UNIT-4
GRAPHS

In this chapter, we turn our attention to a data structure – Graphs - that differs from all of the other in one major concept: each node may have multiple predecessors as well as multiple successors.

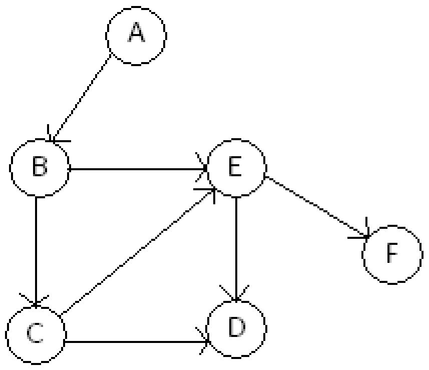
Graphs are very useful structures. They can be used to solve complex routing problems, such as designing and routing airlines among the airports they serve. Similarly, they can be used to route messages over a computer network from one node to another.

Basic Concepts:

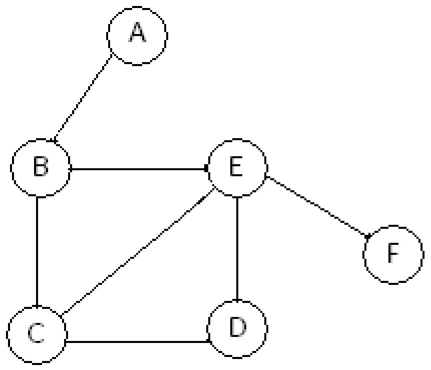
A graph is a collection of nodes, called **vertices** and a collection of segments called **lines** connecting pair of vertices. In other words a graph consists of two sets, a set of vertices and set of lines.

Graphs may be either directed or undirected.

- A **directed graph** or **digraph** is a graph in which each line has a direction (arrow head) to its successor. The line in a directed graph is known as **arc**. The flow along the arc between two vertices can follow only the in directed direction.
- An **undirected graph** is a graph in which there is no direction (arrow head) on any of the lines, which are known as **edges**. The flow between two vertices can go in either direction.



Directed graph



Undirected graph

A path is a sequence of vertices in which each vertex is adjacent to the next one.

For example: {A, B, C, E} is a one path and {A, B, E, F} is another.

Two vertices in a graph are said to be **adjacent vertices** (or neighbors) if there is a path of length 1 connecting them.

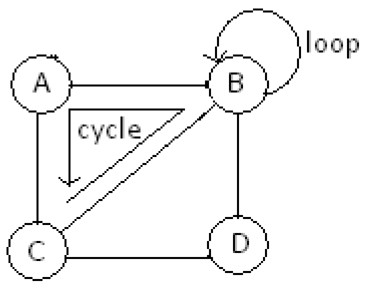
Consider the above diagrams

In directed graph, B is adjacent to A, where as E is not adjacent to D; but D is adjacent to E.

In undirected graph, E and D are adjacent, but D and F are not.

A **cycle** is a path, it start with vertex and ends with same vertex.

Example:

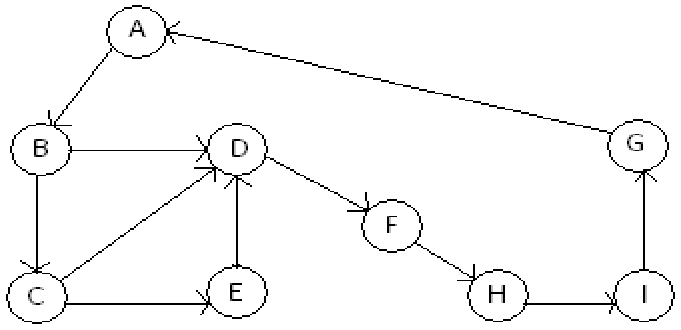


A-B-C-A is a cycle

A **loop** is a special case of cycle in which a single arc begins and ends with the same vertex. In a loop the end points of the line are the same.

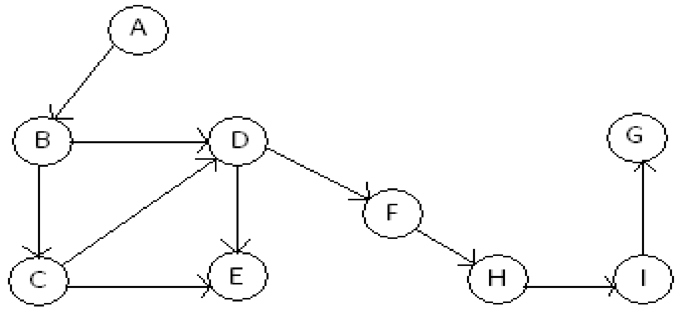
Two vertices are said to be **connected** if there is a path between them. A graph is said to be connected if, ignoring direction, there is a path from any vertex to any other vertex.

A directed graph is **strongly connected** if there is a path from each vertex to every other vertex in the digraph.



strongly connected graph

A directed graph is **weakly connected** if at least two vertices are connected (A connected undirected graph would always be strongly connected, so the concept is not normally used with undirected graphs)



weakly connected graph

A graph is a **disjoint graph** if it is not connected

The **degree** of a vertex is the no/of lines incident to it

The **out - degree** of a vertex in a digraph is the no. of arcs leaving the vertex

The **in - degree** is the no. of arcs entering the vertex

For example: for vertex B; degree = 3, in - degree = 1, out - degree = 2

NOTE: A tree is a graph in which each vertex has only one predecessor; how ever a graph is not a tree.

Operations on Graphs:

There are six primitive graph operations that provide the basic modules needed to maintain a graph. They are

1. Insert a vertex
2. Delete a vertex
3. Add an edge
4. Delete an edge
5. Find a vertex
6. Traverse a graph

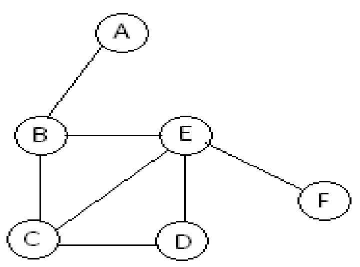
Graph Storage Structure:

To represent a graph, we need to store two sets. The first set represents the vertices of the graph and the second set represents the edges or arcs. The two most common structures used to store these sets are arrays and linked lists. Although the arrays offer some simplicity this is a major limitation.

Adjacency Matrix:

The adjacency matrix uses a vector (one – dimensional array) for the vertices and a matrix (two – dimensional array) to store the edges. If two vertices are adjacent – that is if there is no edge between them, intersect is set to 0.

