mood-book

UNIT - IV

Graph :

A graph is a non-linear datastructure consisting of collection of nodes and edges. The nodes are referred to as vertices and edges are lines that connects pair of vertices.

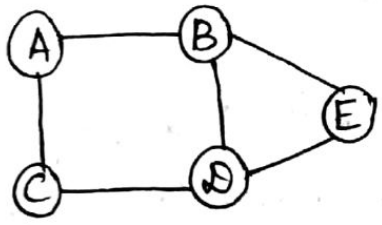
Generally, a graph G is represented as

$G=(V, E)$.

V → Set of vertices

E → Set of Edges

Eg:



→ fig (i)

Here, $G=(V,E)$

$V=\{A, B, C, D, E\}$

$E=\{(A,B), (A,C), (B,E), (C,D), (D,E), (B,D)\}$

A graph may have cycles.

Graph terminology:

Vertex:

An individual data element of a graph is called as Vertex or also known as node. Vertex is represented using a labelled circle. From fig(i), Vertices are A, B, C, D, E.

Edge:

A link that connects two Vertices is known as Edge. From fig(i), the edges are, {(A, B), (A, C), (B, D), (C, D), (B, E), (D, E)}.

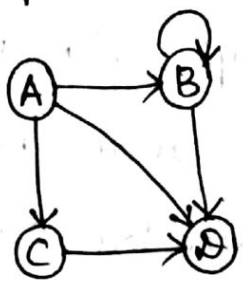
There are three types of edges,

1. Directed edge
2. Undirected edge
3. Weighted edge.

1. Directed edge:

A directed edge is a uni directional edge. It is represented using a '→' pointing from source to distination.

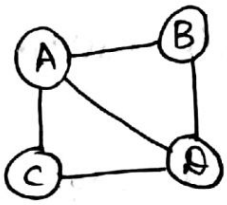
For example,



Here, A → B indicates A is source and B is the destination. Also, (A, B) is not equal to (B, A).

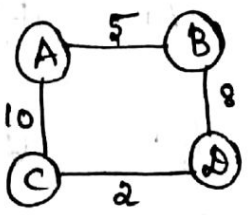
2. Undirected graph edge:

Undirected edge is a bidirectional edge, represented using '___'. Here, (A, B) = (B, A).



3. Weighted edge:

An edge with cost on it is known as weighted edge. It can be both directed and undirected. Eg:



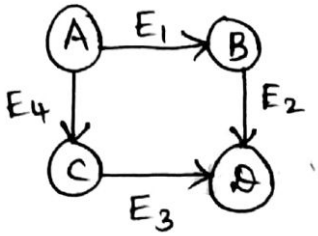
Directed graph – A graph with all directed edges.

Undirected graph- A graph with all undirected edges.

Mixed graph- A graph with both directed and undirected edges.

End points or end vertices- Two vertices joined by an edge are called end vertices of the edge.

Eg:

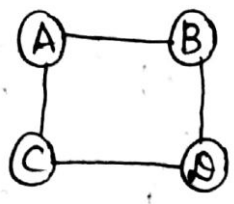


Here A and B are end points of E_1.

Origin – In a directed graph, the vertex from which the edge comes from is called as origin or source of edge.

Destination- In a directed edge, the vertex at which edge ends is called destination of the edge.

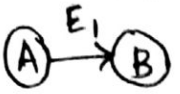
Adjacent- Two vertices are said to be adjacent to each other, if there is an edge that joins the two vertices. Eg:



Here, A and B are adjacent to each other.

Incident – An edge is incident on a vertex if the vertex is one of the end points of that edge.

Eg:



Here, A & B are incidents for E1.

Incoming edge – A directed edge on its destination vertex.

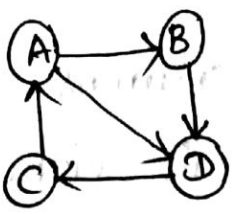
Outgoing edge – A directed edge on its source vertex.

Indegree – The total number of incoming edges of that vertex.

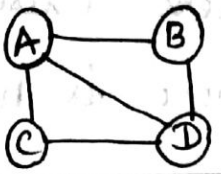
Outdegree – The total number of outgoing edges of that vertex.

Degree – The total number of edges incident on a vertex.

Eg:



Indegree of A : 1
Outdegree of A : 2



Degree of A : 3

Path:- A Sequence of edges between two Vertices.

Self-Loop:- A directed or undirected edge whose endpoints are same ie., an edge joining a Vertex to itself.

Graph Representation:

Graphs are generally represented in the following scheme as,

1. Adjacency matrix representation
2. Adjacency list representation
3. Incidence matrix representation.

1. Adjacency matrix representation:

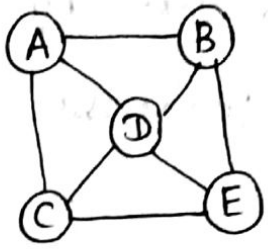
One simple way to represent a graph is to use 2-dimensional array., ie., using a matrix of size V×V, V- rows and V- columns. Here both rows and columns represents Vertices and the value of matrix is either 0 or 1.

