Unit - IV

Graphs : Graph Implementation Methods. Graph Traversal methods.

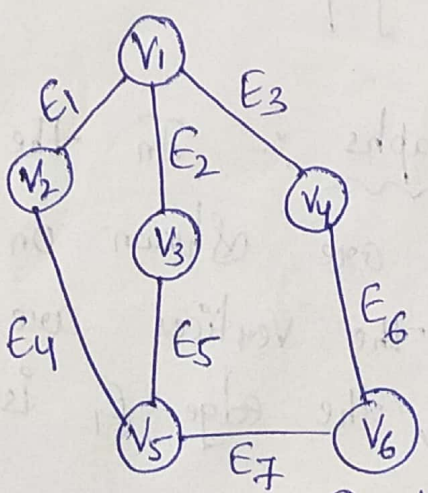
Sorting : Heap Sort, External Sorting - Model for external Sorting, Merge Sort.

Introduction to Graphs

A graph is a collection of two sets V and E where "V" is a finite non-empty set of vertices and "E" is a finite non-empty set of edges.

* Vertices are nothing but the nodes in the graph.
* Two adjacent vertices are joined by edges.
* Any graph is denoted as $G = \{V, E\}$.

Ex:-



Graph G

$G = \{\{V_1, V_2, V_3, V_4, V_5, V_6\}, \{E_1, E_2, E_3, E_4, E_5, E_6, E_7\}\}$

Vertex :- Individual data element of a graph is called as Vertex. Vertex is also known as node. In the above example graph $V_1, V_2, V_3, V_4, V_5, V_6$ are known as Vertices.

Edge : An edge is a Connecting link between two vertices. Edge is also known as Arc. In the above example graph $E_1, E_2, E_3, E_4, E_5, E_6, E_7$ are known as Edges. (ie) (V_1, V_2) (V_1, V_3) (V_1, V_4) (V_2, V_5) (V_3, V_5) (V_4, V_6) (V_5, V_6)

Types of Graphs

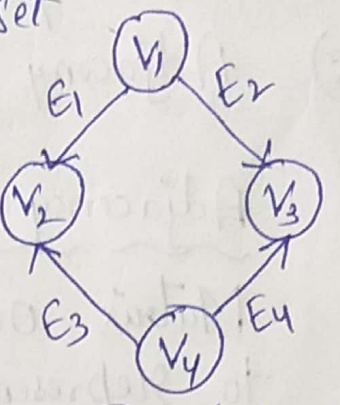
Basically graphs are of two types

1. Directed graphs
2. Undirected graphs.

Directed graphs :- In the directed graph the directions are shown on the edges. the edges between the vertices are Ordered. In this type of graph, the edge E_1 is in between the

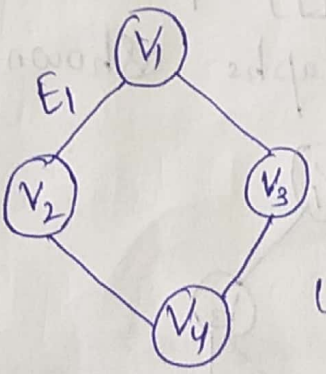
vertices V_1 and V_2. The V_1 is called head and V_2 is called the tail. Also V_1 is head and V_3 is tail and so on.

Now we can say E_1 is the set of (V_1, V_2) and not of (V_2, V_1).

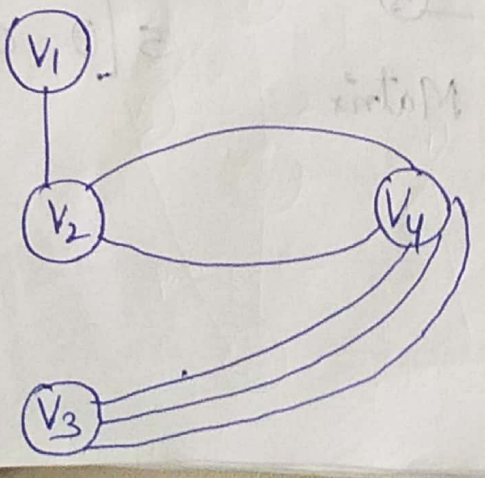


Directed graph.

Undirected Graph: The undirected graph, the edges are not Ordered. In this type of graph the edge E_1 is set of (V_1, V_2) or (V_2, V_1).



Undirected graph



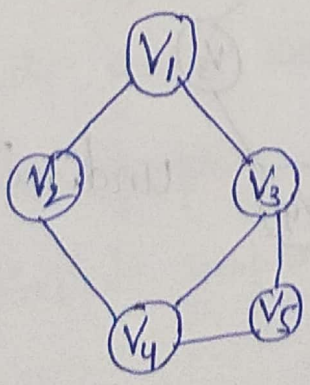
Multigraph

Graph Representations

1) Adjacency Matrix
2) Adjacency list

Adjacency Matrix :- In this representation, Matrix or two dimensional array is used to represent the graph.

Consider a graph "G" of n vertices and the matrix M. If there is an edge present between vertices V_i and V_j then M[i][j]=1 else M[i][j]=0. Note that for an Undirected graph if M[i][j] = 1 then for M[j][i] = 1. Here are some graphs shown by adjacency matrix.



