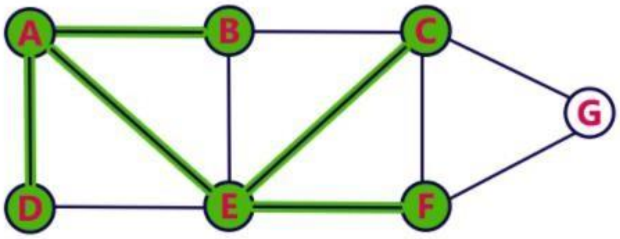


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

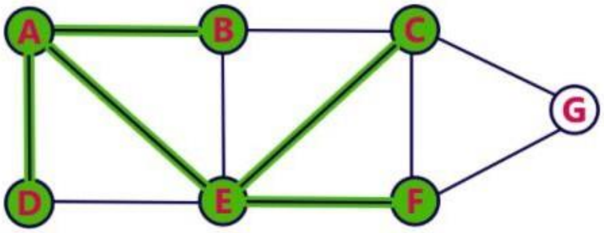


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

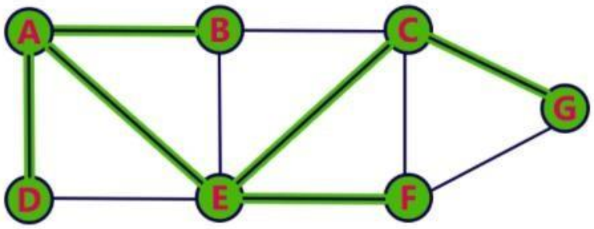


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

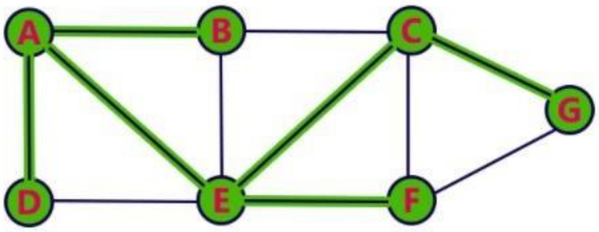


Queue

					F	G

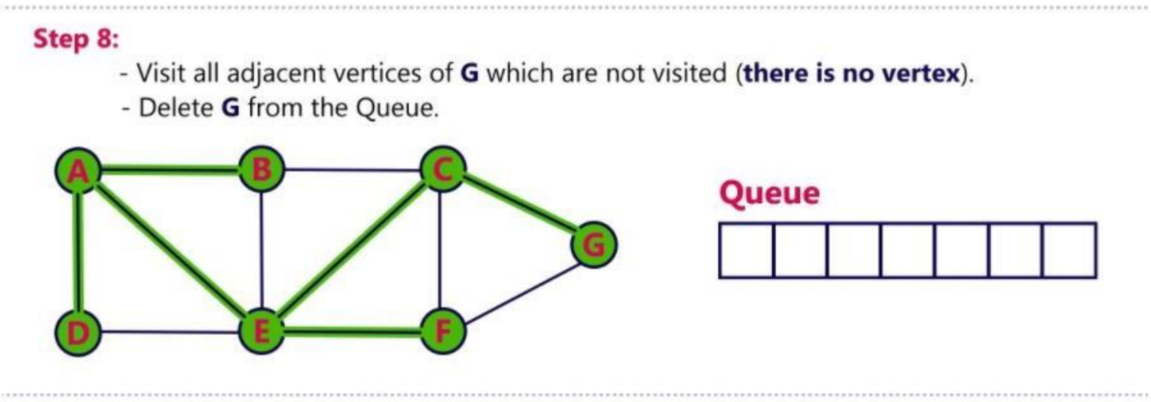
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



Queue

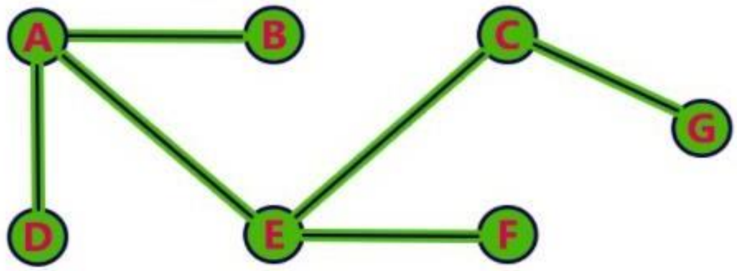
						G



Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.

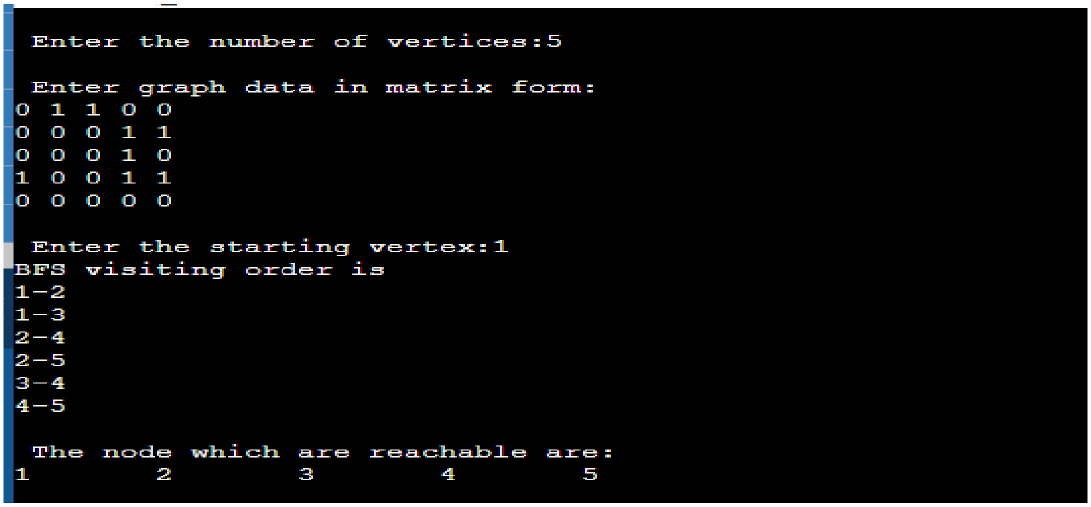
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