1 - if there exists an edge b/w Vertices
0 - if there is no edge between Vertices.

Adjacency matrix is given for both directed and undirected graphs respectively.

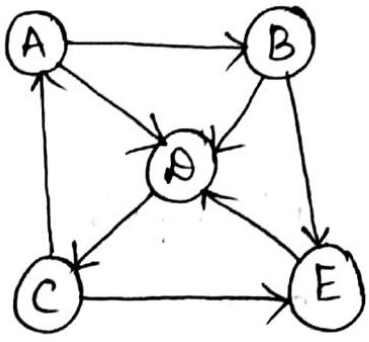
Eg:

Undirected graph



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	1
D	1	1	1	0	1
E	0	1	1	1	0

Directed graph



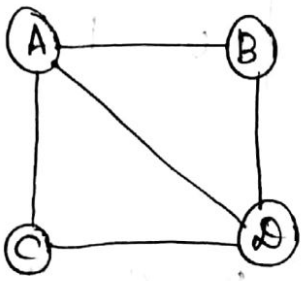
	A	B	C	D	E
A	0	1	0	1	0
B	0	0	0	1	1
C	1	0	0	0	1
D	0	0	1	0	0
E	0	0	0	1	0

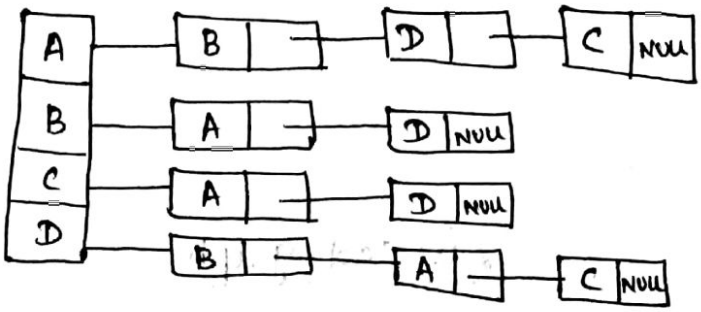
In directed graph, the cell value AB is 1 because there is an edge from A to B, but on the other side cell value of BA is 0.

2. Adjacency List representation:

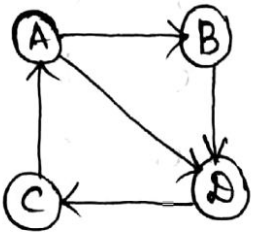
In this representation, graph is stored as a linked structure. If a node has any adjacency to any other node, then the adjacent node will be linked with its predecessor by storing its address in the link field of predecessor node.

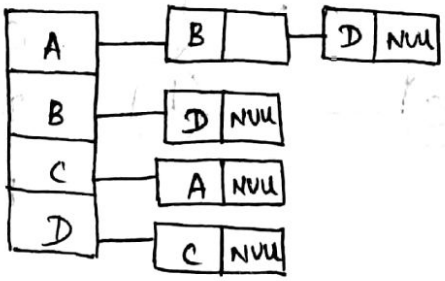
Undirected graph:





Directed graph:





3. Incidence Matrix representation:

In this representation, a graph G(V, E) with V vertices and E edges can be represented using a matrix of size V x E ie., V rows & E columns.

This matrix is filled with either 0, 1, or -1.

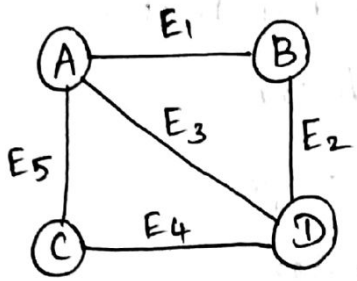
0 → There is no edge between Vertices for both directed & undirected graphs.

1 → There is an edge between Vertices for both directed & undirected graphs.

-1 → If there is an incoming edge from the row vertex to the column vertex, in directed graphs.

For self-loop edges, only 1 is used to represent both incoming and outgoing edge.

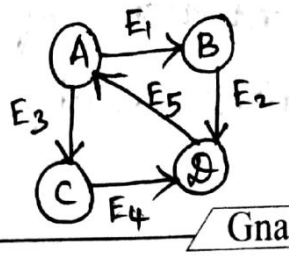
undirected graph:



	E_1	E_2	E_3	E_4	E_5
A	1	0	1	0	1
B	1	1	0	0	0
C	0	0	0	1	1
D	0	1	1	1	0

Directed graph:

	E_1	E_2	E_3	E_4	E_5
A	1	0	1	0	-1
B	-1	1	0	0	0
C	0	0	-1	1	0
D	0	-1	0	-1	1



Implementations of a graph:

① Implementing a directed graph and its operations using adjacency matrix.

Operations on the graph are,

* Print adjacency matrix
* Insert a vertex
* Insert an edge
* Delete a vertex
* Delete an edge.

② Implementing an undirected graph and its operations using adjacency matrix

Operations are,

* Print adjacency matrix
* Insert a Vertex
* Insert an edge
* Delete a Vertex
* Delete an edge

③ Implementation of directed and undirected graph and its operations using adjacent lists.