Adjacency matrix for non - directed graph



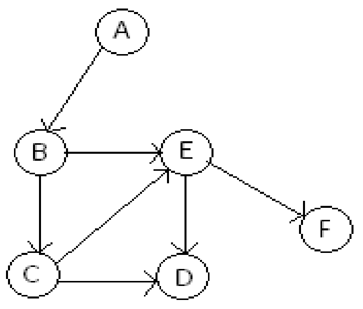
A
B
C
D
E
F

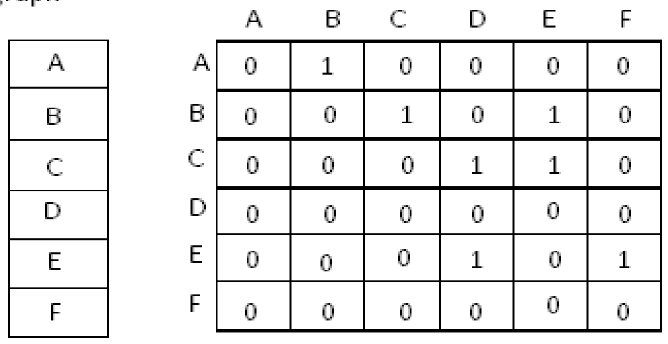
	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

If the graph is directed, the intersection in the adjacency matrix indicates the direction

In the below diagram, there is an arc from sources vertex B to destination vertex C. In the adjacency matrix, this arc is seen as a 1 in the intersection from B (on the left) to C (on the top). Because there is no arc from C to B, however, the intersection from C to B is 0.

Adjacency matrix for directed graph





vertex vector
A
B
C
D
E
F

vertex vector

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

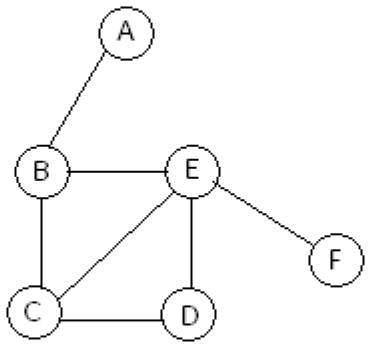
Adjacency matrix

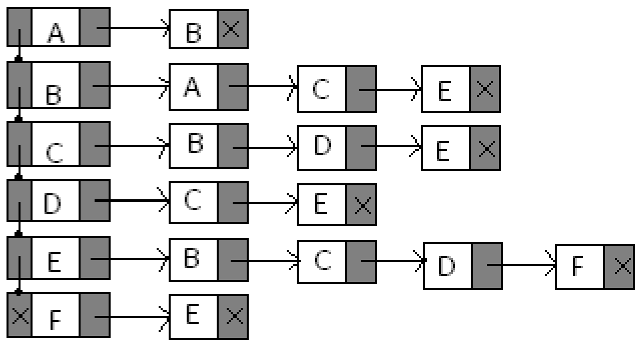
NOTE: In adjacency matrix representation, we use a vector to store the vertices and a matrix to store the edges.

In addition to the limitation that the size of graph must be know before the program starts, there is another serious limitation in the adjacency matrix: only one edge can be stored between any two vertices. Although this limitation does not prevent many graphs from using the matrix format, some network structures require multiple lines between vertices.

Adjacency list:

The adjacency list uses a two – dimensional ragged array to store the edges. An adjacency list is shown below.





vertex list Adjacency list

The vertex list is a singly linked list of vertices in the list. Depending on the application, it could also be implemented using doubly linked lists or circularly linked lists. The pointer at the left of the list links the vertex entries. The pointer at the right in the vertex is a head pointer to a linked list of edges from the vertex. Thus, in the non – directed graph on the left in above figure there is a path from vertex B to vertices A, C, and E. To find these edges in the adjacency list, we start at B's vertex list entry and traverse the linked list to A, then to C, and finally to E.

NOTE: In the adjacency list, we use a linked list to store the vertices and a two – dimensional linked list to store the arcs.

Traverse graph:

There is always at least one application that requires that all vertices in a given graph be visited; as we traverse the graph, we set the visited flag to on to indicate that the data have been processed

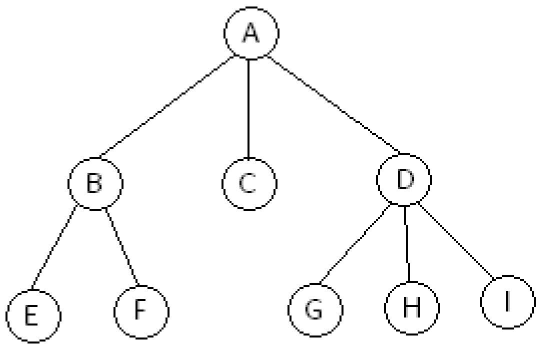
That is traversal of a graph means visiting each of its nodes exactly once. This is accomplished by visiting the nodes in a systematic manner

There are two standard graph traversals: **depth first** and **breadth first**. Both use visited flag

Depth – First Traversal:

In the depth – first traversal, we process all of a vertex's descendants before we move to an adjacent vertex. This concept is most easily seen when the graph is a tree

In the below figure we show the tree pre – order traversal processing sequence, one of the standard depth – first traversals



Depth - first traversal: A B E F C D G H I

In a similar manner, the depth – first traversal of a graph starts by processing the first vertex; we select any vertex adjacent to the first vertex and process it. This continues until we found no adjacent entries

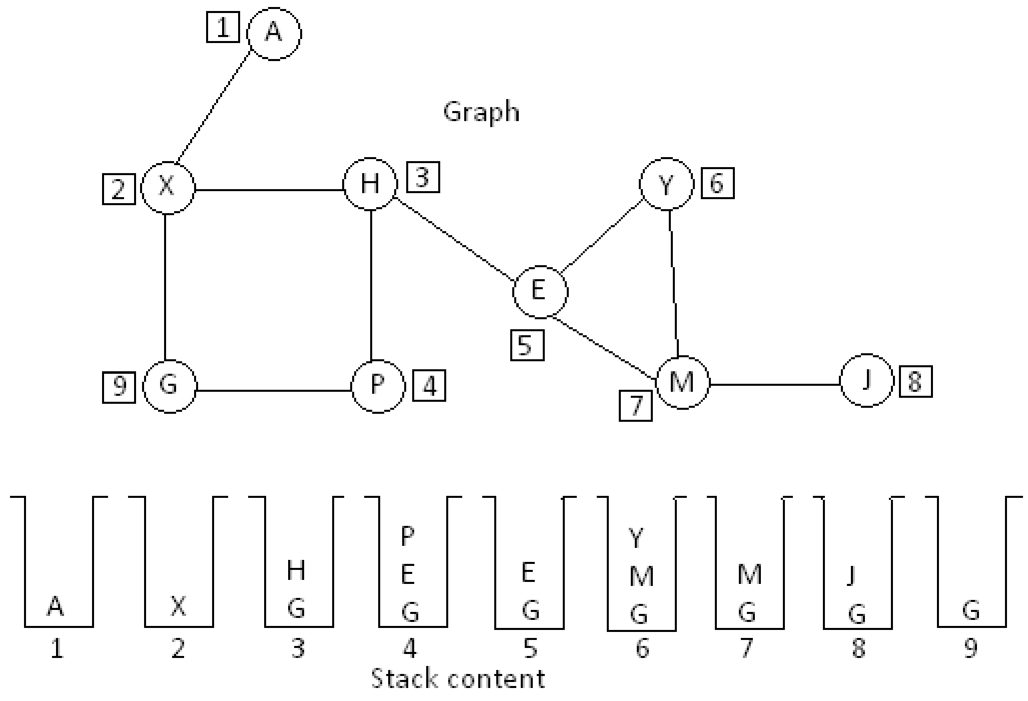
This is similar to reaching a leaf in a tree. We require a stack to complete the traversal
i.e. last – in – first – out (LIFO) order