PROGRAM :

```
#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)
{
            visited[v]=1;
            for (i=1;i<=n;i++)
            {
              if(a[v][i] && !visited[i])
              {
               printf("%d-%d\n",v,i);
               q[++r]=i;
              }
            }
            if(f<=r)
            {
                        visited[q[f]]=1;
                        bfs(q[f++]);
            }
}
void main()
{
            int v;
            printf("\n Enter the number of vertices:");
            scanf("%d",&n);
            for (i=1;i<=n;i++)
            {
                        q[i]=0;
                        visited[i]=0;
            }
           // GRAPH IS GIVEN AS ADJACENCY MATRIX
            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);
            printf("BFS visiting order is\n");
            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");
}
```

OUTPUT :



```
 Enter the number of vertices:5

 Enter graph data in matrix form:
0 1 1 0 0
0 0 0 1 1
0 0 0 1 0
1 0 0 1 1
0 0 0 0 0

 Enter the starting vertex:1
BFS visiting order is
1-2
1-3
2-4
2-5
3-4
4-5

 The node which are reachable are:
1       2       3       4       5
```

UNIT IV

Sorting: Heap Sort, External Sorting- Model for external sorting, Merge Sort

SORTING INTRODUCTION

Sorting is nothing but arranging the data in ascending or descending order.

The term **sorting** came into picture, as humans realized the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

Sorting Efficiency

The two main criteria to judge which algorithm is better than the other have been:

1. Time taken to sort the given data.
2. Memory Space required to do so.

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering here.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Merge Sort
5. Heap Sort

Sorting Terminology

What is in-place sorting?

An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list. For example, Insertion Sort and Selection Sorts are in-place sorting algorithms as they do not use any additional space for sorting the list and a typical implementation of Merge Sort is not in-place.

What are Internal and External Sorting?

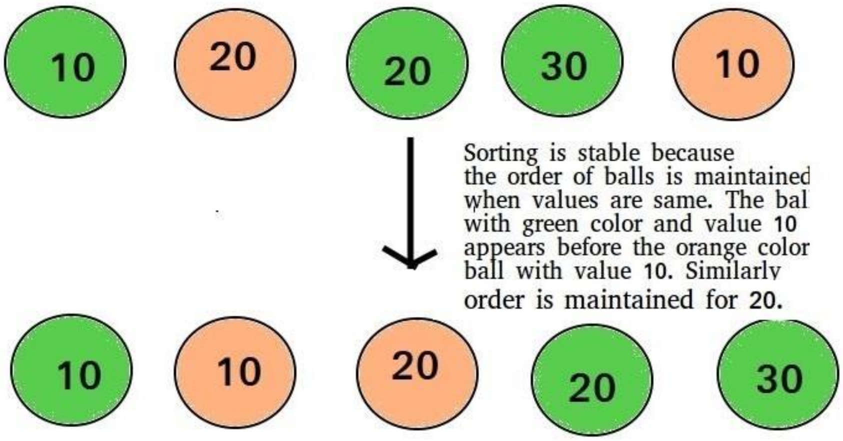
When all data that needs to be sorted cannot be placed in-memory at a time, the sorting is called external sorting. External Sorting is used for massive amount of data. Merge Sort and its variations are typically used for external sorting. Some external storage like hard-disk, CD, etc is used for external storage.
When all data is placed in-memory, then sorting is called internal sorting.

Example of external sorting is Merge Sort.

What is stable sorting?