Graph traversal Implementation:

Graph traversal is a technique used for searching a vertex in a graph. The graph traversal finds the edges to be used in the search process without creating loops. The graph traversal decides the order of visiting of the vertices in search process.

There are two major graph traversal techniques such as,

* Depth First Search
* Breadth First Search.

Depth First Search:

It produces spanning tree as final result. Spanning tree is a graph without any loops. In dfs, starting at some vertex v, we process v then recursively traverse all vertices adjacent to v. To do this we mark a vertex v as visited once we visit it. We use stack data structure with maximum size of vertices in the graph to implement DFS traversal.

The Graph is represented using adjacency list method. Hence the node of the adjacency list is defined as,

```
Struct node
{
  Struct node *next;
  int Vertex;
};
  typedef Struct node *GNODE;
```

The graph array is,

```
GNODE graph[20];
```

The visited array is,

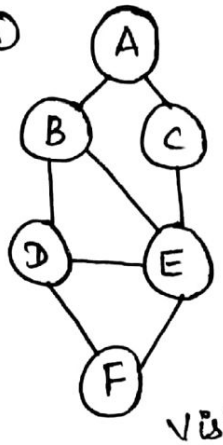
```
int Visited[20];
```

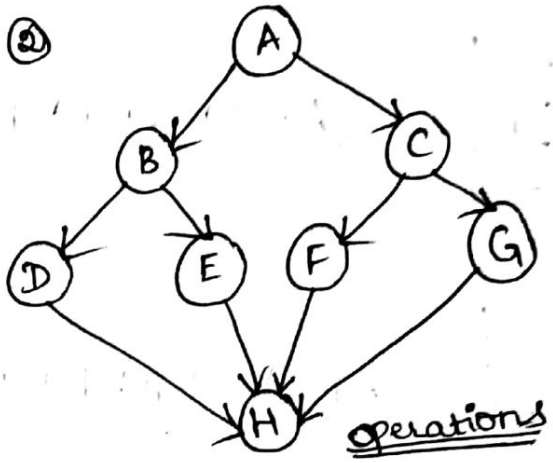
DFS(int i):

```
Void DFS(int i) {
      Declare a var p of type GNODE.
       Print %d
       Assign graph[i] to p.
      Set visited[i]=1
        while ( P!=NULL) {
          Assign Vertex of p to i
        if ( visited[i]!=1)
        { DFS(i); }
          Assign next of P to P
      } }
```

DFS is implemented using Stack. The time complexity is $(O(|V|+|E|))$.

Eg:

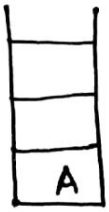
① 

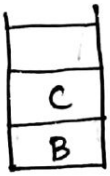
② 

By Solving ②

Visited

Stack operations

① 

② 

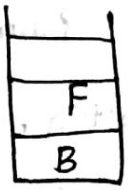
– (A)

③ 

– (A) (C)

④ 

– (A) (C) (G)

⑤ 

– (A) (C) (G) (H)

Initially, push A into stack. Start from the node A and traverse to the adjacent nodes of A. Then, mark node A as visited node, and pop it from the Stack.

Now push the adjacent nodes of A into stack. Continue the process until all nodes are visited once.

Stack

visited nodes

(6) 

- A C G H F

(7) 

- A C G H F B

(8) 

- A C G H F B E

(9) 
- A C G H F B E D

The Stack becomes empty and Stop the process.

Now, the order of traversal = A→C→G→H→F→B→E→D

Breadth First Search:

BFS traversal produces a spanning tree as a final result. We use Queue data structure with maximum size of number of vertices in the graph to implement BFS traversal. Here, a node is selected randomly as the start position. Starting from that node, all of its unvisited nodes are visited. This process is done recursively until all the nodes are visited.

The graph is represented using the adjacency list method.

```
Struct node {
    int vertex;
    Struct node * next;
};
Typedef Struct node *GNODE;
```

Graph array and visited node array is,

```
GNODE graph[20];
int visited[20];
```

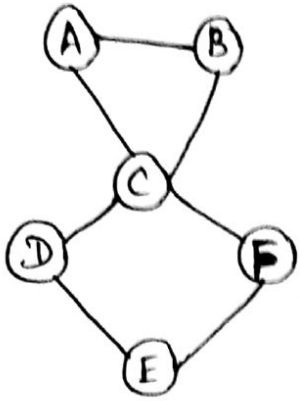
The queue array with front and rear variables are,

```
int queue[99], front=-1, rear=-1;
```

BFS(int V):

```
Void BFS(int V)
{ declare a var W.
Call: InsertQueue(V0)
   { v= dequeue(V);
     Print the Vertex value y
   Set visited[v]=1.
GNODE g=graph[v];
   for(; g!=NULL; g=g->next)
{
assign Vertex of g to W
      if(visited[w]==0) { call: InsertQueue(w); visited[w]=1;
      Print w; yyy print("\n"); }
```

BFS for the graph is Implemented as,



① B C - A

② C - A B

③ D F - A B C

④ F E - A B C D

⑤ E - A B C D F

⑥ - A B C D F E

Select node A as start node and make it as visited. Traverse to the adjacent nodes of A ie., B and C. Then, add them into a queue.