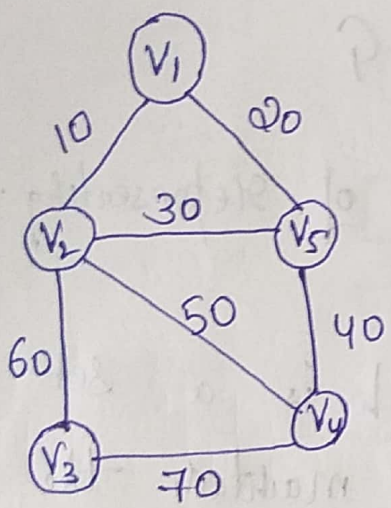
Adjacency Matrix

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

Adjacency Matrix for Weighted graph

In the weighted graph, weights or distances are given along every edge. Hence in an adjacency matrix representation any edge which is present between vertices V_i and V_j is denoted by its weight.

Hence M[i][j] = Weight of edge.



Weighted graph.

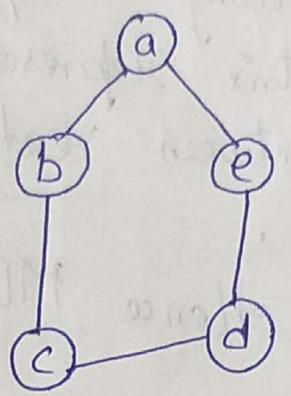
	1	2	3	4	5
1	0	10	0	0	20
2	10	0	60	50	30
3	0	60	0	70	0
4	0	50	70	0	40
5	20	30	0	40	0

Adjacency Matrix

Adjacency matrix representation for weighted graph

If there is no edge between V_i and V_j then M[i][j] = 0.

Adjacency list : In this representation, a linked list is used to represent a graph.



Graph G

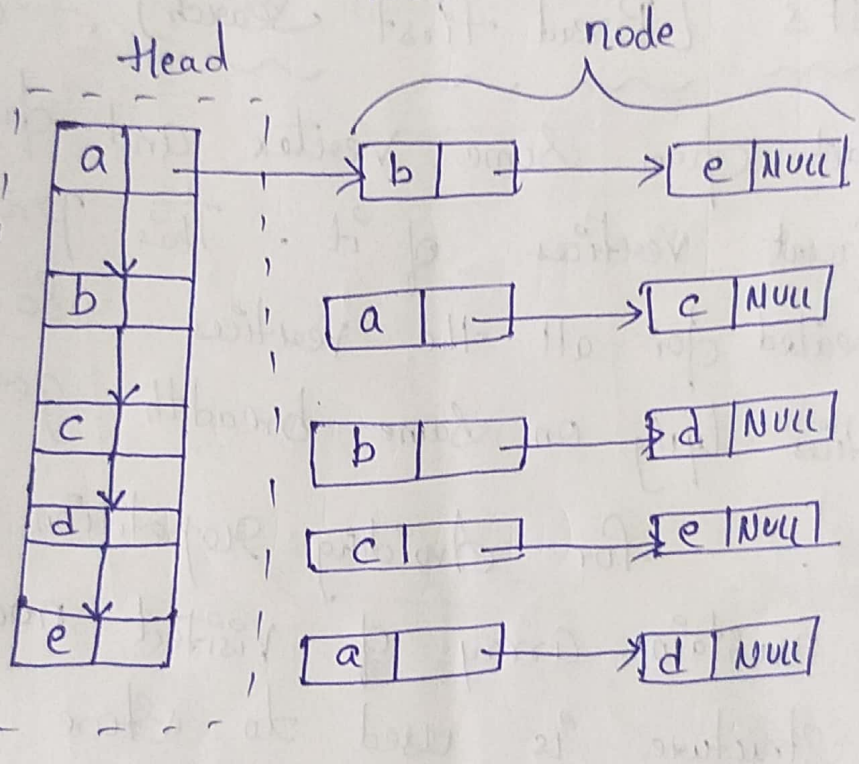
There are two methods of representing graph using adjacency list.

Method 1:- The graph is a set of vertices and edges, we will maintain the two structures for vertices and edges respectively.

Ex:- graph has the nodes as a, b, c, d, e. we will maintain the linked list of these head nodes as well as the adjacent nodes.

[Cont..d]

Explanation :- This is purely the adjacency list graph. The down pointer helps us to go to each node in the graph where as the next node is for going to adjacent node of each of the head node.



Adjacency list

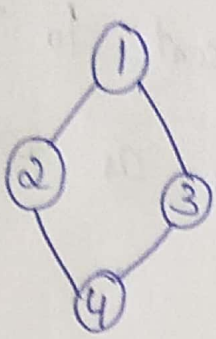
Graph Traversals

1) BFS 2) DFS

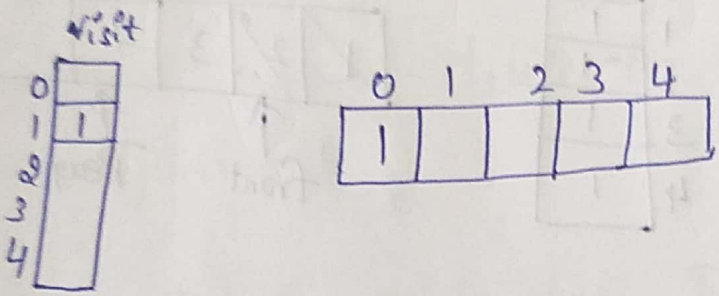
BFS [Bread first Search] :- In BFS we start from some vertex and find all the adjacent vertices of it. This process will be repeated for all the vertices. So that the vertices lying on same breadth get printed.

For Avoiding repetition of vertices, we maintain array of visited nodes. A queue data structure is used to store adjacent vertices.

Implementation of BFS :- In BFS the queue is maintained for storing the adjacent nodes and an array 'visited' is maintained for keeping the track of visited nodes. i.e Once a particular node is visited it should not be revisited again.

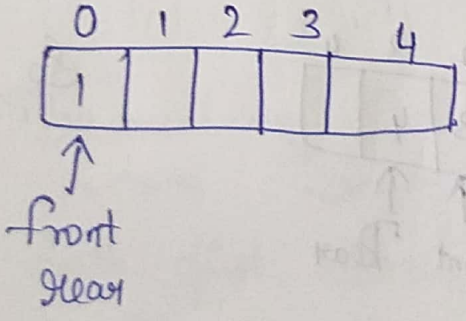


Step 1: Start with Vertex 1.



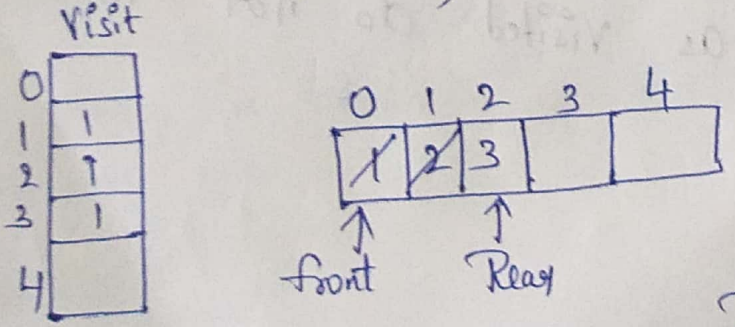
Inserted Vertex 1 in queue and marked the index 1 of visited array by 1.

Step 2:



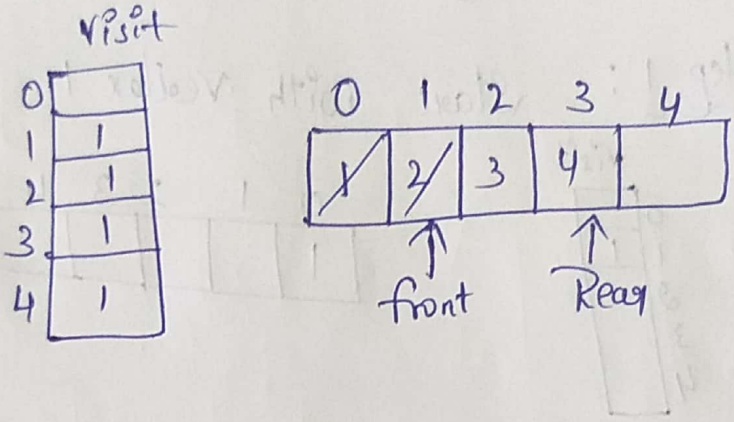
Delete '1' and print it. So '1' gets printed.

Step 3: find adjacent Vertices of Vertex 1 and mark them as Visited, Insert those in Queue.

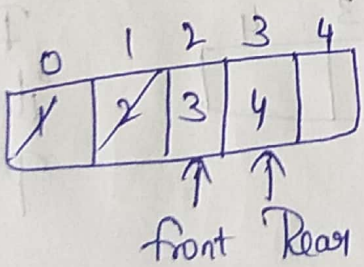


Increment front by 1 delete '2' from Queue and print it. So '2' gets printed.

Step 4: find Adjacent to '2' and insert those nodes in Queue as well as mark them as visited



Step 5: Increment front and delete the node Print it.

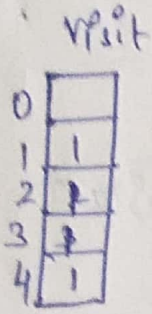


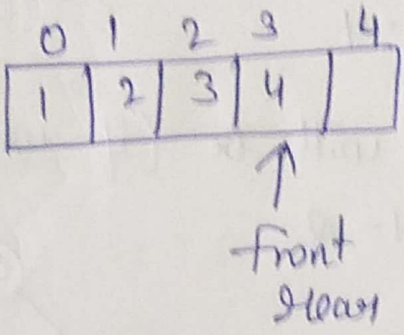
So '3' gets printed

Step 6: find adjacent to '3' i.e 4 check whether it is marked as visited. If it is marked as visited do not insert in the queue.

[Cont..d]

[Cont... d]





So, '4' gets Printed Since front = rear Stop the Procedure.

Increment front, delete the node from Queue and Print it.

So, Output will be - B-f-s for above graph as 1 2 3 4.

DFS Traversal of Graph

In depth first Search traversal we start from one Vertex and traverse the path as deeply as we can go. when there is no Vertex further, we traverse back and Search for unvisited Vertex.

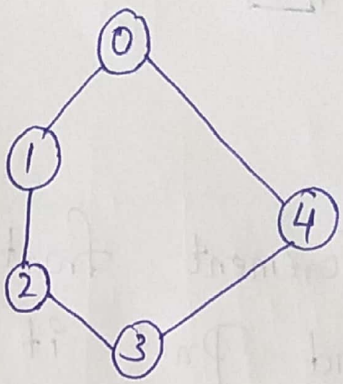
An array is maintained for storing the visited Vertex.

The DFS will be (if we start from Vertex 0)

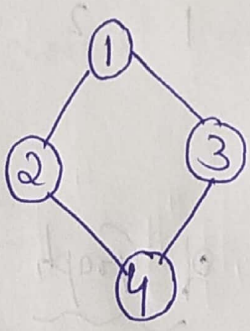
0-1-2-3-4

The DFS will be (if we start from vertex 3)

3-4-0-1-2



Explanation of logic for Depth first Traversal



In DFS the basic data structure for storing the adjacent nodes is Stack.

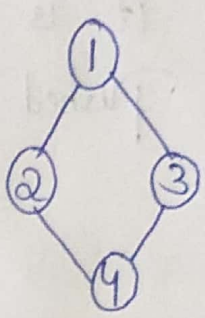
In Our Program we use a recursive call to DFS function. When a recursive call is invoked actually Push operation gets performed. When we exit from loop pop operation performs.

[Cont...d]

[Cont...d]

Step 1 : Start with Vertex 1, Print it So '1' gets Printed. Mark 1 as Visited.

	Visited
0	0
1	1
2	0
3	0
4	0

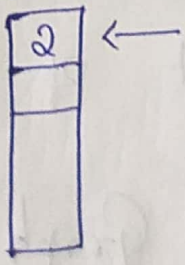


Graph 'G'

Step 2 : find adjacent Vertex to 1, Say i.e '2'. if it is not Visited, Call DFS(2) i.e 2 will get inserted in the stack, mark is as visited.

	Visited
0	0
1	1
2	1
3	0
4	0

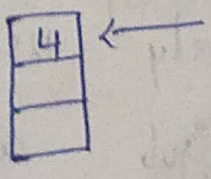
Stack



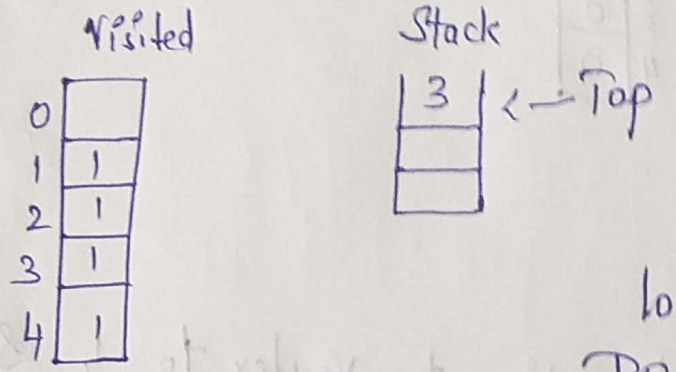
Step 3 : find adjacent to '2' i.e Vertex 4 if it is not Visited Call DFS (4) i.e, 4 will get pushed On to the stack mark it as Visited.

	Visited
0	0
1	1
2	1
3	0
4	1

Stack



Step 4: find adjacent to '4' ie Vertex 3 if it is not visited Call DFS(3) ie 3 will be Pushed Onto the Stack mark it visited.



After exiting the loop 3 will be Popped print '3'.

Since all the nodes are Covered Stop the Procedure.

So, Output of DFS is : 1 2 4 3.

Applications of Graph

1) In Computer networking Such as local Area Network [LAN], Wide Area Networking [WAN], internetworking.
2) In telephone Cabling graph theory is effectively used.
3) In job Scheduling Algorithms.

Depth-first Traversal

Algorithm

1) Initialize all the nodes to ready state and stack to empty

   State [V] = 1 [∵ 1 indicates Ready state]

2) Begin with any arbitrary node 's' in graph, Push it onto stack State [S] = 2 [∵ 2 indicates waiting state]

3) Repeat through step 5 while stack is not empty

4) Pop node N of stack and mark the status of node to be visited.

5) Push all nodes w adjacent to N into stack and mark their status as waiting.

   State [w] = 2

6) If the graph still contains nodes which are in ready state goto step 2.

7) Return.

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");
      getch();
```

Breadth First Traversal

Algorithm:

1) Initialize all nodes to ready state
   State[v] = 1 [Here v represents all nodes of graph]
2) Place starting node 's' in queue and change its state to waiting.

3) Repeat through step 5 until queue is not empty.

4) Remove a node N from queue and change its Status to visited.

State [N] = 3.

5) Add to queue all neighbours

Program for BFS [Breadth first Search]

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

```
        visited[q[f]] = 1;
        bfs(q[f++]);
    }
    }
    void main()
    {
        int v;
    printf("\n Enter the number of vertices:");
    scanf("%d", &n);
    for(i=1; i<=n; i++)
    {
        q[i] = 0;
        visited[i] = 0;
    }
    printf("\n Enter graph data in matrix form:\n");
    for(i=1; i<=n; i++)
    {
    for(j=1; j<=n; j++)
    {
    scanf("%d", &a[i][j]);
    }}
    printf("\n Enter the Starting Vertex:");
    scanf("%d", &v);
```

```
    bfs(v);
Printf("\n The node which are reachable are: \n");
for (i=1; i<=n; i++)
{
  if (visited[i])
Printf("%d\t", i);
  else
  {
   Printf("\n Bfs is not possible. Not all nodes
   are reachable");
   break;
  }
 }
}
```

Output

```
Enter the number of Vertices : 4
Enter graph data in matrix form:
1 1 1 1
0 1 0 0
```

Sorting

Sorting : The Sorting is a technique by which we expect the list of elements to be arrang as Ascending or Descending order.

Ascending Order : The elements are Sorted in low - high order.

Descending Order : The elements are Sorted in high - low order.

Heap Sort : Heap Sort is a Sorting method discovered by J. W. J. Williams. It works in two stages.

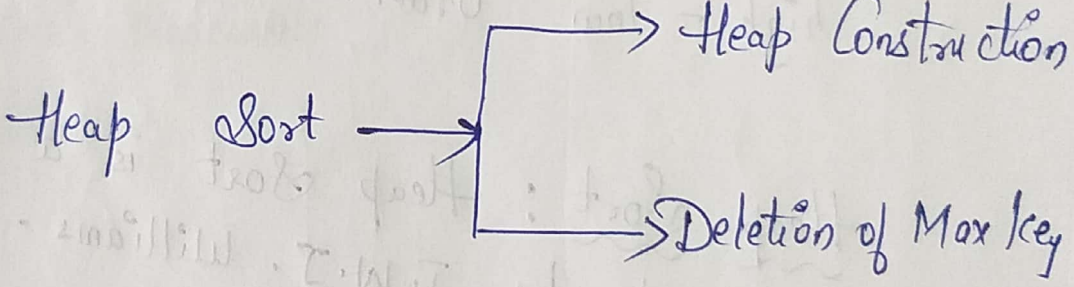
1) Heap Construction
2) Deletion of Maximum key

Heap Construction : first Construct a heap for given numbers.

Deletion of Maximum key : Delete root key always for (n-1) times to remaining heap. Hence we will get the elements in decreasing order.

For an array heap implementation of delete the element from heap and put the deleted element in the last position in array.

Thus after deleting all the elements one by one, if we collect these deleted elements in an array starting from last index of array.



Ex:-

1) Sort the elements using Heap Sort.

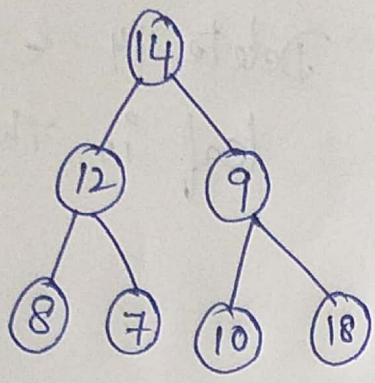
14, 12, 9, 8, 7, 10, 18.

Step 1: Heap Construction

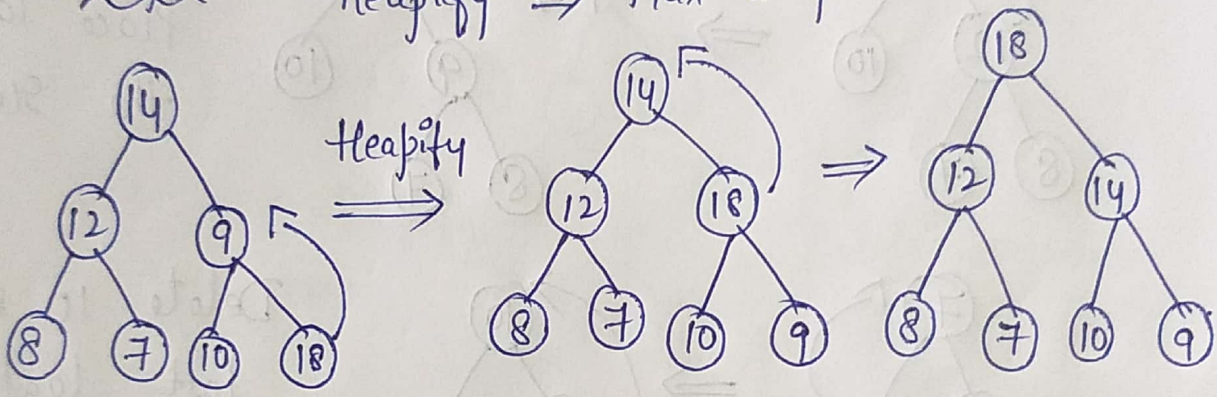
Here Construct the tree as in the given Sequence.

(Cont...d)

Heapify: Swaping the elements.



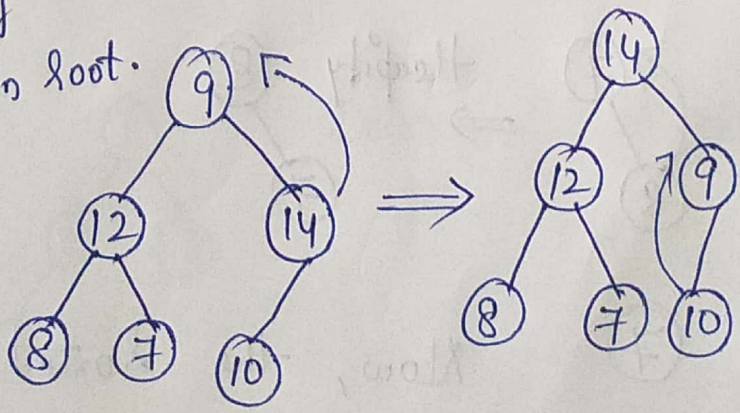
Step 2: Heapify ⇒ Max Heap



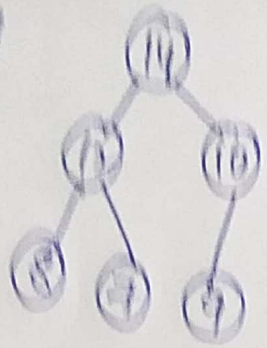
Max Heap Sort (ie) delete the largest element and keep it in Root node.

Step 3: Delete 18

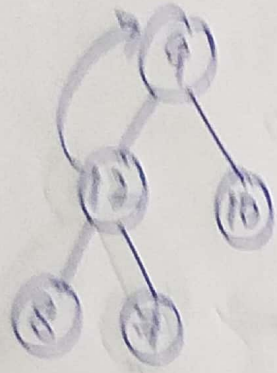
After deleting we have to choose last leaf of the tree in Root.



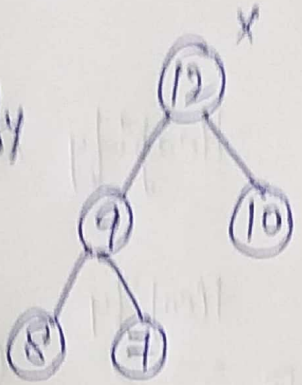
Heapify



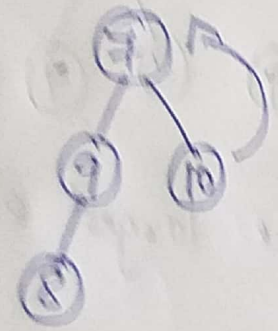
Delete 14 & Place last leaf in the Root node.

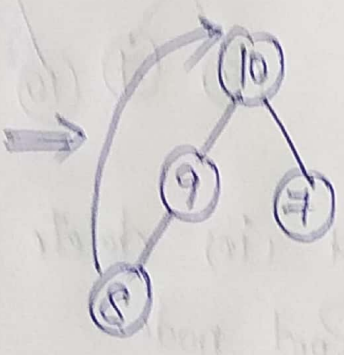


Heapify

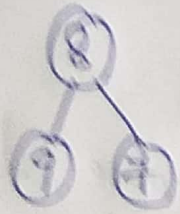


Delete 12 & Place last leaf in the root ie 7.

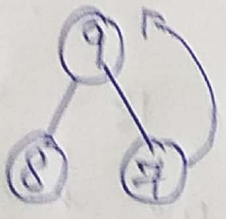




Delete 10 & Place last leaf in the root. ie 8

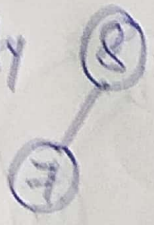


Heapify



Delete '9' & Place last leaf in the root. (ie) 7

Heapify



Delete '8' & Place last leaf in the Root. (ie) '7'.

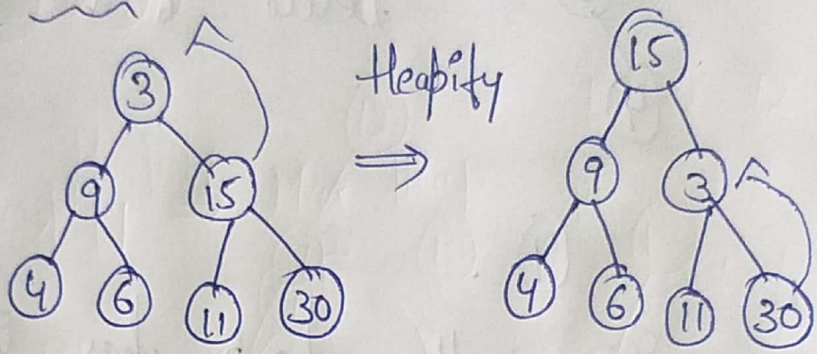
Now, the Sorted Order

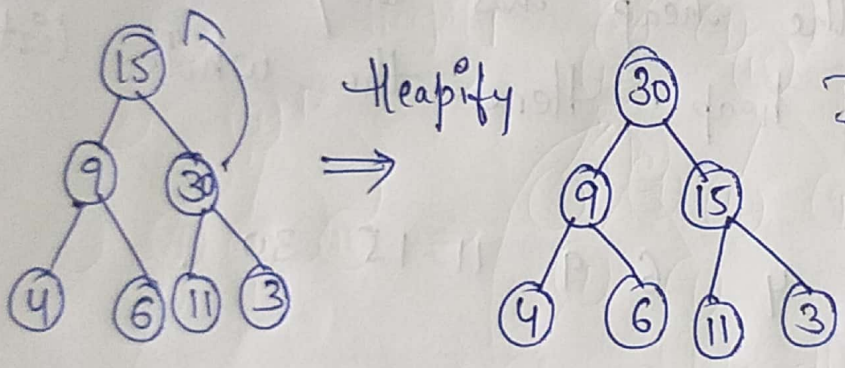
7, 8, 9, 10, 12, 14, 18.

Consider the following list using Heap Sort.

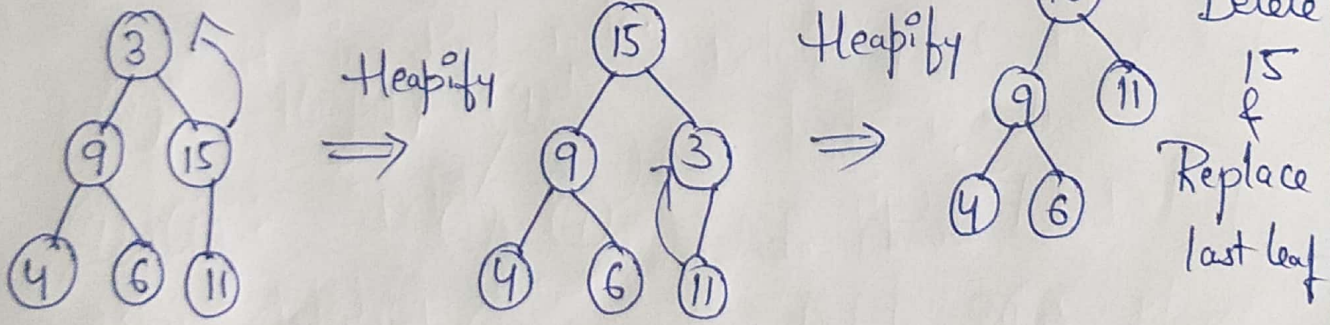
3, 9, 15, 4, 6, 11, 30

Heap Construction

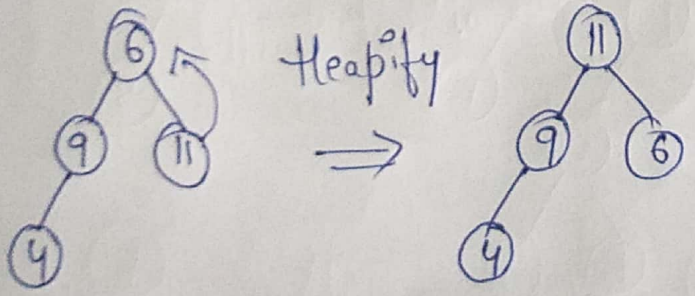




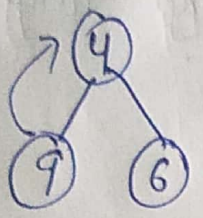
Delete 30 and Replace last leaf in the root.



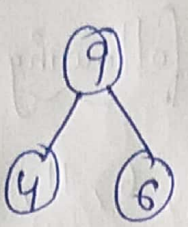
Delete 15 & Replace last leaf



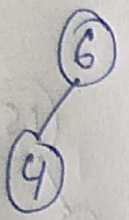
Delete 11 & Replace last leaf



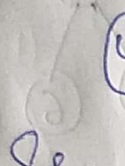
Heapify ⟹



Delete '9' & Replace last leaf.



Delete '6' & Replace '4'



Since, the heap has only one element it is already a heap. Hence the whole list is sorted.

3 4 6 9 11 15 30.

External Sorting

External Sorting Process is used when the number of elements (or records) to be Sorted are in large number, such that all of them cannot be accommodated in the internal memory of Computer. These files containing huge records to be Sorted are stored on external storage devices and external Sorting process is applied.

The external Sorting process performs

1) Bring few records [from external storage] into the main memory.
2) Apply internal Sort algorithm on those records to generate "runs".
3) Write "runs" onto the external storage devices.
4) Merge the "runs" generated in the steps
5) Repeat Steps ~~4 to~~ 2, 3, 4 until all runs are merged and a single "run" which is a Sorted file of records are left out.

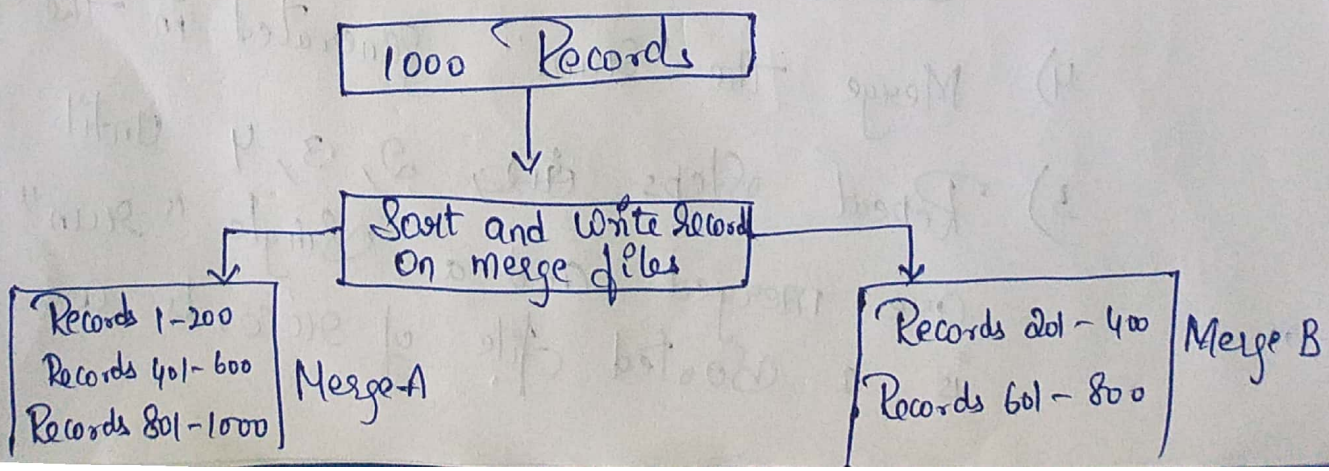
Ex:-

Consider a file containing 1000 records. But main memory can accommodate only 200 records at a time. Therefore, External Sorting technique is applied as follows.

1) Read the first 200 records from the input file, sort them and write them to an output merge file [say merge A].

2) Read another 200 records, sort them and write them to an Alternate merge file [say merge B].

3) Again another 200 records are read from input file, sorted and written to merge file Merge A. This process is repeated until all records are read and sorted.



Examples of external Sorting algorithms

1) Multiway Merge
2) Polyphase Merge.

Multiway Merge: A multiway merge is an n-way merge that uses additional tapes and makes Sorting of input more Simpler by minimizing the total number of passes.

let $T_{x_1}, T_{x_2}, T_{x_3}, \ldots T_{x_n}$,

$T_{y_1}, T_{y_2}, T_{y_3}, \ldots T_{y_n}$ be the tapes.

Out of these $2n$ tapes, half of them will works as input tapes and the other half will work as Output tapes.

Let T_{x_1} work as input tape.

T_{x_1} 75 88 1 80 5 89 8 90 15 61 32 69 7

T_{x_2}
:
T_{x_n}
T_{y_1}
T_{y_2}
:
T_{y_n}

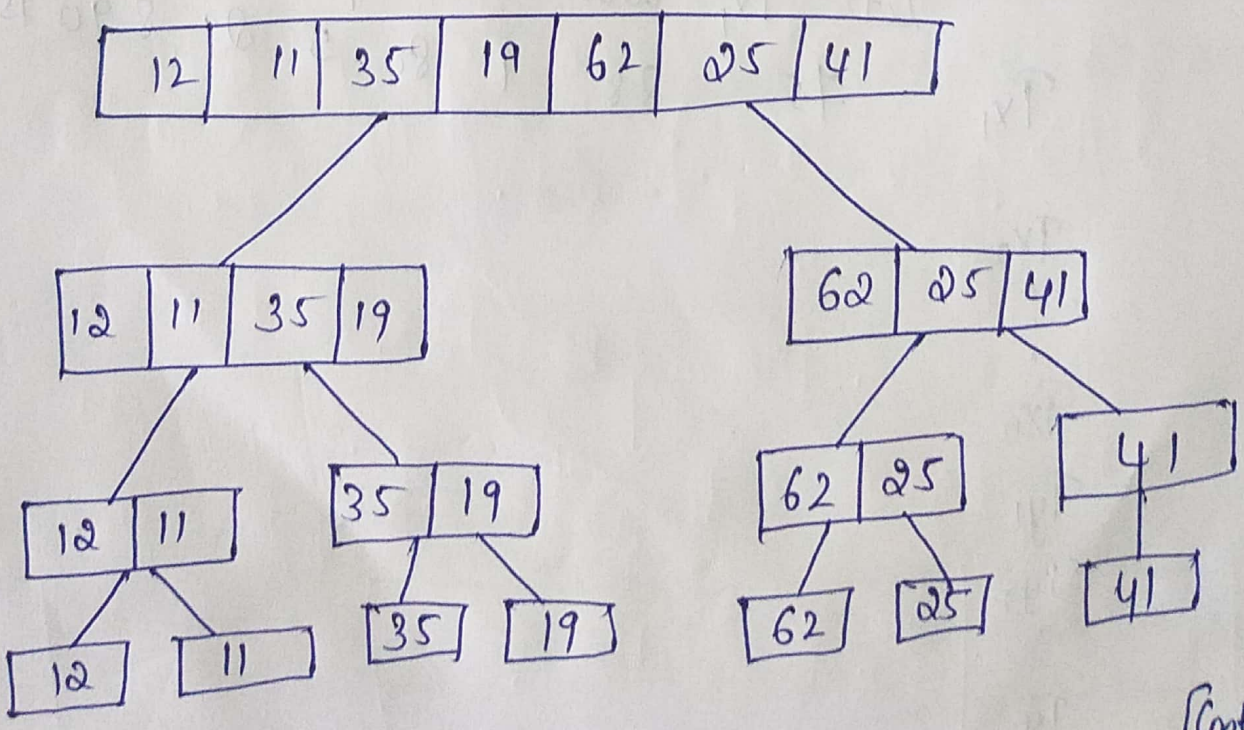
Merge Sorting

Merge Sort

Merge Sort is an external Sorting algorithm which makes use of Secondary Storage. It uses divide and Conquer approach to Sort a given list. It divides the list into n Sublists of equal size until only one element is left in Sublist. Then it recursively applies merge Sort on two Sorted Sublists to form a Combined list.

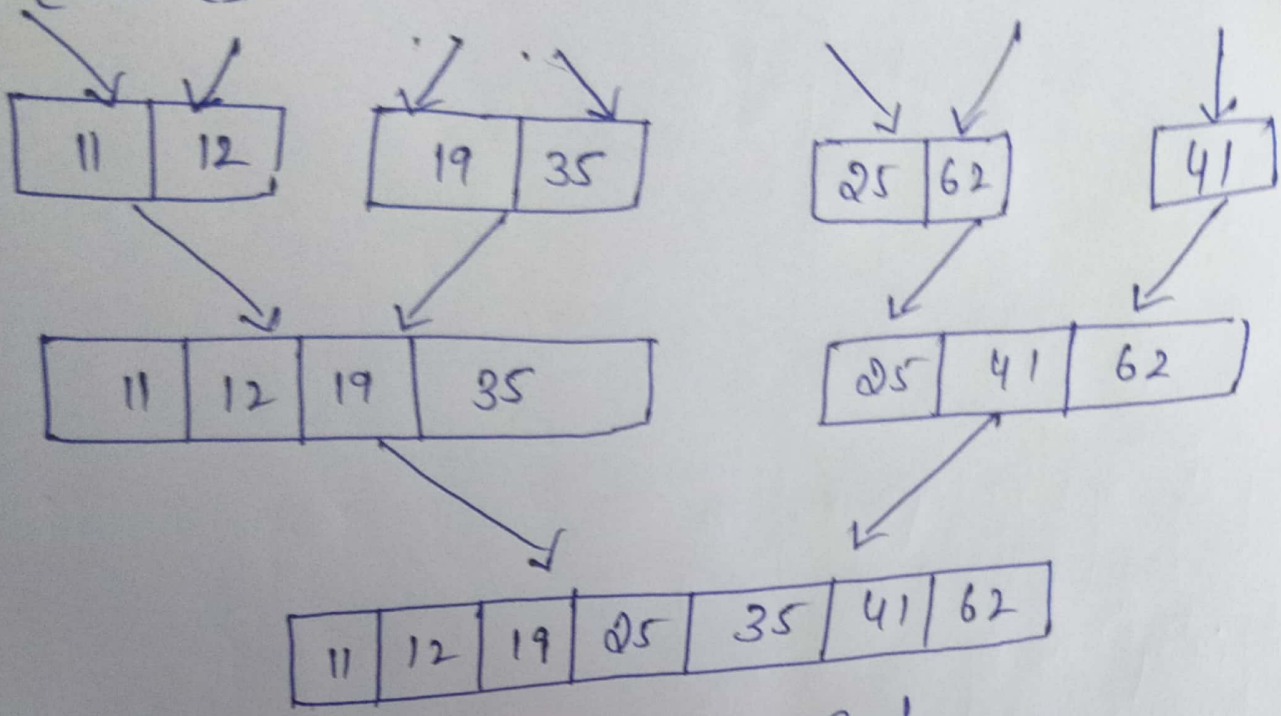
Ex:- Apply Merge Sort on the following list

12, 11, 35, 19, 62, 25, 41



[Cont..d]

[Cont--d]



Merge Sort .