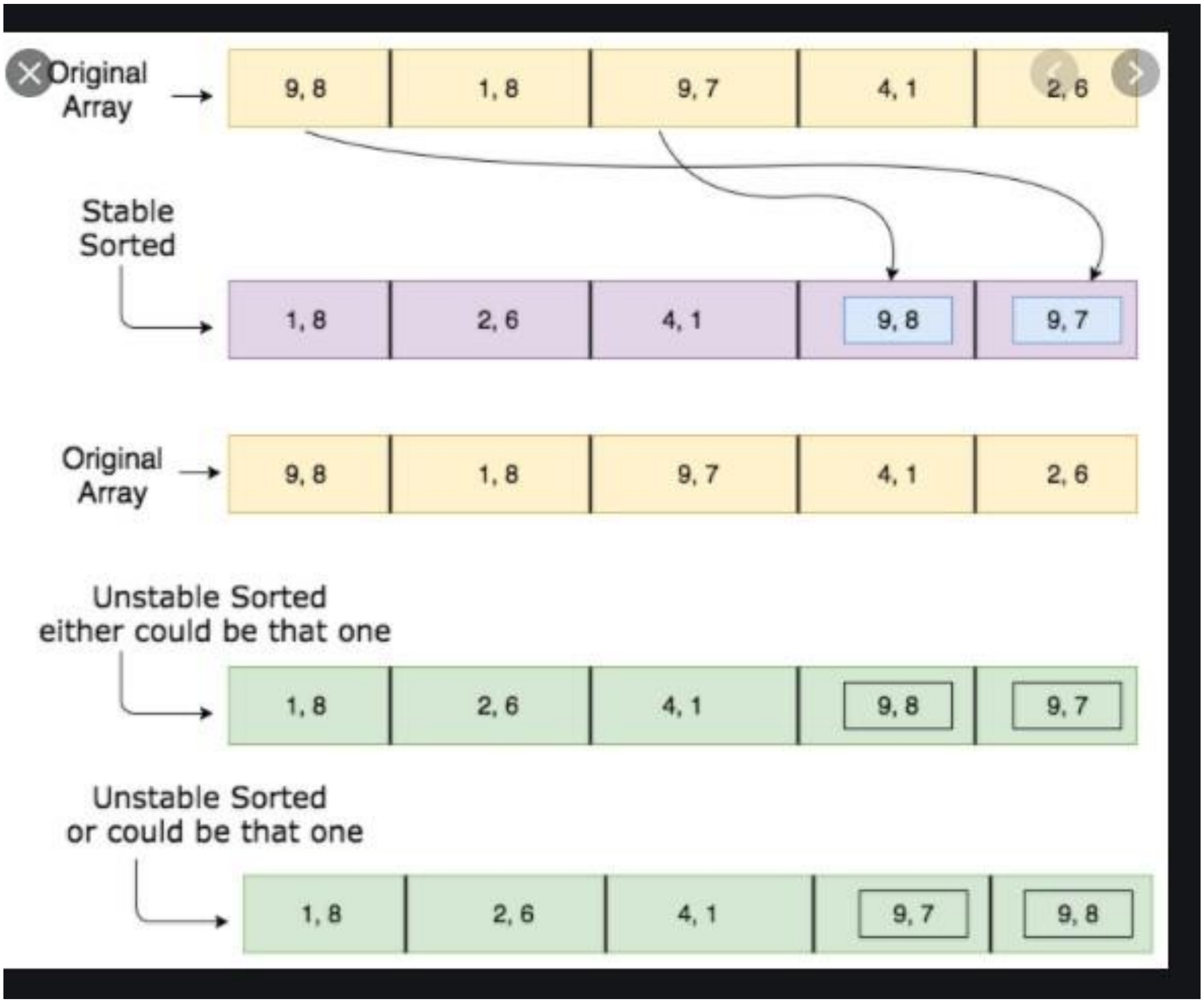
Stability is mainly important when we have key value pairs with duplicate keys possible (like people names as keys and their details as values). And we wish to sort these objects by keys.

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.
Informally, stability means that equivalent elements retain their relative positions, after sorting.



Sorting is stable because the order of balls is maintained when values are same. The bal with green color and value 10 appears before the orange color ball with value 10. Similarlv order is maintained for 20.

When equal elements are indistinguishable, such as with integers or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different.

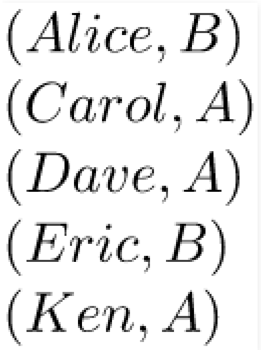


An example where it is useful

Consider the following dataset of Student Names and their respective class sections.

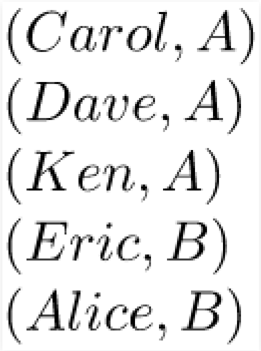
$$
\begin{array}{l}
(Dave, A) \\
(Alice, B) \\
(Ken, A) \\
(Eric, B) \\
(Carol, A)
\end{array}
$$

If we sort this data according to name only, then it is highly unlikely that the resulting dataset will be grouped according to sections as well.



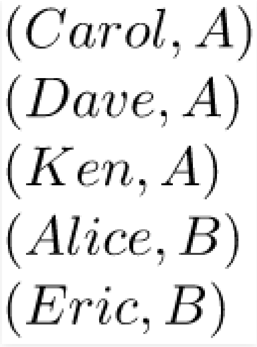
$$
\begin{array}{l}
(Alice, B) \\
(Carol, A) \\
(Dave, A) \\
(Eric, B) \\
(Ken, A)
\end{array}
$$

So we might have to sort again to obtain list of students section wise too. But in doing so, if the sorting algorithm is not stable, we might get a result like this-



$$
\begin{array}{l}
(Carol, A) \\
(Dave, A) \\
(Ken, A) \\
(Eric, B) \\
(Alice, B)
\end{array}
$$

The dataset is now sorted according to sections, but not according to names.
In the name-sorted dataset, the tuple (alice , B)was before (ERIC,B), but since the sorting algorithm is not stable, the relative order is lost.
If on the other hand we used a stable sorting algorithm, the result would be-



$$
\begin{array}{l}
(Carol, A) \\
(Dave, A) \\
(Ken, A) \\
(Alice, B) \\
(Eric, B)
\end{array}
$$

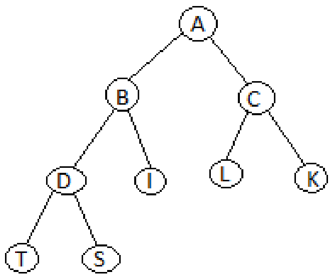
HEAP SORT

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a **Heap** data structure from the given array and then utilizing the Heap to sort the array.