Delete the front item from queue and traverse its adjacents. Continue the process until it reaches the end ie., until all the nodes become visited.

The Queue becomes empty. So, Stop the Process. The order of traversal are,

A → B → C → D → F → E.

Heap Sort:

Heap Sort is widely used for its efficiency and well-defined data structure. The data structure used in heap sort is a binary tree heap. The binary heap data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The binary tree is completely filled at all level except at lowest level.

Heap tree:

A heap is a balanced binary tree that stores a collection of keys at its internal nodes and in which no node has a value greater than the value in its parent.

Heap tree satisfies two additional properties,

* Heap-order Property
* Structural property

Heap-order property :: In a heap H, for every node n, except the root, the key stored in n is smaller than or greater than or equal to the key stored in n's parent (Min heap and Max heap).

So, $A(i) \leq A(\text{parent}[i])$ → Min heap

$A(i) \geq A(\text{parent}[i])$ → Max heap.

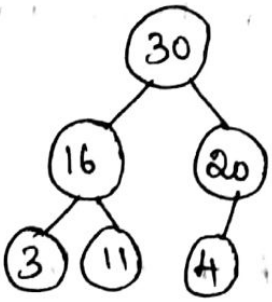
Structural property :

A Binary tree T is complete if each level but the last is full, and, in the last level, all of the internal nodes are to the left of the external nodes.

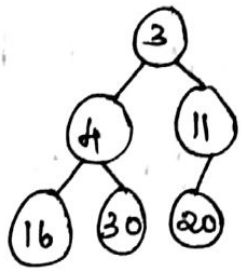
A binary tree of depth n is balanced if all the nodes at depths 0 through n-2 have two children. A balanced binary tree of depth n is left-justified if:

* It has 2^n nodes at depth n
* It has 2^k nodes at depth k, for all $k<n$, and all leaves at depth n are as far left as possible.

Representation of heap in binary tree and its corresponding array representation.



a) Max heap



b) Min heap.

Index	1	2	3	4	5	6
Value	30	16	20	3	11	4

a) Heap Array

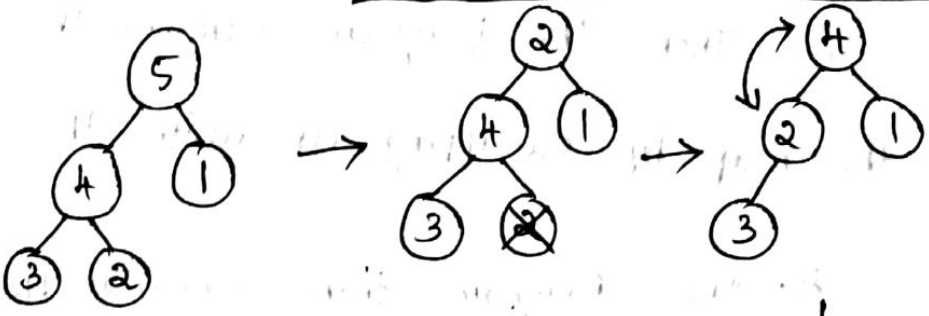
Index	1	2	3	4	5	6
value	3	4	11	16	30	20

b) Heap Array

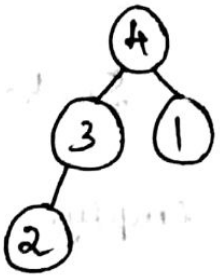
parent(i) = floor(i/2)
left(i) = 2i
Right(i) = 2i+1

Example for Heap Sort:

Max heap:



Now, 5 is removed from the tree.



The number of nodes possible in a heap of height 'h' is 2^h.

Procedure :-

1. User inputs the list of elements to be sorted. Then generates a binary tree with nodes having randomly generated key values.

2. Build heap operation – Let n be the number of nodes in the tree and i be the key of the tree. The program uses operation Heapify to let it settle down to a position by swapping itself with larger of its children; whenever the heap property is not satisfied till the heap property be satisfied in the tree which was rooted at (i).