Let's trace a depth – first traversal through the graph in below figure the numbering in the box next to a vertex indicates the processing order

Trace of the DFS:

1. We begin by pushing the first vertex, into the stack
2. We then loop, pop the stack and after processing the vertex, push all of the adjacent vertices into the stack
3. When the stack is empty traversal is completed

NOTE: In the depth – first traversal, all of a node’s descendents are processed before moving to an adjacent node



Depth - first traversal: A X H P E Y M J G

Consider the above graph, let node A be the starting vertex

1. Begin with node A push onto stack
2. While stack not equal to empty

   Pop A; state A is visited

   Push nodes adjacent to A to stack and make their state waiting
3. Pop X; state B is visited

   Push nodes adjacent to X into stack
4. Pop H; state H is visited

   Push nodes adjacent to H into stack already G is in waiting state, then push nodes E and P
5. Pop P; state P is visited

Push nodes adjacent to P are H, G, E; H is already in visited state, G and E are in waiting state

6. Pop E; state E is visited

   Push adjacent nodes, H is already visited, so push Y and M into the stack

7. Pop Y; state Y is visited

   Push nodes adjacent to Y into stack, E is visited, M already in waiting state

8. Pop M; state M is visited

   Push nodes adjacent to M, which is J

9. Pop J; state J is visited

   No nodes are there to be process

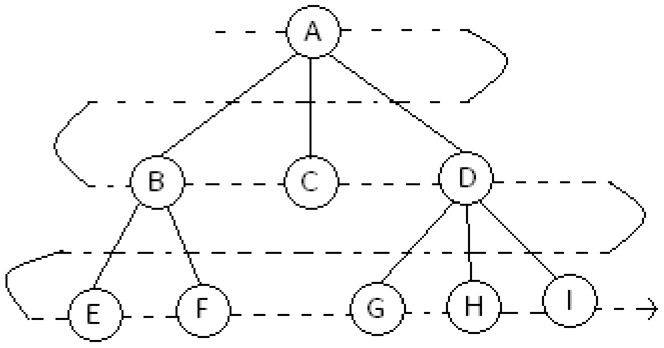
10. Pop G; state G is visited

    Now the stack is empty

The depth – first order of the visited nodes are **A X H P E Y M J G**

Breadth – First traversal:

In the breadth – first traversal of a graph, we process all adjacent vertices of a vertex before going to the next level. We first saw the breadth – first traversal of a tree as shown in below



Breadth - first traversal: A B C D E F G H I

This traversal starts at level 0 and then processes all the vertices in level 1 before going on to process the vertices in level 2.

The breadth – first traversal of a graph follows the same concept, begin by picking a starting vertex A after processing it, process all of its adjacent vertices and continue this process until get no adjacent vertices

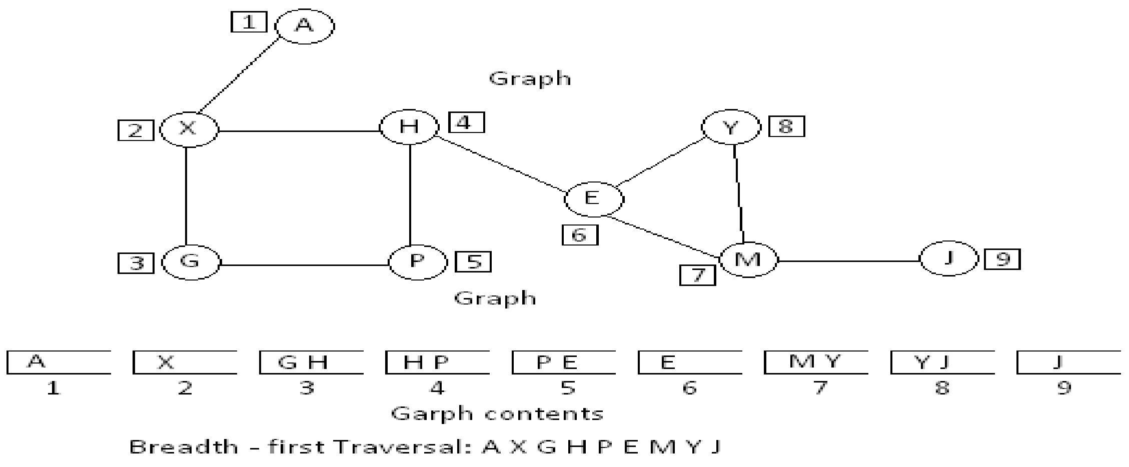
The breadth – first traversal uses a queue rather than a stack. As we process each vertex, we place all of its adjacent vertices in the queue. Then to select the next vertex to be processed, we delete a vertex from the queue and process it.

Trace of the BFS:

1. We begin by enqueuing vertex A in the queue
2. We then loop, dequeuing the queue and processing the vertex from the front of the queue. After processing the vertex, we place all of its adjacent vertices into the queue. Thus in the above diagram we dequeue vertex X, process it, and then place vertices G and H in the queue.
3. When the queue is empty, the traversal is complete.

NOTE: In the breadth – first traversal, all adjacent vertices are processed before processing the descendents of a vertex.

Let's trace this logic through the graph in below figure:



Algorithms:

Depth – First Search:

```
Policy: Don't push nodes twice
// non-recursive, preorder, depth-first search
void dfs (Node v) {
  if (v == null)
    return;
  push(v);
  while (stack is not empty) {
    pop(v);
    if (v has not yet been visited)
      mark&visit(v);
    for (each w adjacent to v)
      if (w has not yet been visited && not yet stacked)
        push(w);
  } // while
} // dfs
```

Breadth-First Search:

```
// non-recursive, preorder, breadth-first search
void bfs (Node v) {
  if (v == null)
     return;
  enqueue(v);
  while (queue is not empty) {
    dequeue(v);
    if (v has not yet been visited)
      mark&visit(v);
    for (each w adjacent to v)
      if (w has not yet been visited && has not been queued)
        enqueue(w);
  } // while
} // bfs
```

SORTING

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order.

Sorting can be classified in two types;

Internal Sorts:- This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting. There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints.

There are 3 types of internal sorts.

(i) **SELECTION SORT :-** Ex:- Selection sort algorithm, Heap Sort algorithm

(ii) **INSERTION SORT :-** Ex:- Insertion sort algorithm, Shell Sort algorithm

(iii) **EXCHANGE SORT :-** Ex:- Bubble Sort Algorithm, Quick sort algorithm

External Sorts:- Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together. **Ex:- Merge Sort**

Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960‘s. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by “divide and conquer” technique.
The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.
The function partition () makes use of two pointers up and down which are moved toward each other in the following fashion:
1. Repeatedly increase the pointer ‘up‘ until a*up+ >= pivot.
2. Repeatedly decrease the pointer ‘down‘ until a*down+ <= pivot.
3. If down > up, interchange a[down] with a[up]
4. Repeat the steps 1, 2 and 3 till the ‘up‘pointer crosses the ‘down‘ pointer. If ‘up‘ pointer crosses ‘down‘ pointer, the position for pivot is found and place pivot element in ‘down‘ pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array ‘a‘ between positions ‘low‘ and ‘high‘.

1. It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.
2. Here we choose the first element as the _pivot‘. So, pivot = x*low+. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . . x[high].
3. It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).
4. It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . x[high] between positions j+1 and high.

The time complexity of quick sort algorithm is of ***O(n log n)***.

Example:

Select first element as the pivot element. Move ‛up‘pointer from left to right in search of an element larger than pivot. Move the ‛down‘pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the ‛up‘ pointer crosses the ‛down‘ pointer. If ‛up‘ pointer crosses ‛down‘ pointer, the position for pivot is found and interchange pivot and element at ‛down‘ position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						dow n			swap up & down
pivot				04						79			
pivot					up			dow n					swap up & down
pivot					02			57					
pivot						dow n	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					do wn	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot & down
	pivot	down	up										

	(04)	**06**											swap pivot & down
	04 pivot down, up												
				16 pivot, down, up									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45)	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	56	(58	79	70	57)	
							45						swap pivot & down
							pivot, down, up						
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	58	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Algorithm

Sorts the elements a[p],..........,a[q] which reside in the global array a[n] into ascending order. The a[n + 1] is considered to be defined and must be greater than all elements in a[n]; a[n + 1] = + ∝

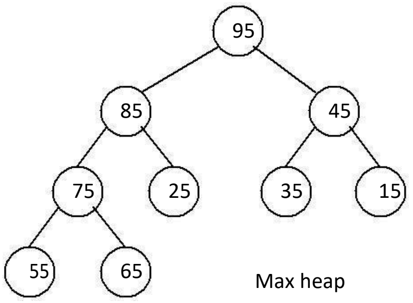
```
quicksort (p, q)

{
    if ( p < q ) then
    {
        call j = PARTITION(a, p, q+1); // j is the position of the partitioning element

        call quicksort(p, j –
        1); call quicksort(j + 1
        , q);
    }
}
partition(a, m, p)

{
    v = a[m]; up = m; down = p;            // a[m] is the partition element

    do
    {
        repeat
            up = up + 1;
        until (a[up] ≥ v);

        repeat
            down = down --
        1; until (a[down] ≤ v);
        if (up < down) then call interchange(a,
     up, down); } while (up ≥ down);

    a[m] = a[down];

    a[down] = v;

    return (down);
}

interchange(a, up, down)

{
    p = a[up];
    a[up] = a[down];
    a[down] = p;
}
```

Heap and Heap Sort:

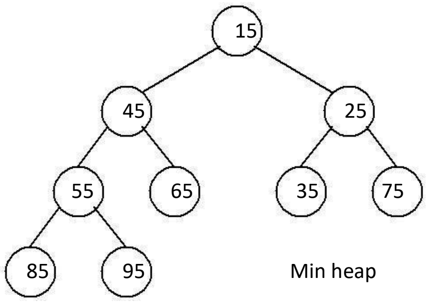
Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



Max heap



Min heap

A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

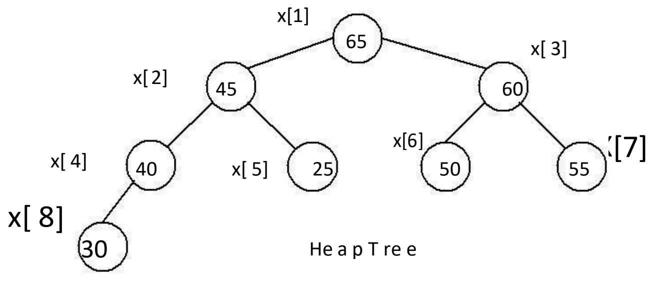
Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location *i* can be found in location 2**i*.
- The right child of an element stored at location *i* can be found in location 2**i*+1.
- The parent of an element stored at location *i* can be found at location floor(*i*/2).

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



He a p T re e

Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent‘s value, and to be a max heap, parent‘s value greater than child‘s value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

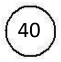
Max_heap_insert (a, n)

```
{
        //inserts the value in a[n] into the heap which is stored at a[1] to a[n-
        1] int i, n;
        i = n;
        item = a[n];
        while ( (i > 1) and (a[ i/2 ] < item ) do
        {
                a[i] = a[  i/2  ] ;                     // move the parent
                down i =  i/2                           ;
        }
        a[i]   =
        item  ;
        return
        true ;
}
```

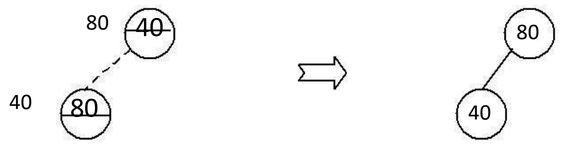
Example:

Form a heap using the above algorithm for the data: 40, 80, 35, 90, 45, 50, 70.

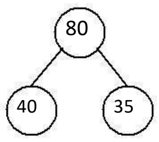
1. Insert 40:



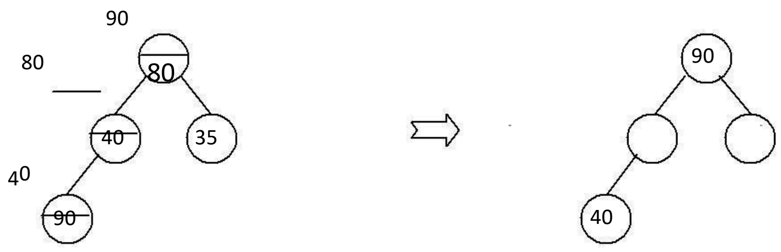
2. Insert 80:



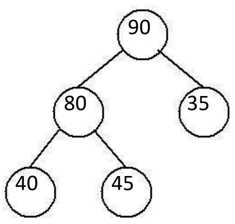
3. Insert 35:



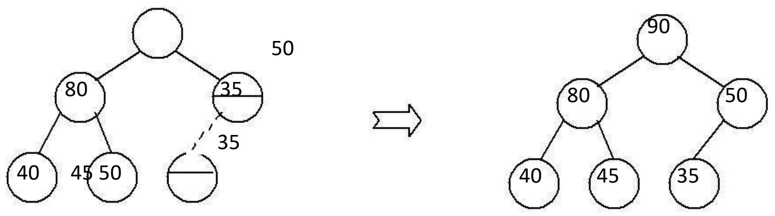
4. Insert 90:



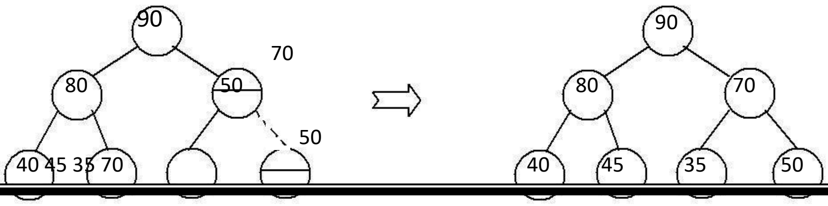
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
    - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
    - Make X as the current node.
    - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

delmax (a, n, x)

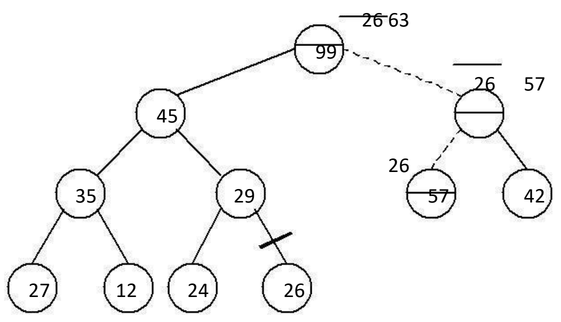
```
// delete the maximum from the heap a[n] and store it in x
{
        if (n = 0) then
        {
                write   (—heap   is
                empty‖);      return
                false;
        }
        x = a[1]; a[1] = a[n];
        adjust (a, 1, n-
        1);      return
        true;
}
```

adjust (a, i, n)

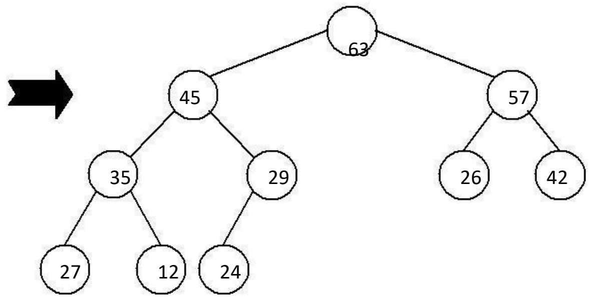
```
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to
form a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i ;
        item    =
        a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j          j + 1;
                        // compare left and right child and let j be the
                larger child if (item ≥ a (j)) then break;
                                        // a position for item is found
                else a[ j / 2 ] = a[j] // move the larger child up a level j = 2 *
                j;

        }
        a [ j / 2 ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged.

Now, 26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appears as the leave node, hence re-heap is completed.



De le t ing t he no de w it h data 99

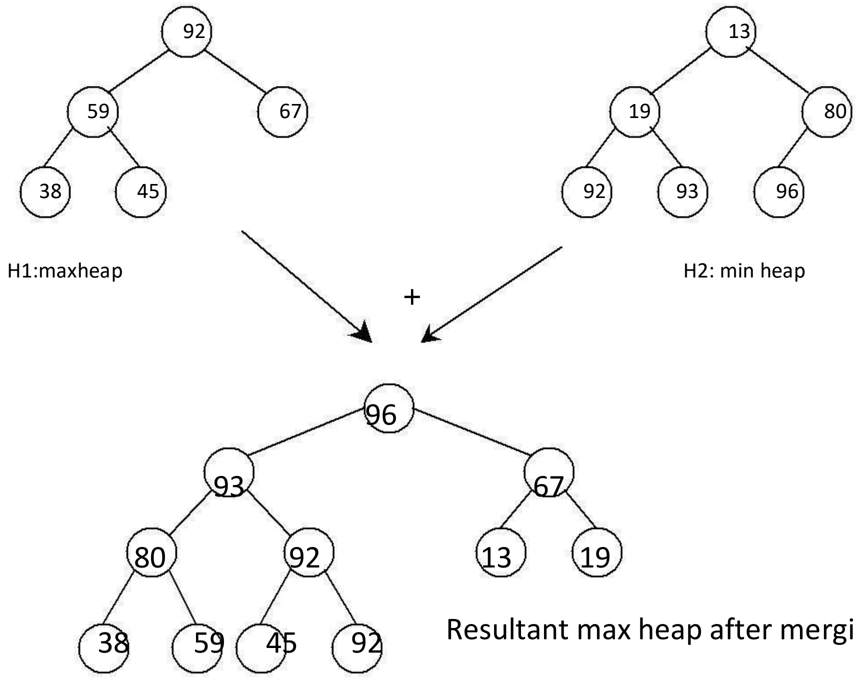


Aft er De le t io n of no de w it h data 99

Merging two heap trees:

Consider two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap. Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.



Resultant max heap after merging H1 and H2

Application of heap tree:

They are two main applications of heap trees known are:

1. Sorting (Heap sort) and Priority queue implementation

HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2. a. Remove the top most item (the largest) and replace it with the last element in the heap.

   b. Re-heapify the complete binary tree.

   c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements a[n]. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

heapsort(a, n)

```
{
        heapify(a, n);
        for i = n to 2 by – 1 do
        {
                temp    =
                a[i];   a[i]
                =     a[1];
                a[1]      =
                temp;

                adjust (a, 1, i – 1);
        }
}
```

heapify (a, n)

```
//Readjust the elements in a[n] to form a heap.
{
        for i        n/2 to 1 by – 1 do adjust (a, i, n);
}
```

adjust (a, i, n)

// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form a single heap, $1 \leq i \leq n$. No node has an address greater than n or less than 1. //

```
{
        j = 2 *i ;
        item =
        a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j            j + 1;
                        // compare left and right child and let j be the
                larger child if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[ j / 2 ] = a[j] // move the larger child up a level j = 2 *
```

```
                    j;

            }
            a [ j / 2 ] = item;
}
```

Time Complexity:

Each ‗n‘ insertion operations takes O(log k), where ʹk‘ is the number of elements in the heap at the time. Likewise, each of the ʹn‘ remove operations also runs in time O(log k), where ʹk‘ is the number of elements in the heap at the time.

Since we always have k ≤ n, each such operation runs in O(log n) time in the worst case.

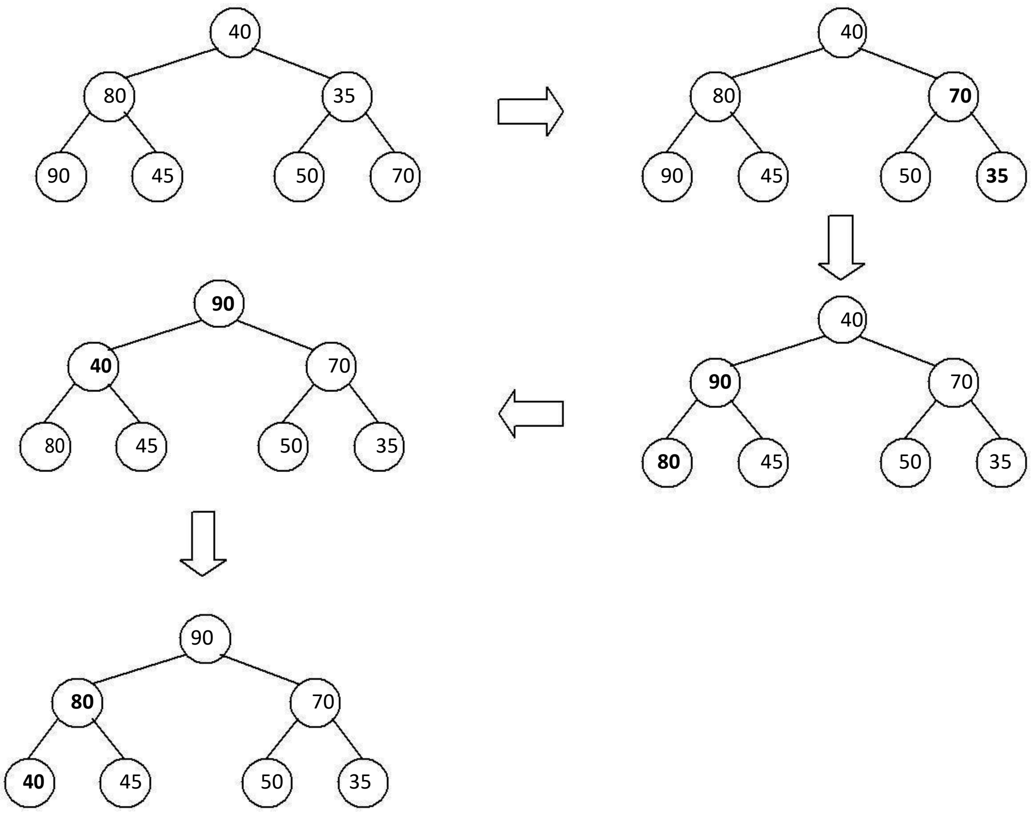
Thus, for ʹn‘ elements it takes O(n log n) time, so the priority queue sorting algorithm runs in O(n log n) time when we use a heap to implement the priority queue.

Example 1:

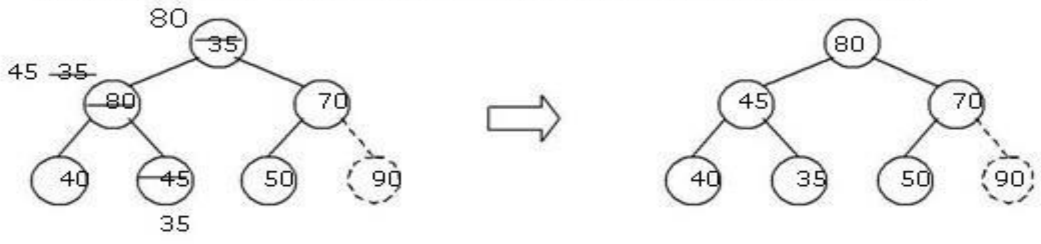
Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

Solution:

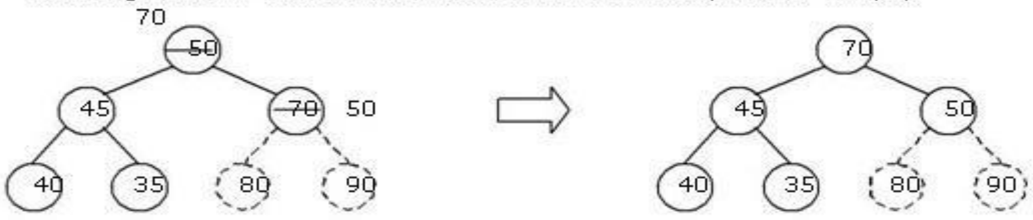
First form a heap tree from the given set of data and then sort by repeated deletion op:



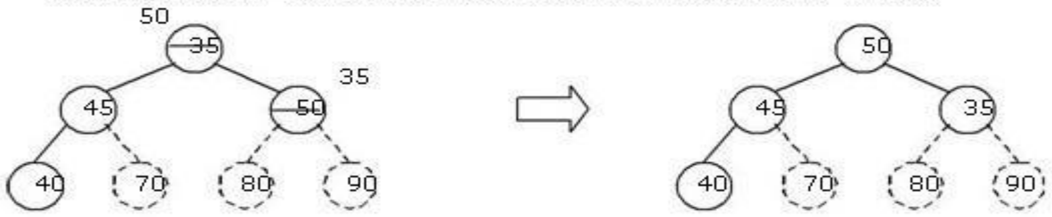
1. Exchange root 90 with the last element 35 of the array and re-heapify



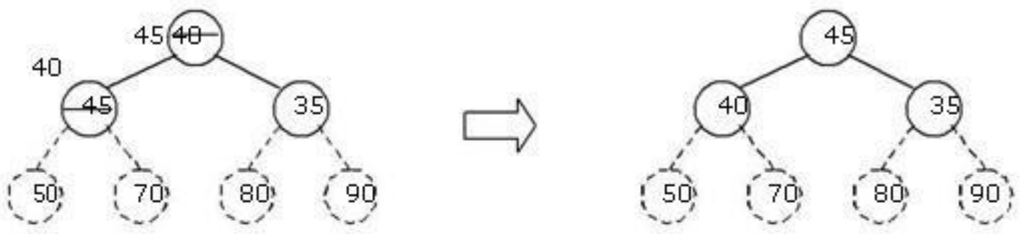
2. Exchange root 80 with the last element 50 of the array and re-heapify



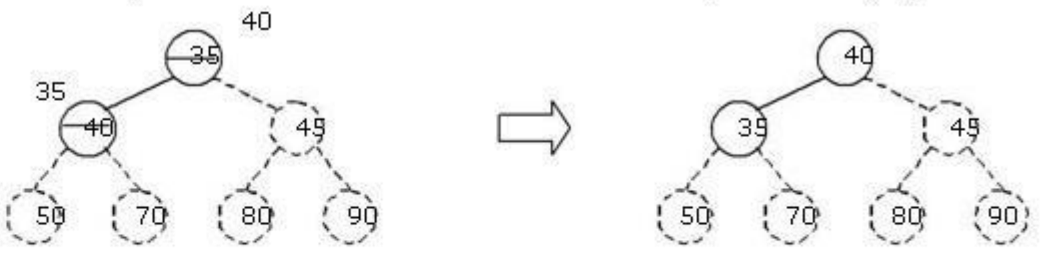
3. Exchange root 70 with the last element 35 of the array and re-heapify



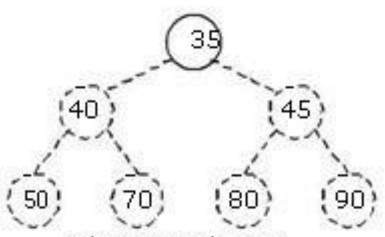
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

Program for Heap Sort:

```
  void adjust(int i, int n, int a[])
  {
            int j, item;
            j= 2 * i;
            item = a[i];
            while(j <= n)
            {
                    if((j < n) && (a[j] < a[j+1]))
                           j++;
                     if(item >= a[j])     break;
                     else
                     {
                               a[j/2] = a[j];
                               j = 2*j;
                     }
             }
            a[j/2] = item;
  }
  void heapify(int n, int a[])
  {
            int i;
            for(i = n/2; i > 0; i--)
                     adjust(i, n, a);
  }
  void heapsort(int n,int a[])
  {
            int temp, i;
            heapify(n, a);
            for(i = n; i > 0; i--)
            {
                     temp = a[i];
                     a[i] = a[1];
                     a[1] = temp;
                     adjust(1, i - 1, a);
            }
}
  void main()
  {
            int i, n, a[20];
            printf("\n How many element you want: ");
            scanf("%d", &n);
            printf("Enter %d elements: ", n);
             for (i=1; i<=n; i++)
                     scanf("%d", &a[i]);
             heapsort(n, a);
            printf("\n The sorted elements are: \n");
            for (i=1; i<=n; i++)
                     printf("%5d", a[i]);
}
```

Merge Sort:

Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 which is already sorted. The two lists of size 1 are then merged.

Merge Sort Procedure:

This is a divide and conquer algorithm. This works as follows :

1. Divide the input which we have to sort into two parts in the middle. Call it the left part and rightpart.

2. Sort each of them separately. Note that here sort does not mean to sort it using some other method. We use the same function recursively.

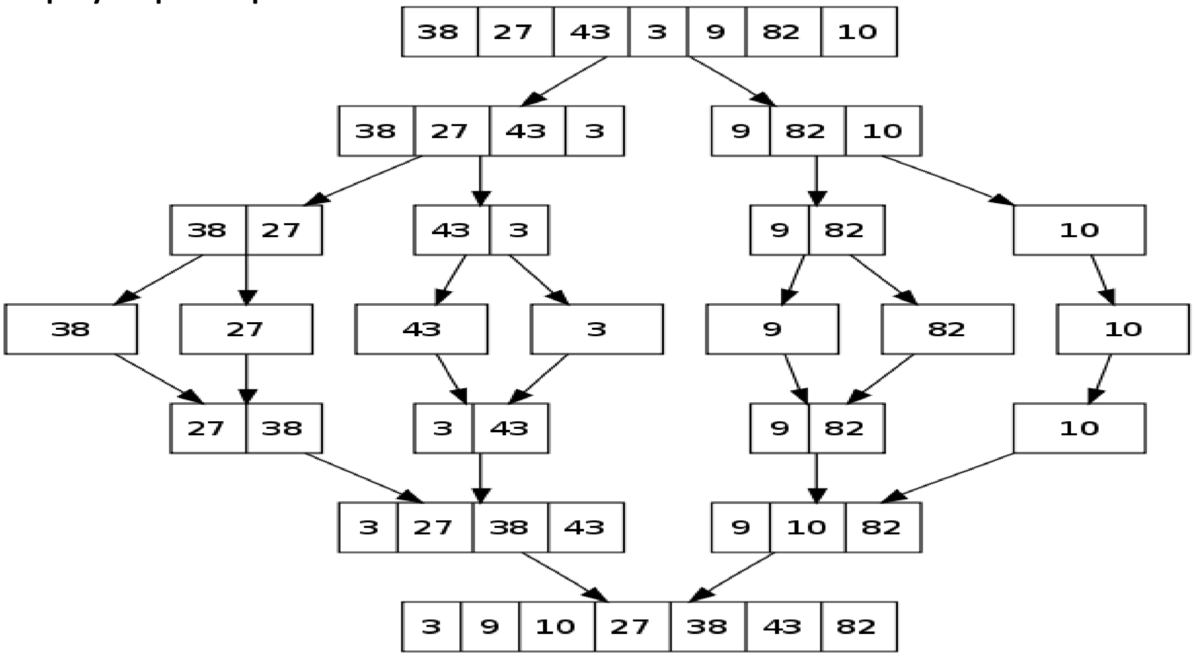
3. Then merge the two sorted parts.

Input the total number of elements that are there in an array (number_of_elements). Input the array (array[number_of_elements]). Then call the function MergeSort() to sort the input array. MergeSort() function sorts the array in the range [left,right] i.e. from index left to index right inclusive. Merge() function merges the two sorted parts. Sorted parts will be from [left, mid] and [mid+1, right]. After merging output the sorted array.

MergeSort() function:

It takes the array, left-most and right-most index of the array to be sorted as arguments. Middle index (mid) of the array is calculated as (left + right)/2. Check if (left<right) cause we have to sort only when left<right because when left=right it is anyhow sorted. Sort the left part by calling MergeSort() function again over the left part MergeSort(array,left,mid) and the right part by recursive call of MergeSort function as MergeSort(array,mid + 1, right). Lastly merge the two arrays using the Merge function.

Step-by-step example:



Program:

```
#include <stdio.h>
#define max 10
int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];
void merging(int low, int mid, int high)
{
   int l1, l2, i;
   for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++)
  {
      if(a[l1] <= a[l2])
               b[i] = a[l1++];
      else
               b[i] = a[l2++];
   }
   while(l1 <= mid)
         b[i++] = a[l1++];
   while(l2 <= high)
      b[i++] = a[l2++];
   for(i = low; i <= high; i++)
      a[i] = b[i];
}
void sort(int low, int high)
 {
      int mid;
      if(low < high)
     {
               mid = (low + high) / 2;
                sort(low, mid);
                sort(mid+1, high);
                merging(low, mid, high);
      }
     else
      {
               return;
      }
}
int main()
{
   int i;
   printf("List before sorting\n");
   for(i = 0; i <= max; i++)
          printf("%d ", a[i]);
   sort(0, max);
   printf("\nList after sorting\n");
    for(i = 0; i <= max; i++)
           printf("%d ", a[i]);
}
```

External Sorting

All the *internal sorting* algorithms require that the input fit into main memory. There are, however, applications where the input is much too large to fit into memory. For those external sorting algorithms, which are designed to handle very large inputs.

Why We Need New Algorithms

Most of the internal sorting algorithms take advantage of the fact that memory is directly addressable. Shell sort compares elements *a*[*i*] and *a*[*i* - *hk*] in one time unit. Heap sort compares elements *a*[*i*] and *a*[*i* * 2] in one time unit. Quicksort, with median-of-three partitioning, requires comparing *a*[*left*], *a*[*center*], and *a*[*right*] in a constant number of time units. If the input is on a tape, then all these operations lose their efficiency, since elements on a tape can only be accessed sequentially. Even if the data is on a disk, there is still a practical loss of efficiency because of the delay required to spin the disk and move the disk head.
The time it takes to sort the input is certain to be insignificant compared to the time to read the input, even though sorting is an $O(n \log n)$ operation and reading the input is only $O(n)$.

Model for External Sorting

The wide variety of mass storage devices makes external sorting much more device dependent than internal sorting. The algorithms that we will consider work on tapes, which are probably the most restrictive storage medium. Since access to an element on tape is done by winding the tape to the correct location, tapes can be efficiently accessed only in sequential order (in either direction).

We will assume that we have at least three tape drives to perform the sorting. We need two drives to do an efficient sort; the third drive simplifies matters. If only one tape drive is present, then we are in trouble: any algorithm will require $O(n^2)$ tape accesses.

The Simple Algorithm

The basic external sorting algorithm uses the *merge* routine from merge sort. Suppose we have four tapes, *Ta*1, *Ta*2, *Tb*1, *Tb*2, which are two input and two output tapes. Depending on the point in the algorithm, the *a* and *b* tapes are either input tapes or output tapes.

Suppose the data is initially on *Ta*1. Suppose further that the internal memory can hold (and sort) *m* records at a time. A natural first step is to read *m* records at a time from the input tape, sort the records internally, and then write the sorted records alternately to *Tb*1 and *Tb*2. We will call each set of sorted records a *run*. When this is done, we rewind all the tapes.
Phase1: Divide the file into blocks of size m and sorting of blocks, store on output tapes.
Phase2: Merging of Runs

Consider the example:

T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

If *m* = 3, then after the runs are constructed, the tapes will contain the data indicated in the following figure.

T_{a1}									
T_{a2}									
T_{b1}	11	81	94	17	28	99	15
T_{b2}	12	35	96	41	58	75	

Now *Tb*1 and *Tb*2 contain a group of runs. We take the first run from each tape and merge them, writing the result, which is a run twice as long, onto *Ta*1. Then we take the next run from each tape, merge these, and write the result to *Ta*2. We continue this process, alternating between *Ta*1 and *Ta*2, until either *Tb*1 or *Tb*2 is empty. At this point either both are empty or there is one run left. In the latter case, we copy this run to the appropriate tape. We rewind all four tapes, and repeat the same steps, this time using the *a* tapes as input and the *b* tapes as output. This will give runs of 4*m*. We continue the process until we get one run of length *n*.

This algorithm will require log(*n/m*) passes, plus the initial run-constructing pass. For instance, if we have 10 million records of 128 bytes each, and four megabytes of internal memory, then the first pass will create 320 runs. We would then need nine more passes to complete the sort. Our example requires log 13/3 = 3 more passes, which are shown in the following figure.

T_{a1}	11	12	35	81	94	96	15
T_{a2}	17	28	41	58	75	99	
T_{b1}							
T_{b2}							

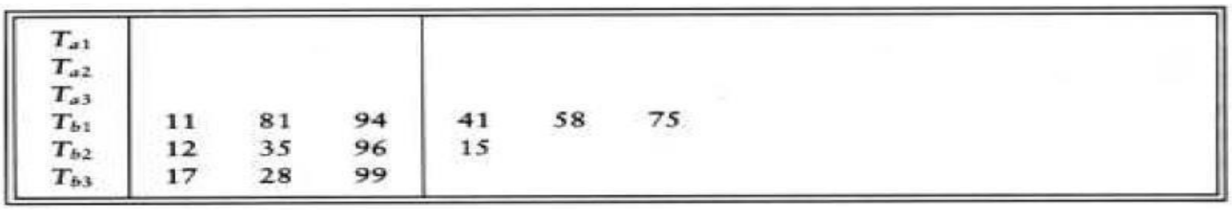
T_{a1}												
T_{a2}												
T_{b1}	11	12	17	28	35	51	58	75	81	94	96	99
T_{b2}	15											

T_{a1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{a2}													
T_{b1}													
T_{b2}													

Multiway Merge

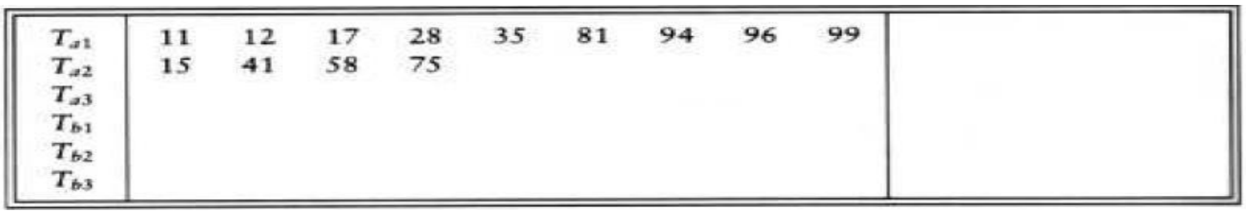
If we have extra tapes, then we can expect to reduce the number of passes required to sort our input. We do this by extending the basic (two-way) merge to a *k*-way merge.

Merging two runs is done by winding each input tape to the beginning of each run. Then the smaller element is found, placed on an output tape, and the appropriate input tape is advanced. If there are *k* input tapes, this strategy works the same way, the only difference being that it is slightly more complicated to find the smallest of the *k* elements. We can find the smallest of these elements by using a priority queue. To obtain the next element to write on the output tape, we perform a *delete_min* operation. The appropriate input tape is advanced, and if the run on the input tape is not yet completed, we *insert* the new element into the priority queue. Using the same example as before, we distribute the input onto the three tapes.

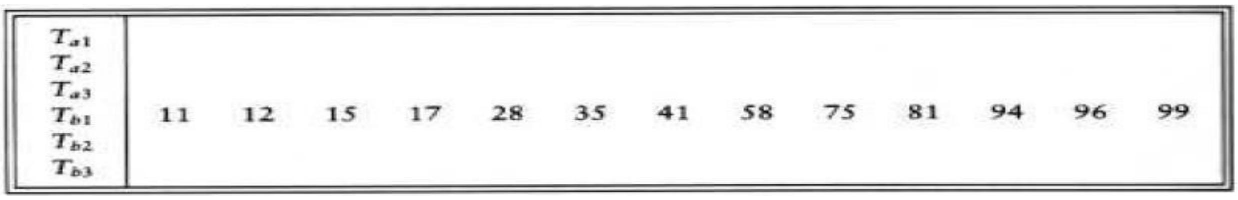


T_{a1}						
T_{a2}						
T_{a3}						
T_{b1}	11	81	94	41	58	75
T_{b2}	12	35	96	15		
T_{b3}	17	28	99			

We then need two more passes of three-way merging to complete the sort.



T_{a1}	11	12	17	28	35	81	94	96	99
T_{a2}	15	41	58	75					
T_{a3}									
T_{b1}									
T_{b2}									
T_{b3}									



T_{a1}													
T_{a2}													
T_{a3}													
T_{b1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{b2}													
T_{b3}													

After the initial run construction phase, the number of passes required using *k*-way merging is log*k*(*n/m*) , because the runs get *k* times as large in each pass. For the example above, the formula is verified, since log3 13/3 = 2. If we have 10 tapes, then *k* = 5, and our large example from the previous section would require log5 320 = 4 passes.