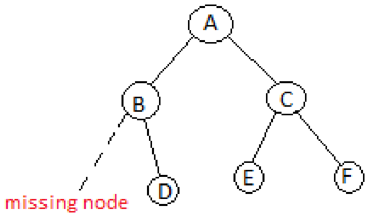
What is a Heap?

Heap is a special tree-based data structure that satisfies the following special heap properties:

1. Shape Property: Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

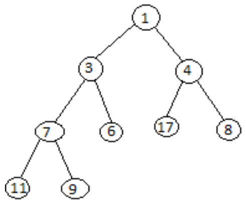


Complete Binary Tree



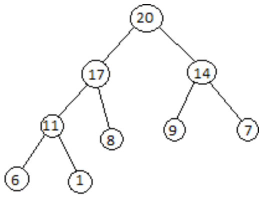
In-Complete Binary Tree

Heap Property: All nodes are either **greater than or equal to** or **less than or equal to** each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Building Heap from Array

Algorithm

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

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

Step 4 − If value of parent is less than child, then swap them.

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

Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Example:

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}

Corresponding Complete Binary Tree is:

```
           1

         /    \

        3        5

      /   \    / \

     4      6  13 10

    / \   / \

   9  8 15 17
```

The task to build a Max-Heap from above array.

Total Nodes = 11.

Last Non-leaf node index = (11/2) - 1 = 4.

Therefore, last non-leaf node = 6.

To build the heap, heapify only the nodes:

[1, 3, 5, 4, 6] in reverse order.

Heapify 6: Swap 6 and 17.

```
          1

        /    \

      3        5

    /   \     / \

   4      17  13 10

  / \   / \

 9  8 15   6
```

Heapify 4: Swap 4 and 9.

```
          1

        /    \

      3        5

    /   \     / \

   9      17  13 10

  / \   / \

 4  8 15   6
```

Heapify 5: Swap 13 and 5.

```
          1
```

```
      /    \
     3        13
   /   \     /  \
  9      17  5   10
 / \   / \
4  8 15   6
```

Heapify 3: First Swap 3 and 17, again swap 3 and 15.

```
          1
      /    \
     17        13
   /   \     / \
  9      15   5  10
 / \   / \
4  8 3  6
```

Heapify 1: First Swap 1 and 17, again swap 1 and 15,

Finally swap 1 and 6.

```
         17
      /     \
    15       13
   /   \    / \
  9     6   5  10
 / \   / \
4  8 3    1
```

Heap Sort Algorithm

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heap sort algorithm uses one of the tree concepts called **Heap Tree**. In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in ascending order.

Step by Step Process

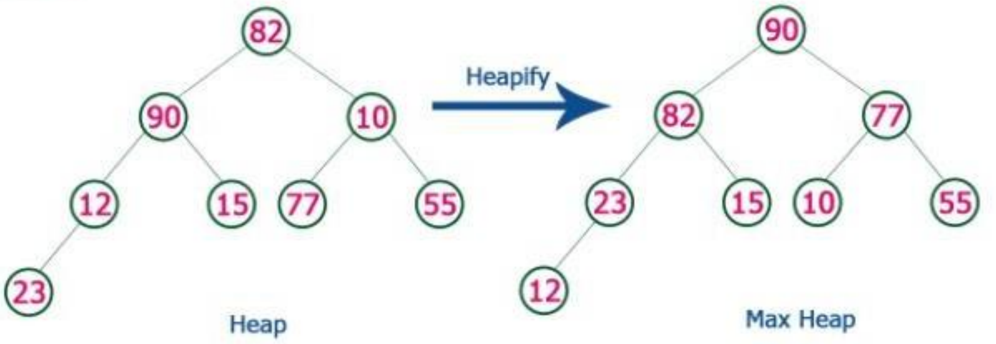
The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Min Heap (descending order/max heap (Ascending order).**
- **Step 3 -** Delete the root element from Min Heap/max heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Min Heap becomes empty.
- **Step 6 -** Display the sorted list.

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap

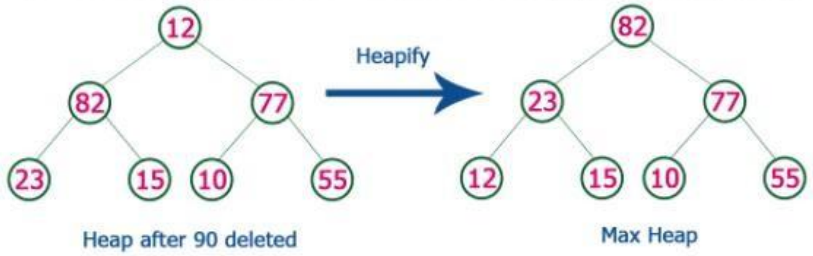


list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

90, 82, 77, 23, 15, 10, 55, 12

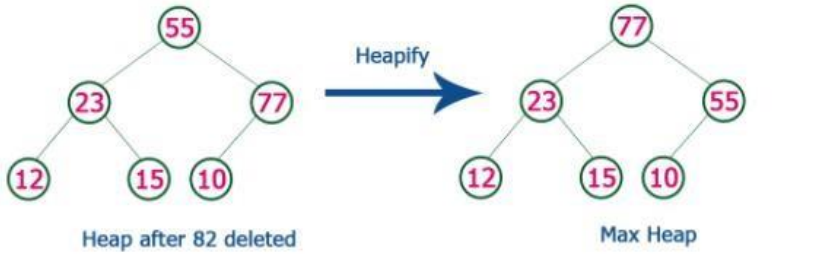
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