3. Then, the program removes the largest element of the heap by swapping it with the last element.

4. The program then executes Heapify (new node) so that the resulting tree satisfies the heap property.

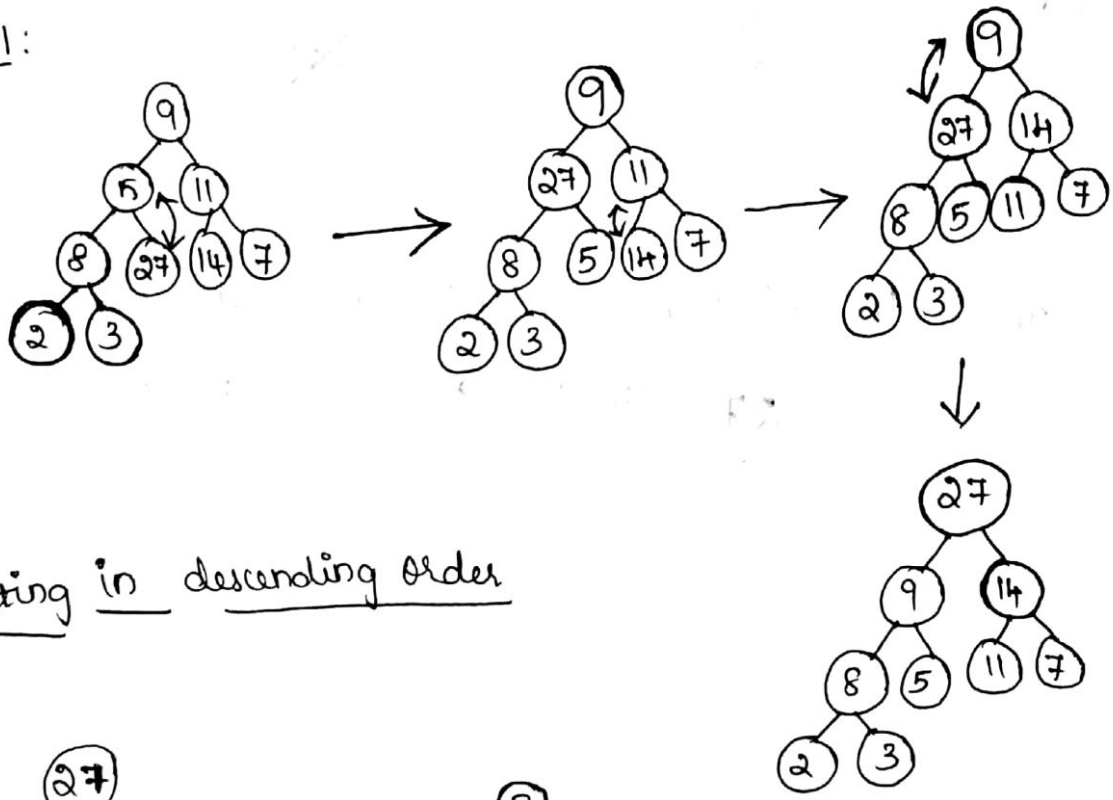
5. Repeat the process of step 3 till heap becomes empty.

Heap Sort

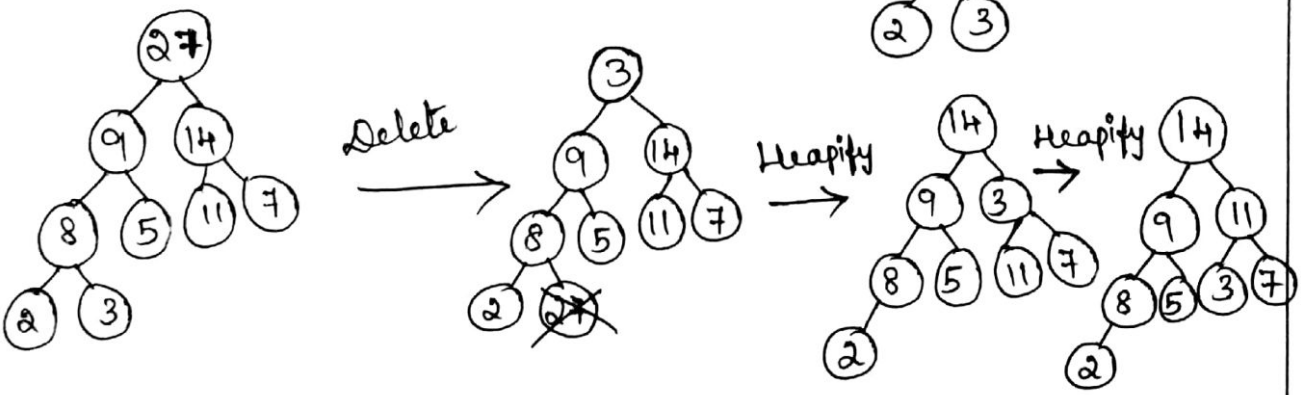
Example illustration : (Max heap)