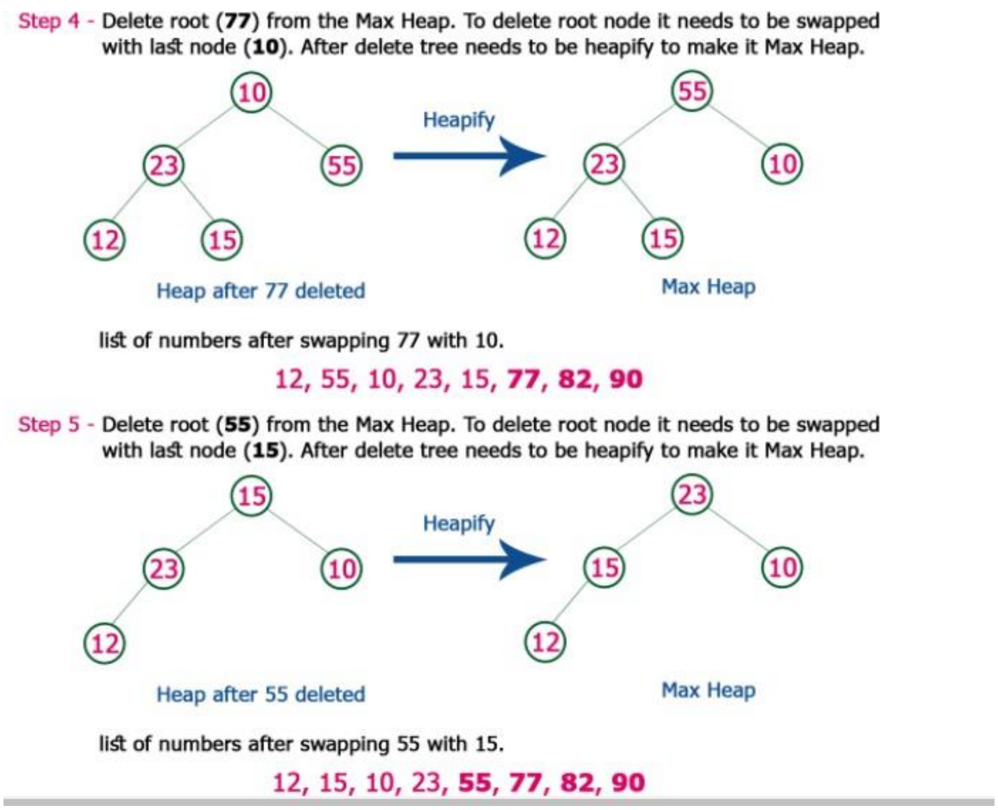
12, 82, 77, 23, 15, 10, 55, **90**

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82**, **90**



Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

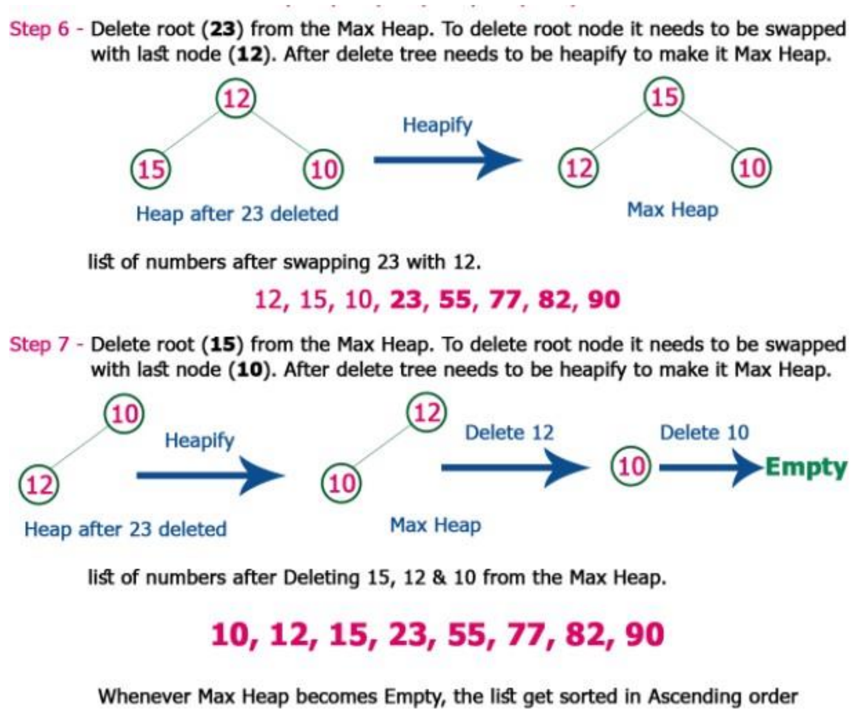
list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77**, **82**, **90**

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 55 with 15.

12, 15, 10, 23, **55**, **77**, **82**, **90**



Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, **23**, **55**, **77**, **82**, **90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Note:

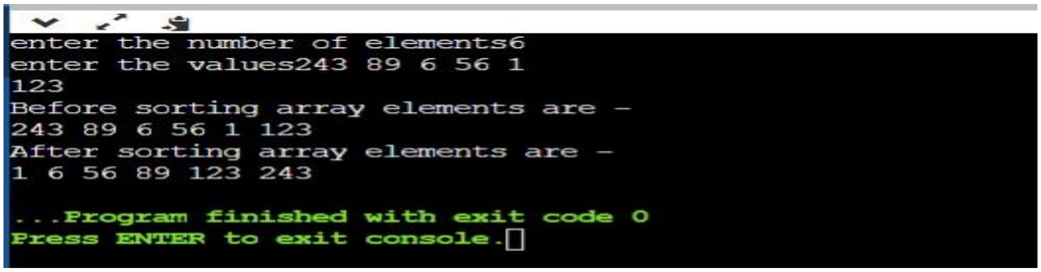
Heap sort is an in-place algorithm.
Its typical implementation is not stable, but can be made stable.

PROGRAM

```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
  int largest = i; // Initialize largest as root
  int left = 2 * i + 1; // left child
  int right = 2 * i + 2; // right child
  // If left child is larger than root
  if (left < n && a[left] > a[largest])
    largest = left;
  // If right child is larger than root
  if (right < n && a[right] > a[largest])
    largest = right;
  // If root is not largest
  if (largest != i) {
    // swap a[i] with a[largest]
    int temp = a[i];
    a[i] = a[largest];
    a[largest] = temp;
    heapify(a, n, largest);
  }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
  for (int i = n / 2 - 1; i >= 0; i--)
    heapify(a, n, i);
  // One by one extract an element from heap
  for (int i = n - 1; i >= 0; i--) {
    /* Move current root element to end*/
    // swap a[0] with a[i]
    int temp = a[0];
```

```
        a[0] = a[i];
        a[i] = temp;
        heapify(a, i, 0);
    }
}
/* function to print the array elements */
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        printf("%d", arr[i]);
        printf(" ");
    }
}
int main()
{
    int a[100],n ;
    printf("enter the number of elements");
    scanf("%d",&n);
    printf("enter the values");
    for(int i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    heapSort(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

Output:



```
enter the number of elements6
enter the values243 89 6 56 1
123
Before sorting array elements are -
243 89 6 56 1 123
After sorting array elements are -
1 6 56 89 123 243