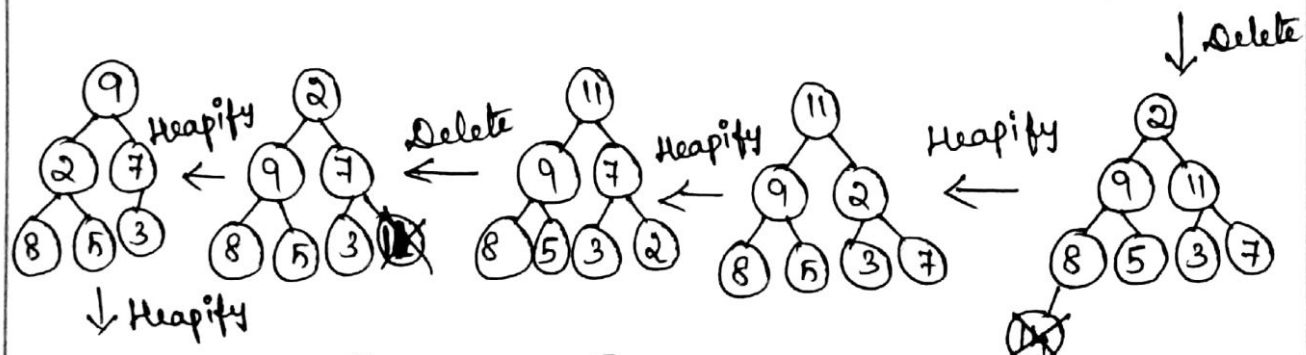
A = {9, 5, 11, 8, 27, 14, 7, 2, 3}

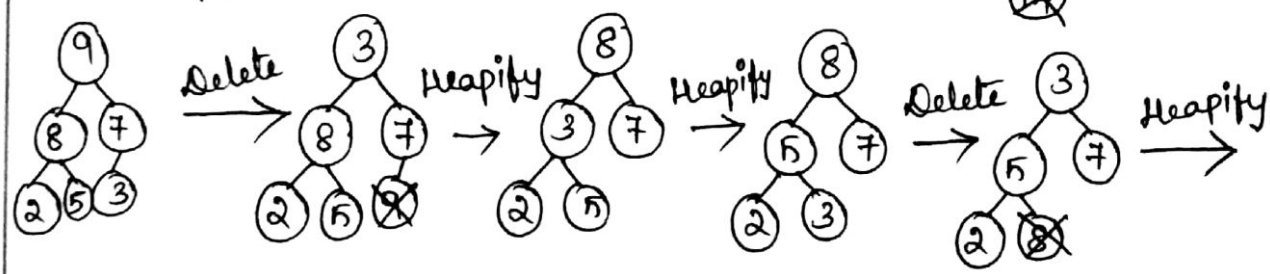
Step 1:

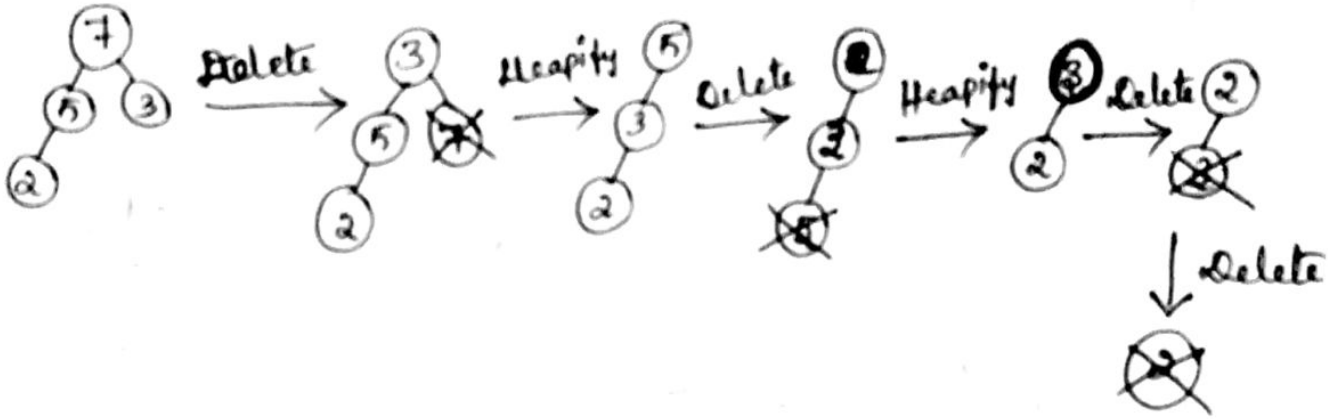


Sorting in descending order









Thus the Sorted elements are in the order,

27 14 11 9 8 7 5 3 2

Routine:-

```
Void heapify(Input_type a[], int i, int n)
{ int child; Input_type tmp;
 for(tmp=a[i]; i*2<=n; i=child)
 { child=i*2;
  if ((child!=n) && (a[child+1] > a[child]))
  child++;
   if(tmp<a[child])
  a[i]=a[child];
  else break; }
  a[i]=tmp; }

Void heapsort(input_type a[], int n)
{ int i,j; j=n;
for(i=n/2; i>0; i--)
heapify(a,i,j); i=1;
for(i=n; j>=2; j--)
{ swap(&a[i], &a[j]);
heapify(a,i,j-1); }}
```

Analysis

The time taken for building heap is $O(n)$

The time taken for finding and removing the smallest object is $O(\log n)$

Total time => $O(n + n\log n) = O(n\log n)$

External Sorting:

External Sorting is required when the data being Sorted do not fit into the main memory of a Computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive).

It is a class of Sorting algorithms that can handle massive amounts of data.

Merge Sort:

The merge Sort is an external Sorting technique which is based on a divide and conquer strategy. It produces a Sorted Sequence by Sorting its two halves and merging them, with a time Complexity $O(n \log(n))$. This technique can be described in three steps,

1. Divide Step: divide an array into two sub-arrays of equal size. ($A_1 + A_2$).
2. Recursion Step: recursively Sort arrays A1 & A2.
3. Conquer Step: Combine elements back into A by merging the Sorted arrays $A_1 + A_2$ into a Sorted Sequence.