...Program finished with exit code 0
Press ENTER to exit console.
```

Time Complexity:

Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

MERGE SORT

Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.

Before jumping on to, how merge sort works and its implementation, first let's understand what the rule of Divide and Conquer is?

Divide and Conquer

If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

Let's take an example, **Divide and Rule**.

When Britishers s came to India, they saw a country with different religions living in harmony, hard working but naive citizens, unity in diversity, and found it difficult to establish their empire. So, they adopted the policy of **Divide and Rule**. Where the population of India was collectively a one big problem for them, they divided the problem into smaller problems, by instigating rivalries between local kings, making them stand against each other, and this worked very well for them.

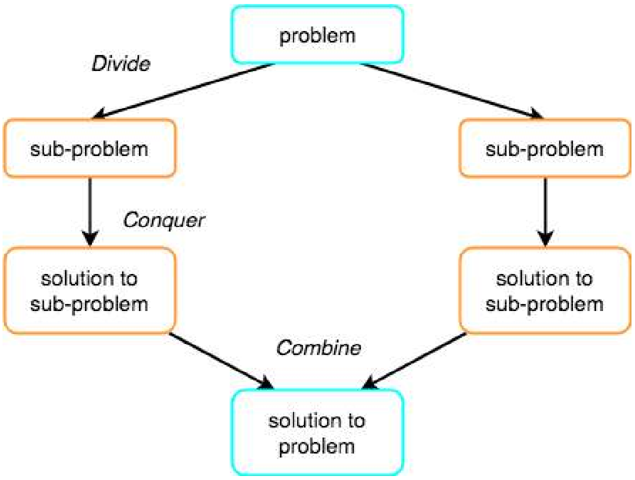
Well that was history, and a socio-political policy (**Divide and Rule**), but the idea here is, if we can somehow divide a problem into smaller sub-problems, it becomes easier to eventually solve the whole problem.

In **Merge Sort**, the given unsorted array with n elements is divided into n sub arrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these sub arrays, to produce new sorted sub arrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.

2. **Conquer** the sub problems by solving them. The idea is to break down the problem into atomic sub problems, where they are actually solved.
3. **Combine** the solutions of the sub problems to find the solution of the actual problem.



How Merge Sort Works?

As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, **sorting a given array**.

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two sub arrays with 3 elements each.

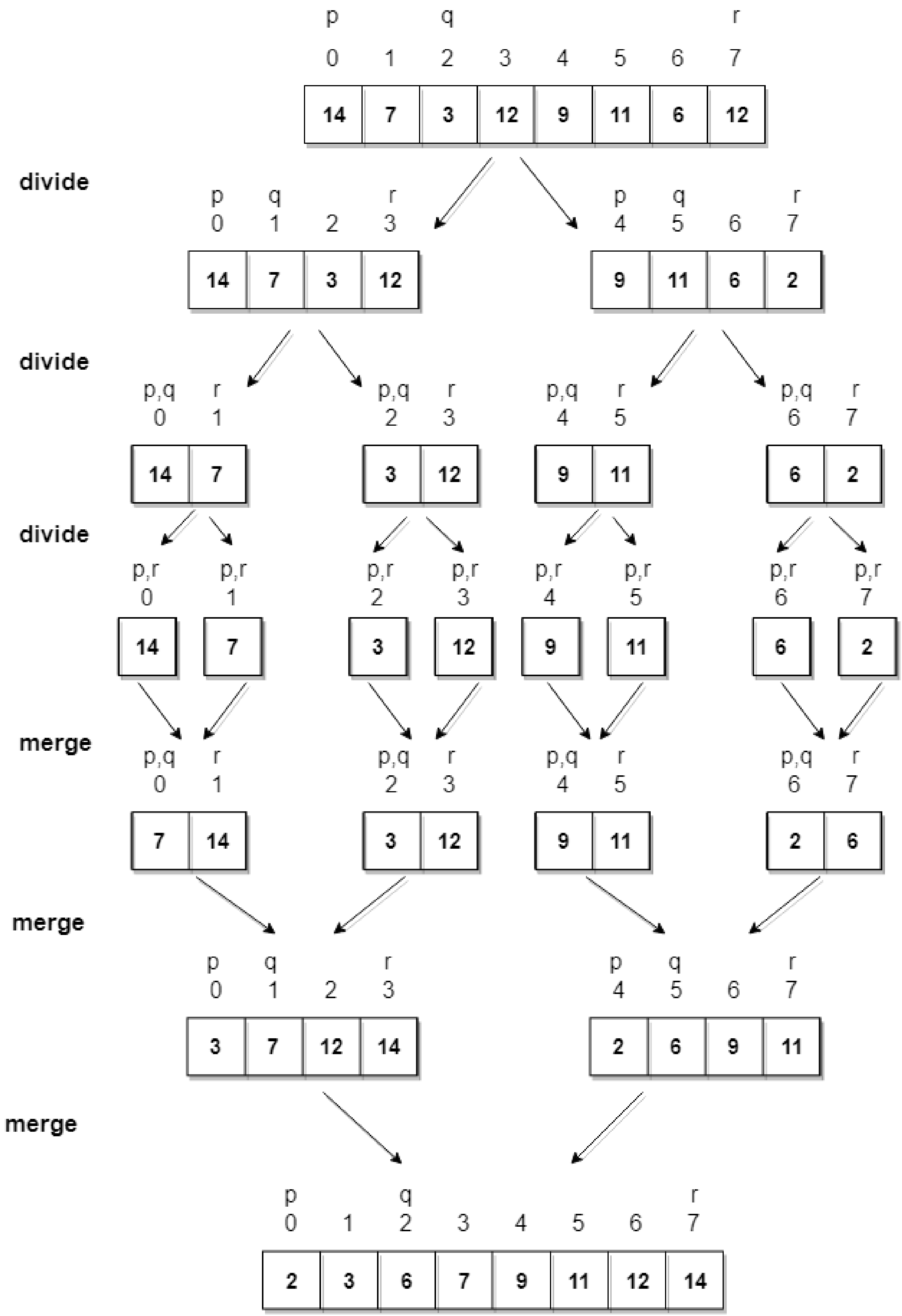
But breaking the original array into 2 smaller sub arrays is not helping us in sorting the array.

So we will break these sub arrays into even smaller sub arrays, until we have multiple sub arrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into sub arrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted sub arrays, step by step to form one single sorted array.

Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}

Below, we have a pictorial representation of how merge sort will sort the given array.



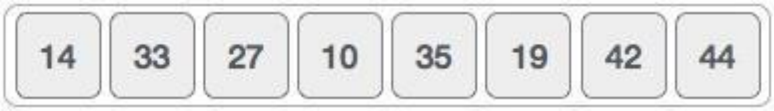
p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 12
divide
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
divide
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
divide
p,r p,r
0 1
14 7
p,r p,r
2 3
3 12
p,r p,r
4 5
9 11
p,r p,r
6 7
6 2
merge
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
merge
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
merge
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

In merge sort we follow the following steps:

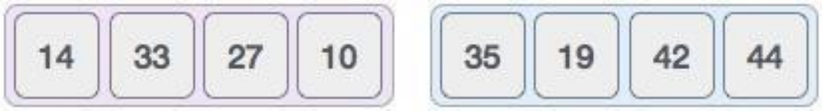
1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
2. Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two sub arrays, from p to q and from q + 1 to r index.
3. Then we divide these 2 sub arrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into sub arrays with single elements, then we start merging the sub arrays.

Example

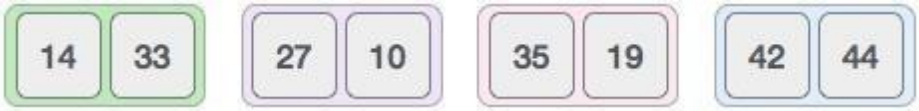
To understand merge sort, we take an unsorted array as the following −



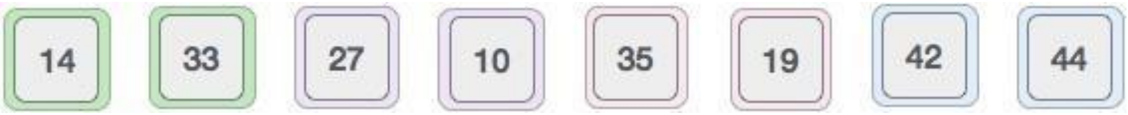
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

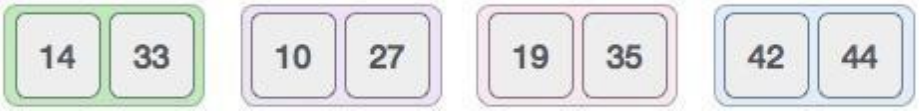


We further divide these arrays and we achieve atomic value which can no more be divided.

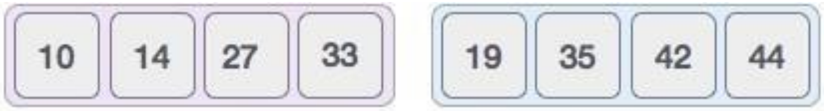


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

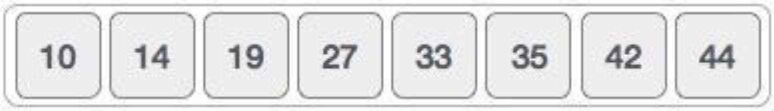
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this –



PROGRAM

```
#include <stdio.h>
void mergeSort(int [], int, int, int);
void partition(int [],int, int);
int main()
{
   int list[50];
   int i, size;
   printf("Enter total number of elements:");
   scanf("%d", &size);
   printf("Enter the elements:\n");
   for(i = 0; i < size; i++)
   {
       scanf("%d", &list[i]);
   }
   partition(list, 0, size - 1);
```

```
    printf("After merge sort:\n");
    for(i = 0;i < size; i++)
    {
         printf("%d  ",list[i]);
    }
   return 0;
}
void partition(int list[],int low,int high)
{
    int mid;
    if(low < high)
    {
        mid = (low + high) / 2;
        partition(list, low, mid);
        partition(list, mid + 1, high);
        mergeSort(list, low, mid, high);
    }
}
void mergeSort(int list[],int low,int mid,int high)
{
    int i, mi, k, lo, temp[50];
    lo = low;
    i = low;
    mi = mid + 1;
    while ((lo <= mid) && (mi <= high))
    {
        if (list[lo] <= list[mi])
        {
            temp[i] = list[lo];
            lo++;
        }
        else
        {
            temp[i] = list[mi];
            mi++;
        }
        i++;
    }
    if (lo > mid)
    {
        for (k = mi; k <= high; k++)
        {
            temp[i] = list[k];
            i++;
        }
    }
    else
    {
        for (k = lo; k <= mid; k++)
        {
             temp[i] = list[k];
             i++;
        }
    }

    for (k = low; k <= high; k++)
    {
        list[k] = temp[k];
    }
}
```

TIME COMPLEXITY

The time complexity of Merge Sort is O(n*Log n) in all the 3 cases (worst, average and best) as the merge sort always divides the array into two halves and takes linear time to merge two halves

COMPARISON OF SORTING TECHNIQUES

Insertion Sort

Properties:

- **INSERTION-SORT** can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are.
- In INSERTION-SORT, the **best case** occurs if the array is already sorted**.**

T [Best Case]= O(n)

- If the array is in reverse sorted order i.e. in decreasing order, INSERTION-SORT gives the **worst case** results.

T [Worst Case]= *o*(*n*²)

- **Average Case**: When half the elements are sorted while half not
- The running time of insertion sort therefore belongs to both Ω(n) and O(n²)

Pros:

- For nearly-sorted data, it's incredibly efficient (very near O(n) complexity)
- It works in-place, which means no auxiliary storage is necessary i.e. requires only a constant amount O(1) of additional memory space
- Efficient for (quite) small data sets.
- Stable, i.e. does not change the relative order of elements with equal keys

Cons:

- It is less efficient on list containing more number of elements
- Insertion sort needs a large number of element shifts

Merge Sort:

Properties

- Merge Sort's running time is **0(nlogn) in best, worst and average case**
- The **space complexity** of Merge sort is **O(n)**. This means that this algorithm takes a lot of space and May slower down operations for the last data sets.
- Merge sort is external sorting.

Pros:

- It is **quicker for larger lists** because unlike insertion it doesn't go through the whole list several times.
- The merge sort is **slightly faster than the heap sort** for larger sets
- $(nlogn)$ worst case asymptotic complexity.
- Stable sorting algorithm
- Not a in-place sorting technique

Cons

- **Slower** comparative to the other sort algorithms **for smaller data sets**
- Marginally slower than quick sort in practice
- Goes through the whole process even if the list is sorted
- It uses more memory space to store the sub elements of the initial split list.
- It requires twice the memory of the heap sort because of the second array.

Insertion sort vs. Merge Sort

Similarity

- Both are comparison based sorting algorithms

Difference:

- To work on an almost sorted array, Insertion sort takes linear time i.e. O(n) while Merge takes O(n*logn) complexity to sort

Heap Sort

Properties:

- Heap sort involves **building a Heap data structure** from the given array and then **utilizing the Heap to sort the array**

- Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled

- A.heap_size of an array is initially the size of the array. At first iteration, after exchanging root of the max_heap tree (A[1]) with A[i] = A[A.length] (last element inside array A)

- Doing extract_max(), A.heap_size value will be decreased by 1

- max_heap structure should be max_heapified: A[Parent(i)] >= A[i], where Parent(i) returns i/2 of heap tree.

- Initially create a Heap. extract_max(), put element of the heap in the array until we have the complete sorted list in our array.

- Time complexity of heap sort is o(nlogn) in all the cases

- The Heap Sort sorting algorithm seems to have a worst case complexity of O(n log(n))
- Heap sort is in place sorting techniques.

Pros:

- Heap sort and merge sort are asymptotically optimal comparison sorts

Cons: N/A

Heap Sort vs. Merge Sort:

- The time required to merge in a merge sort is counterbalanced by the time required to build the heap in heap sort
- **Heap Sort is better :**

  The Heap Sort sorting algorithm uses O(1) space for the sorting operation while Merge Sort which takes O(n) space

- **Merge Sort is better**
  * The merge sort is slightly faster than the heap sort for larger sets
  * Heap sort is not stable because operations on the heap can change the relative order of equal items.

Heap Sort vs. Insertion Sort:

Similarity

- Heap sort and insertion sort are both used comparison based sorting technique

Differences

- Heap Sort is not stable whereas Insertion Sort is.
- When already sorted, Insertion Sort will not sort every element again where as Heap Sort will use extract max and heapify again and again When already sorted, Insertion Sort takes O(n) TC whereas Heap Sort will take O(n log(n)) time Insertion Sort is not efficient for large input data whereas Heap Sort is.