Routine :

```
Void mergeSort (int lo, int hi)
{ if (lo<hi)
   int m=(lo+hi)/2;
   mergeSort (lo, m);
   mergeSort (m+1, hi);
   merge (lo, m, hi); }
Void merge (int lo, int m, int hi)
{ int i, j, k;
   for (i=lo; i<=hi; i++)
   b[i]=a[i]; i=lo; j=m+1; k=lo;
   while (i<=m && j<=hi)
   if (b[i]<=b[j])
      a[k++]=b[i++];
   else a[k++]=b[j++];
   while (i<=m)
      a[k++]=b[i++]; }
```

Bubble Sort :

Bubble Sort is an internal sorting technique in which adjacent elements are compared and exchanged if necessary.

In bubble Sort, the list at any moment is divided into two Sublists: Sorted and unsorted. The smallest element is bubbled from the unsorted Sublist and moved to the Sorted Sublist. Then, the wall moves ones element to the right, increasing number of Sorted elements and decreasing number of unsorted elements.

Given a list of 'n' elements, the bubble Sort requires upto n-1 passes to Sort the data.

The working Procedure is:-

1. Let an array of n elements (a[n]) to be Sorted.
2. Compare first two elements in the array ie., a[0] & a[1]
   If a[1] < a[0] then Swap the two values.
3. Next, Compare a[1] & a[2], if a[2] < a[1] then Swap.
4. Continue, until the last two elements are compared and interchanged.
5. Repeat the above Steps for n-1 passes.

The Average and worst-case complexity of bubble Sort is $O(n^2)$.

Let us consider an example of array numbers

50, 20, 40, 10, 80 and sort them from lowest number to highest number.

Pass-1:

50 ↔ 20 40 10 80

20 50 ↔ 40 10 80

20 40 50 ↔ 10 80

20 40 10 50 ↔ 80

20 40 10 50 [80]

Pass-2:

20 ↔ 40 10 50 [80]

20 40 ↔ 10 50 [80]

20 10 40 ↔ 50 [80]

20 10 40 [50] [80]

Pass-3:

20 ↔ 10 40 [50] [80]

10 20 ↔ 40 [50] [80]

10 20 [40] [50] [80]

pass-4:

10 ↔ 20 [40] [50] [80]

[10] [20] [40] [50] [80] ⇒ Sorted array //.

For 5 elements it takes 4 Passes. ie., for n elements it is (n-1) passes.

Insertion Sort :

Insertion Sort is one of the most common Sorting techniques used by card players. As they pick up each card, they insert it into the proper sequence in their hand. Likewise, insertion sort is one that Sorts a set of elements by inserting an element into existing sorted elements.

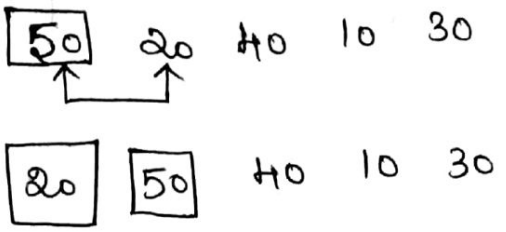
Working Procedure :

1. Consider an array of 'n' elements to be Sorted.
2. a[0] is itself trivially Sorted.
3. Now, a[1] is compared with a[0] and it will be inserted either before or after first element.
4. Now, a[2] is compared with a[0] and a[1] and it will be placed in a proper position by checking conditions, so the first 3 elements are sorted.
5. Repeat the same process for n-1 passes.

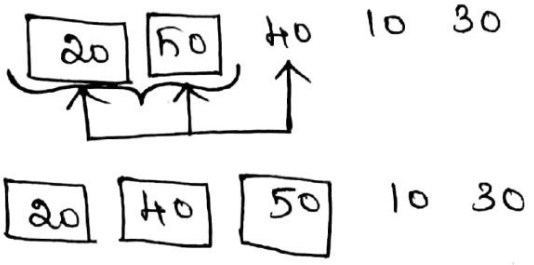
The average case complexity of insertion sort is, $O(n^2)$.

Let us consider an example of array numbers "50 20 40 10 30" to be Sorted.

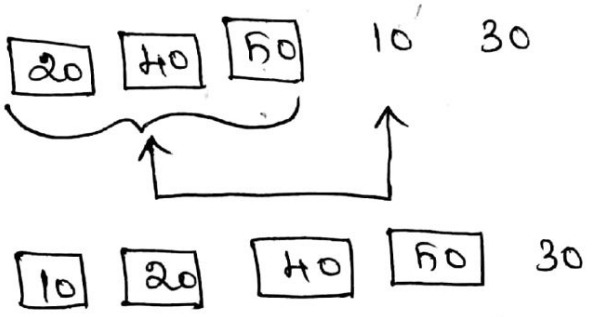
Pass-1:



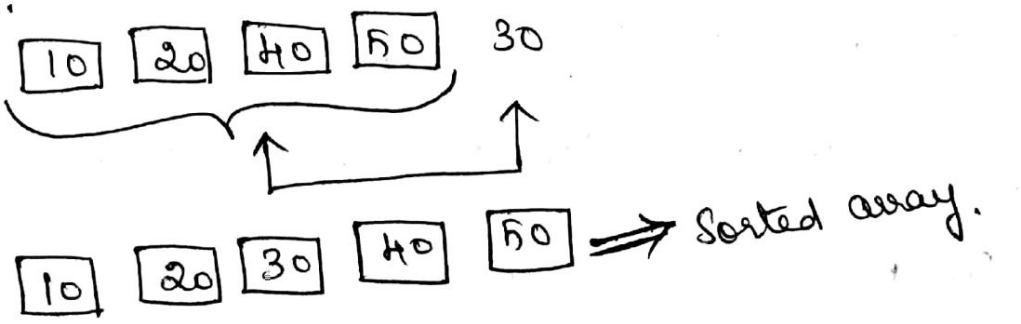
Pass-2:



Pass-3:



Pass-4:



The result is : 10 20 30 40 50 //.

Selection Sort:

The Selection Sort is a Sorting algorithm which has the time complexity of $O(n^2)$, making it inefficient on large lists, and generally performs worse than the Insertion Sort.

This can be done in two ways,

* Largest element method
* Smallest element method.

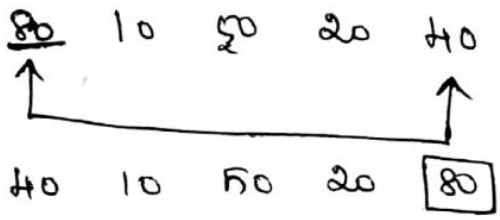
Largest element method:

Here the process starts with finding the largest element in the list. The working Procedure is as follows,

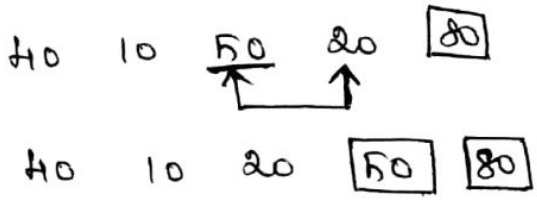
1. Consider an array of 'n' elements to be sorted.
2. Find the largest element in the list and then exchange it with the element placed at last position.
3. Next, it Searches for the next largest element and is interchanged with the element placed at second largest position.
4. This process is repeated for n-1 passes to sort all the elements.

Let us consider an array of numbers "80 10 50 20 40" to be sorted in ascending order using largest element method.

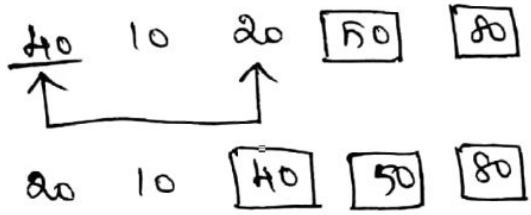
Pass-1:



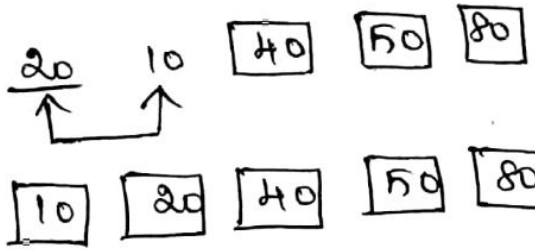
Pass-2:



pass-3:



Pass-4:



⟹ Sorted array.

Smallest element method:

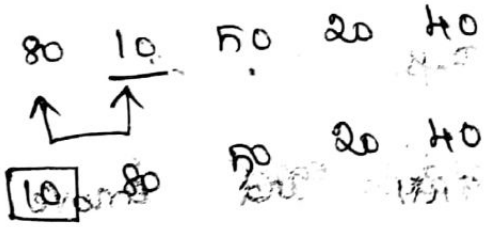
Here the process starts with finding the smallest element in the list, and placing it at the first position.

Working Principle:

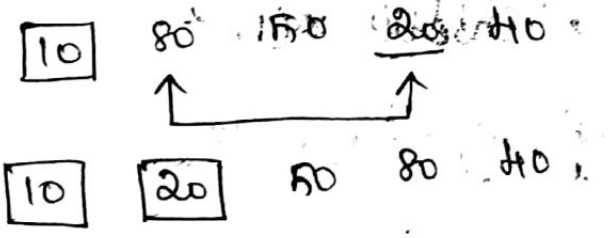
1. Consider an array of 'n' elements to be sorted.
2. Find the smallest element in the list and exchange it with the element at first position.
3. Then, search the second smallest element and interchange it with the element at second position.
4. Repeat the process until (n-1) passes to sort all the elements in the list.

Eg: "80 10 50 20 40"

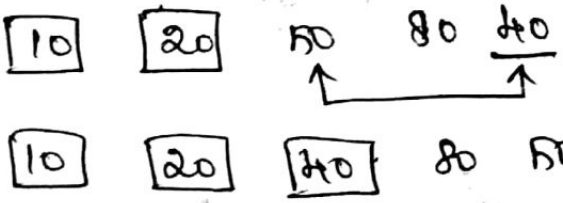
Pass 1:-



Pass 2:



pass 3:



pass 4: