AS-2342

M.C.A.1st Semester, Examination-2013
Paper – fourth
Subject-Data structure with algorithm

Time: Three Hours] **[Maximum Marks: 60**

Note: attempt all questions. Q. no. 1 is compulsory. Answer any four from the remaining.

Section A

Attempts all the questions. All carry equal marks (10×2=20)

1.I In a binary tree with 20 nodes, how many null branches it will contain?
Ans. 21

1.II The minimum spanning tree of a graph give the shortest distance between any 2 specified nodes.
A. True. B. False. C. Depend on the nodes. D. None of these.
Ans. B. False

1.III The sort which inserts each elements A(K) into proper position in the previously sorted sub array A(1), ..., A(K–1)
(A) Insertion sort **(B)** Radix sort **(C)** Merge sort **(D)** Bubble sort
Ans. A. Insertion sort

1.IV Level of any node of a tree is
(A) Height of its left subtree minus height of its right subtree.
(B) Height of its right subtree minus height of its left subtree.
(C) Its distance from the root.
(D) None of these.
Ans. C. Its distance from the root.

1.V The result of evaluating the following postfix expression is …..
5, 7, 9, *, +, 4, 9, 3, /, +, -
Ans. 61

1.VI A full binary tree with 'n' non-leaf nodes contains……..total nodes (leaf+non leaf)
Ans. 2n+1

1.VII In _________ tree the difference between the height of the left sub tree and height of right sub tree, for each node, is not more than one
Ans. AVL

1.VIII The number of comparisons required to sort 5 numbers in ascending order using bubble sort is.
(A) 7 **(B)** 6 **(C)** 10 **(D)** 5
Ans. C. 10

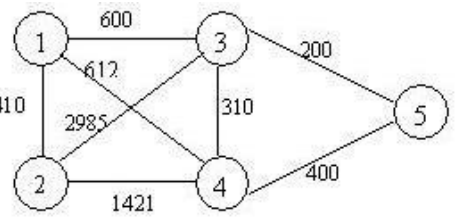
1.IX In a binary tree, the number of terminal or leaf nodes is 10. The number of nodes with two children is….
Ans. 9

1.X The complexity of Binary search algorithm is….
Ans. O(Log n)

Section B

Attempt any four questions out of seven. All carry equal marks (4 ×10=40)

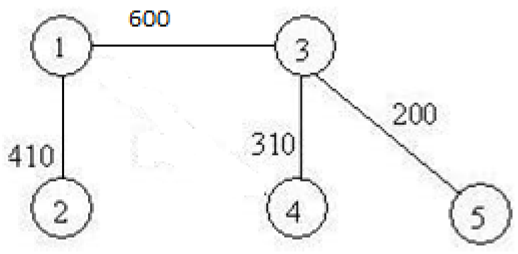
2. **Convert the given graph with weighted edges to minimal spanning tree.**



Ans. To find the minimum spanning tree(MST) there are two method

i. By kruskal algrithm and
ii. By prim's algorithm.

Here we can apply any method as there is no restriction. If we apply either of the algorithm step by step, the MST corresponding to above graph will be given below. The sum of weights of all edges is 1520



MST corresponding the given graph.

3. **How does selection sort work and write** the algorithm **with example**?

Ans. SELECTION SORT

Selection sort algorithm finds the smallest element of the array and interchanges it with the element in the first position of the array. Then it finds the second smallest element from the remaining elements in the array and places it in the second position of the array and so on. Let A be a linear array of '*n*' numbers, A [1], A [2], A [3],...... A [*n*].
Step 1: Find the smallest element in the array of *n* numbers A[1], A[2], ... A[*n*]. Let
LOC is the location of the smallest number in the array. Then interchange A[LOC] and A[1] by swap = A[LOC]; A[LOC] = A[1]; A[1] = Swap.
Step 2: Find the second smallest number in the sub list of *n* – 1 elements A [2] A [3] ...... A [*n* – 1] (first element is already sorted). Now we concentrate on the rest of the elements in the array. Again A [LOC] is the smallest element in the remaining array and LOC the corresponding location then interchange A [LOC] and A [2].Now A [1] and A [2] is sorted, since A [1] less than or equal to A [2].
Step 3: Repeat the process by reducing one element each from the array
Step n – 1: Find the *n* – 1 smallest number in the sub array of 2 elements (*i.e.*, A(*n*–1), A (*n*)). Consider A [LOC] is the smallest element and LOC is its corresponding position. Then interchange A [LOC] and A(*n* – 1). Now the array A [1], A [2], A [3], A [4],………..A [*n*] will be a sorted array.
Following figure is generated during the iterations of the algorithm to sort 5 numbers 42, 33, 23, 74, 44 :

ALGORITHM

Let A be a linear array of *n* numbers A [1], A [2], A [3], ……… A [k], A [k+1], …….. A [n]. *Swap* be a temporary variable for swapping (or interchanging) the position of the numbers. *Min* is the variable to store smallest number and *Loc* is the location of the smallest element.

```
1. Input n numbers of an array A
2. Initialize i = 0 and repeat through step5 if (i < n – 1)
(a) min = a[i]
(b) loc = i
3. Initialize j = i + 1 and repeat through step 4 if (j < n – 1)
4. if (a[j] < min)
(a) min = a[j]
(b) loc = j
5. if (loc ! = i)
(a) swap = a[i]
(b) a[i] = a[loc]
(c) a[loc] = swap
6. display "the sorted numbers of array A"
```

TIME COMPLEXITY

Time complexity of a selection sort is calculated in terms of the number of comparisons *f* (*n*). In the first pass it makes *n* – 1 comparisons; the second pass makes *n* – 2 comparisons and so on. The outer *for loop* iterates for (*n* - 1) times. But the inner loop iterates for *n**(*n* – 1) times to complete the sorting.

$f(n) = (n - 1) + (n - 2) + ... + 2 + 1$

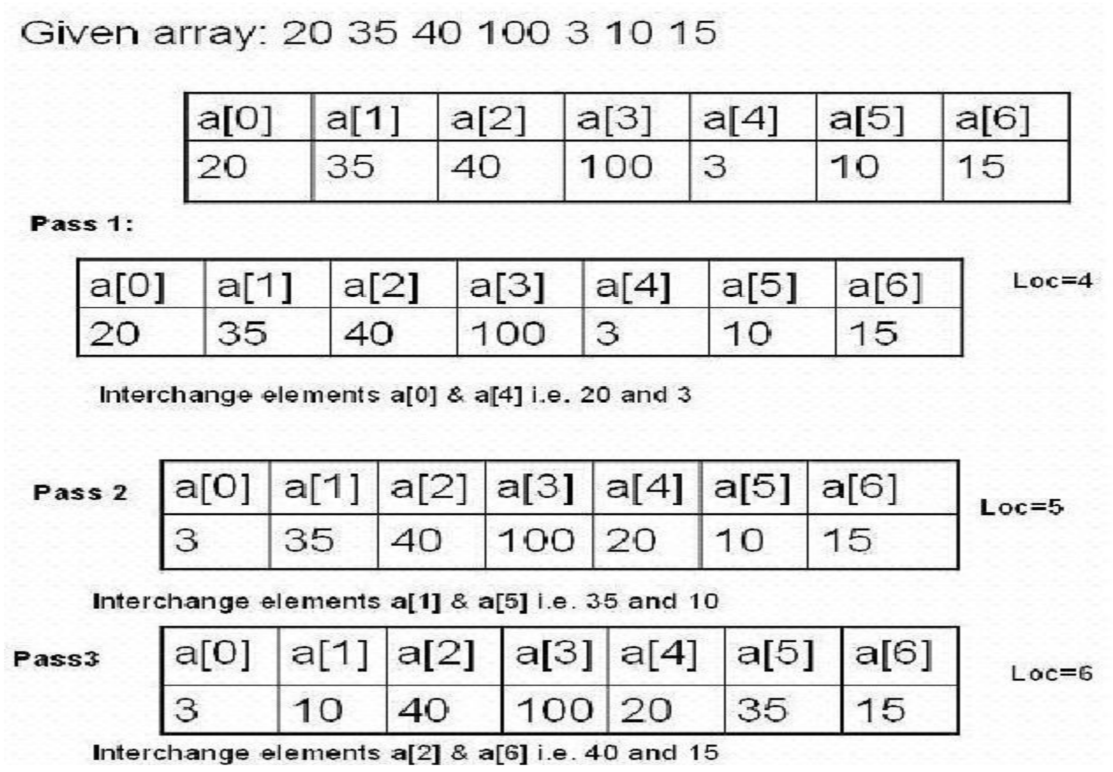
$= (n(n - 1))/2$

$= O(n_2)$.

Best case	Worst case	Average case
$n - 1 = O(n)$	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n)$

Example.

Sort: 20,35,40,100,3,10,15



Given array: 20 35 40 100 3 10 15

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
20	35	40	100	3	10	15

Pass 1:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
20	35	40	100	3	10	15

Loc=4

Interchange elements a[0] & a[4] i.e. 20 and 3

Pass 2

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	35	40	100	20	10	15

Loc=5

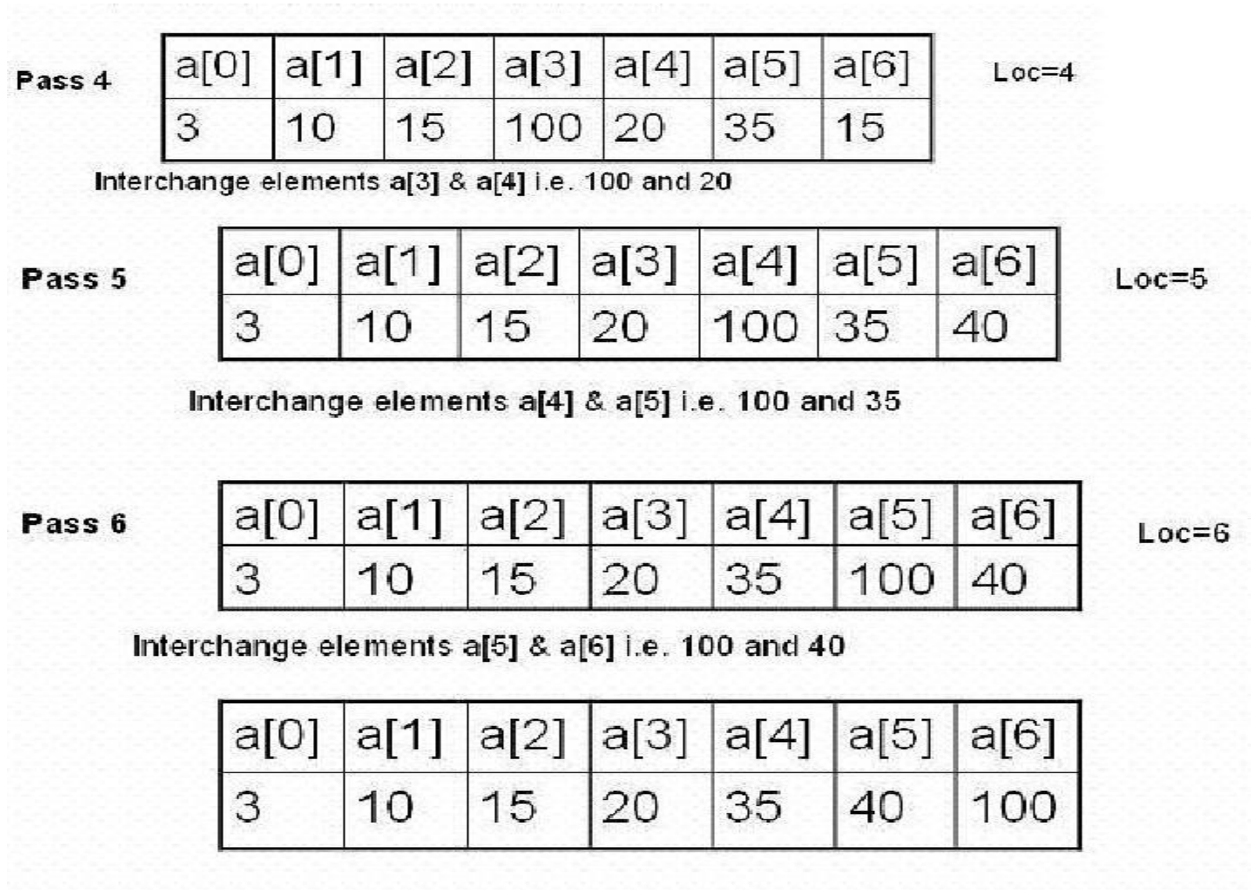
Interchange elements a[1] & a[5] i.e. 35 and 10

Pass3

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	40	100	20	35	15

Loc=6

Interchange elements a[2] & a[6] i.e. 40 and 15


Pass 4

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	15	100	20	35	15

Loc=4

Interchange elements a[3] & a[4] i.e. 100 and 20

Pass 5

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	15	20	100	35	40

Loc=5

Interchange elements a[4] & a[5] i.e. 100 and 35

Pass 6

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	15	20	35	100	40

Loc=6

Interchange elements a[5] & a[6] i.e. 100 and 40

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
3	10	15	20	35	40	100

4. Describe dijkstra's algorithm with example

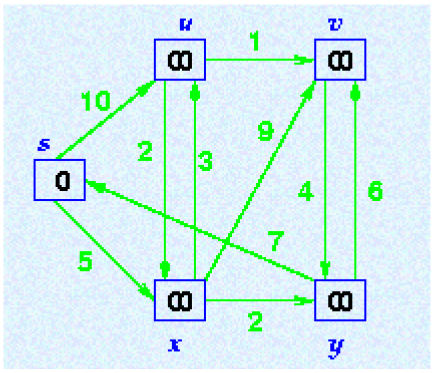
Ans. A path from a source vertex a to b is said to be shortest path if there is no other path from a to b with lower weights. There are many instances, to find the shortest path for traveling from one place to another. That is to find which route can reach as quick as possible or a route for which the traveling cost in minimum. Dijkstra's Algorithm is used find shortest path.

Dijkstra's algorithm solves the single-source shortest-path problem when all edges have non-negative weights. It is a greedy algorithm and similar to Prim's algorithm. Algorithm starts at the source vertex, s, it grows a tree, T, that ultimately spans all vertices reachable from S. Vertices are added to T in order of distance i.e., first S, then the vertex closest to S, then the next closest, and so on. Following implementation assumes that graph G is represented by adjacency lists.

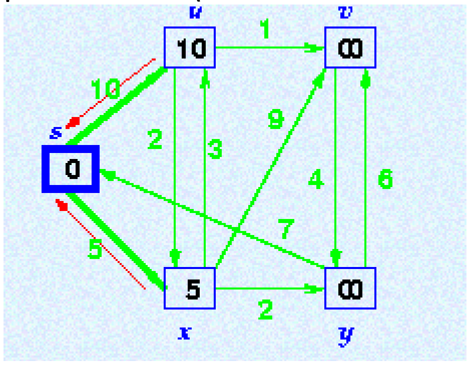
```
DIJKSTRA (G, w, s)
1. INITIALIZE SINGLE-SOURCE (G, s)
2. S ← { } // S will ultimately contains vertices of final shortest-path weights from s
3. Initialize priority queue Q i.e., Q ← V[G]
4. while priority queue Q is not empty do
5. u ← EXTRACT_MIN(Q) // Pull out new vertex
6. S ← S È {u} // Perform relaxation for each vertex v adjacent to u
7. for each vertex v in Adj[u] do
8. Relax (u, v, w)
```

Example.

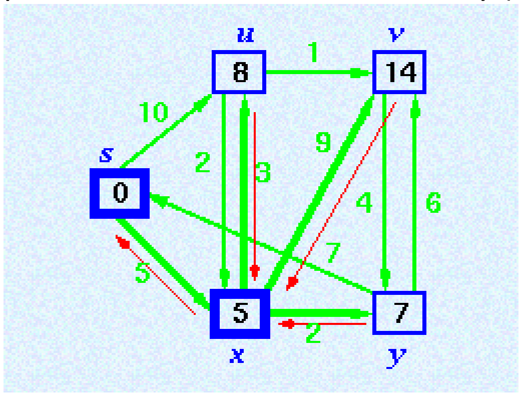
Step1. Given initial graph G=(V, E). All nodes nodes have infinite cost except the source node, s, which has 0 cost.



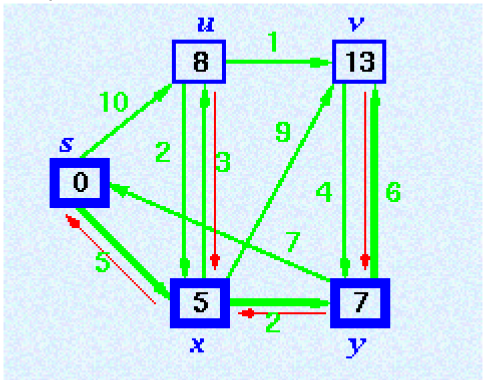
Step 2. First we choose the node, which is closest to the source node, s. We initialize d[s] to 0. Add it to S. Relax all nodes adjacent to source, s. Update predecessor (see red arrow in diagram below) for all nodes updated.



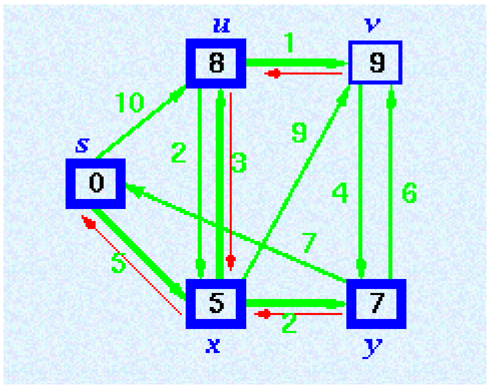
Step 3. Choose the closest node, x. Relax all nodes adjacent to node x. Update predecessors for nodes u, v and y (again notice red arrows in diagram below).



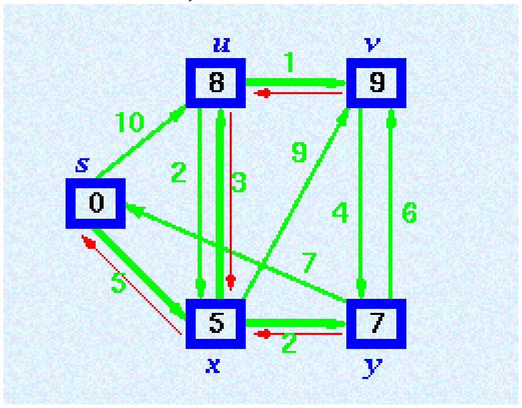
Step 4. Now, node y is the closest node, so add it to S. Relax node v and adjust its predecessor (red arrows remember!).



Step 5. Now we have node u that is closest. Choose this node and adjust its neighbor node v.

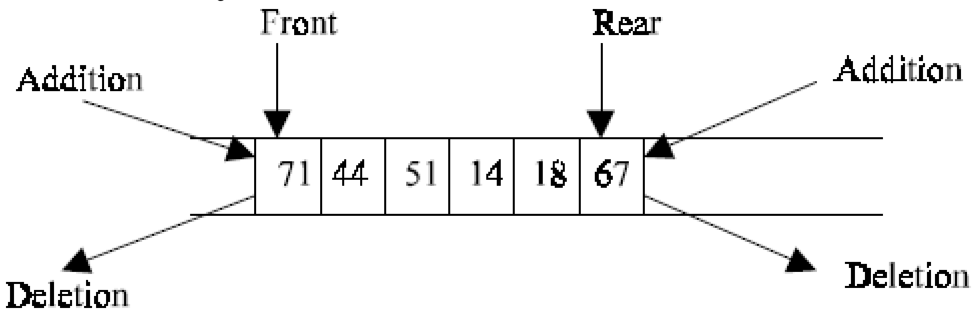


Step 6. Finally, add node v. The predecessor list now defines the shortest path from each node to the source node, s.



5. **Explain dequeue with example**.

Ans. A deque is a homogeneous list in which elements can be added or inserted (called push operation) and deleted or removed from both the ends (which is called pop operation). ie; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.



There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are

1. Input restricted deque
2. Output restricted deque

An input restricted deque is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists.

An output-restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

The possible operation performed on deque is

1. Add an element at the rear end
2. Add an element at the front end
3. Delete an element from the front end
4. Delete an element from the rear end

Only 1st, 3rd and 4th operations are performed by input-restricted deque and 1st, 2nd and 3rd operations are performed by output-restricted deque.

ALGORITHMS FOR INSERTING AN ELEMENT

Let Q be the array of MAX elements. *front* (or left) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. Let DATA be the element to be inserted. Before inserting any element to the queue *left* and *right* pointer will point to the – 1.

INSERT AN ELEMENT AT THE RIGHT SIDE or REAR END or RIGHT MOST OF THE DE-QUEUE

```
1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right + 1))
   (a) Display “Queue Overflow”
   (b) Exit
3. If (left == –1)
   (a) left = 0
   (b) right = 0
4. Else
   (a) if (right == MAX –1)
       (i) left = 0
   (b) else
       (i) right = right+1
5. Q[right] = DATA
6. Exit
```

Also write any suitable example

INSERT AN ELEMENT AT THE LEFT SIDE or FRONT END or LEFT MOST OF THE DE-QUEUE

```
1. Input the DATA to be inserted
2. If ((left == 0 && right == MAX–1) || (left == right+1))
   (a) Display “Queue Overflow”
   (b) Exit
```

```
3. If (left == – 1)
   (a) Left = 0
   (b) Right = 0
4. Else
   (a) if (left == 0)
       (i) left = MAX – 1
   (b) else
       (i) left = left – 1
5. Q[left] = DATA
6. Exit
```

Also write any suitable example

ALGORITHMS FOR DELETING AN ELEMENT

Let Q be the array of MAX elements. *front* (or *left*) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. DATA will contain the element just deleted.

DELETE AN ELEMENT FROM THE RIGHT SIDE or REAR END or RIGHT MOST OF THE DE-QUEUE

```
1. If (left == – 1)
   (a) Display “Queue Underflow”
   (b) Exit
2. DATA = Q [right]
3. If (left == right)
   (a) left = – 1
   (b) right = – 1
```

```
4. Else
    (a) if(right == 0)
        (i) right = MAX-1
    (b) else
        (i) right = right-1
5. Exit
```
Also write any suitable example

DELETE AN ELEMENT FROM THE LEFT SIDE or FRONT END or LEFT MOST OF THE DE-QUEUE

```
1. If (left == – 1)
    (a) Display “Queue Underflow”
    (b) Exit
2. DATA = Q [left]
3. If(left == right)
    (a) left = – 1
    (b) right = – 1
4. Else
    (a) if (left == MAX-1)
        (i) left = 0
    (b) Else
        (i) left = left +1
5. Exit
```
Also write any suitable example

6. Describe priority queue with example.

Ans.

The abstract data type known as a **priority queue** allows us to insert an item with a specified priority (given by a number) and to delete an item having the *highest* priority. For example, if we are processing jobs with specified priorities and there is a single processor available, as the jobs arrive we can insert them into a priority queue. When the processor becomes available, we can delete a job from the priority queue—which, by definition, has the highest priority—and process it.

In the discussion that follows, we list only the priority of an item. In practice, a data item has other attributes besides its priority (e.g., an identification number, time started, time stopped).

Priority Queue is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules.

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were inserted to the queue.

For example, Consider a manager who is in a process of checking and approving files in a first come first serve basis. In between, if any urgent file (with a high priority) comes, he will process the urgent file next and continue with the other low urgent files.

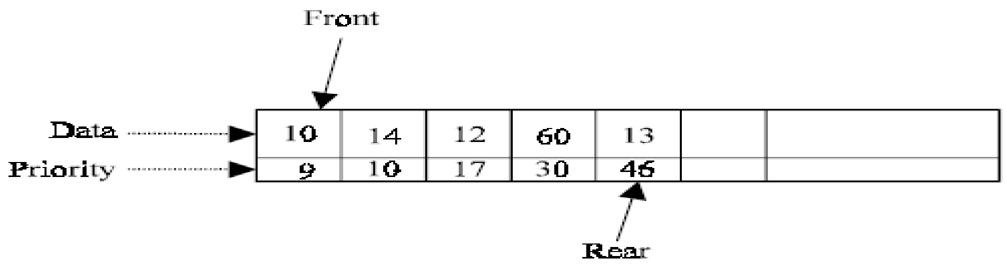


Fig. 1 Priority queue using array

Above fig. 1 gives a pictorial representation of priority queue using arrays after adding 5 elements (10,14,12,60,13) with its corresponding priorities (9,10,17,30,46). Here the priorities of the data(s) are in ascending order. Always we may not be pushing the data in an ascending order. From the mixed priority list it is difficult to find the highest priority element if the priority queue is implemented using arrays. Moreover, the implementation of priority queue using array will yield n comparisons (in liner search), so the time complexity is O(n), which is much higher than the other queue (ie; other queues takes only O(1)) for inserting an element. So it is always better to implement the

priority queue using linked list - where a node can be inserted at anywhere in the list - which is discussed in this section.
A node in the priority queue will contain DATA, PRIORITY and NEXT field. DATA field will store the actual information; PRIORITY field will store its corresponding priority of the DATA and NEXT will store the address of the next node. Fig. 2 shows the linked list representation of the node when a DATA (*i.e.*, 12) and PRIORITY (*i.e.*, 17) is inserted in a priority queue

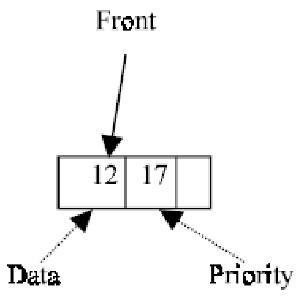


Fig. 2 Linked list representation of priority queue.

When an element is inserted into the priority queue, it will check the priority of the element with the element(s) present in the linked list to find the suitable position to insert. The node will be inserted in such a way that the data in the priority field(s) is in ascending order. We do not use rear pointer when it is implemented using linked list, because the new nodes are not always inserted at the rear end. Following figures will illustrate the push and pop operation of priority queue using linked list.

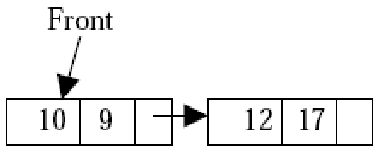


Fig. 3 push(DATA =10, PRIORITY = 9)

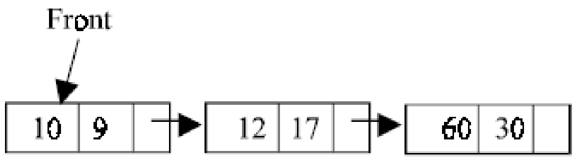


Fig. 4. Push (DATA = 60, PRIORITY = 30)

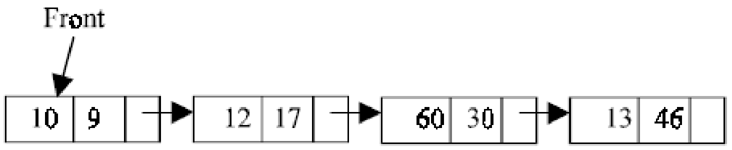


Fig. 5 Push (DATA = 13, PRIORITY = 46)

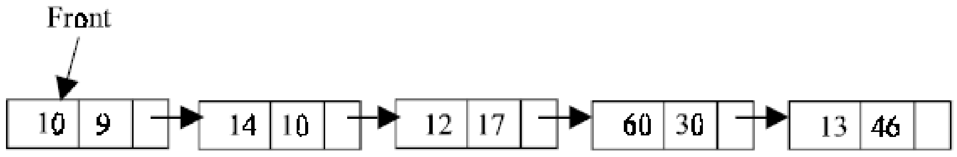


Fig. 6 Push(DATA = 14, PRIORITY = 10)

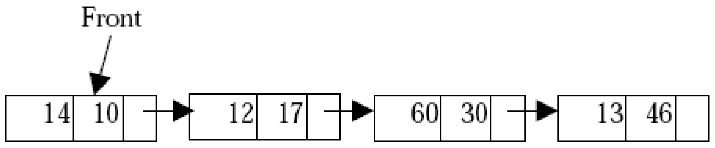


Fig. 7. x = Pop() (i.e., 10)

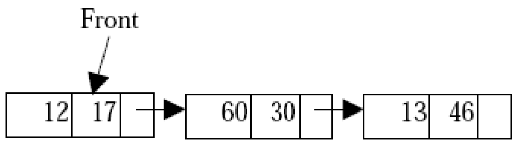


Fig. 8. *x* = Pop() (*i.e.*, 14)

7. **Write the algorithm of postfix evaluation with example**.

Ans. Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

1. Scan P from left to right and repeat Steps 2 and 3 for each element of P until the P is finished.
2. If an operand is encountered, put it on STACK.
3. If an operator is encountered, then:

(*a*) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.

(*b*) Evaluate B operator A.

(*c*) Place the result on to the STACK.

4. Result equal to the top element on STACK.
5. Exit.

Example

Evaluate: 2 3 * 4 + (for infix 2 * 3 + 4)

Steps to evaluate 2 3 * 4 +:

1. Read 2, push 2
2. Read 3, push 3
3. Read *, pop 3, pop 2, mult. 2 * 3, push 6
4. Read 4, push 4
5. Read +, pop 4, pop 6, add 6 + 4, push 10

So expression is over and in this way the result is 10

8. **Explain the queue and circular queue with example and give the difference between them**.

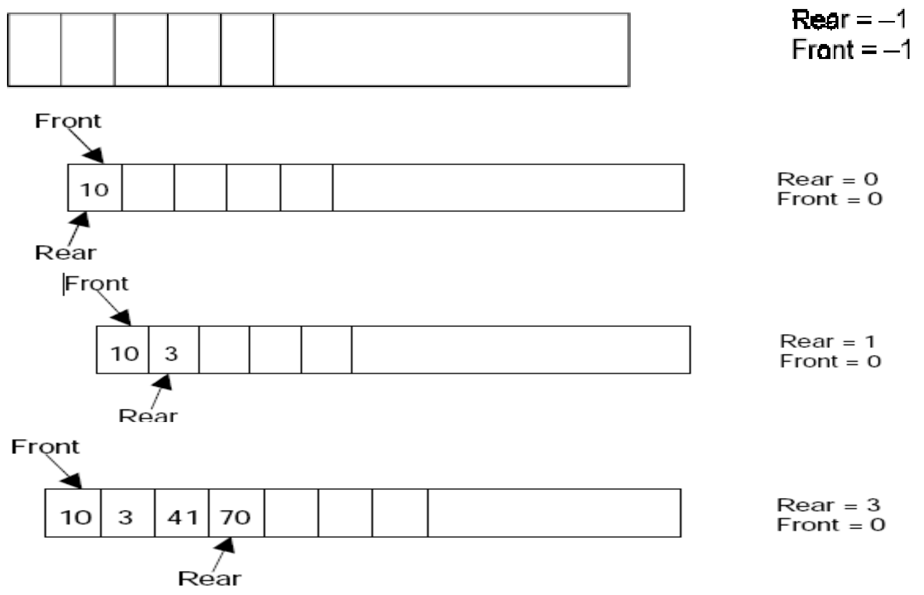
Queue

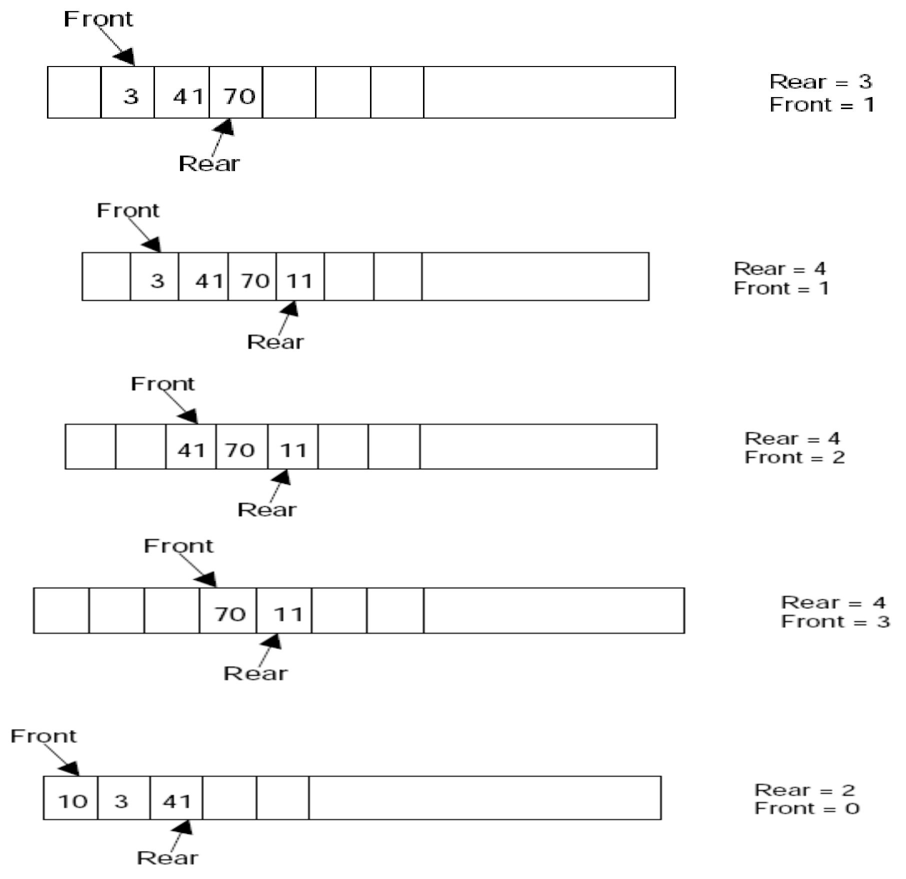
A queue is logically a *first in first out (FIFO or first come first serve)* linear data structure. The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service centre. It is a homogeneous collection of elements in which new elements are added at one end called *rear*, and the existing elements are deleted from other end called *front*.

The basic operations that can be performed on queue are

1. Insert (or add) an element to the queue (push)
2. Delete (or remove) an element from a queue (pop)

Push operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. Pop operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Total number of elements present in the queue is front-rear+1, when implemented using arrays. Following figure will illustrate the basic operations on queue.





Queue can be implemented in two ways:

1. Using arrays (static)
2. Using pointers (dynamic)

Implementation of queue using pointers will be discussed in chapter 5. Let us discuss underflow and overflow conditions when a queue is implemented using arrays.

If we try to pop (or delete or remove) an element from queue when it is empty, underflow occurs. It is not possible to delete (or take out) any element when there is no element in the queue.

Suppose maximum size of the queue (when it is implemented using arrays) is 50. If we try to push (or insert or add) an element to queue, overflow occurs. When queue is full it is naturally not possible to insert any more elements

ALGORITHM FOR QUEUE OPERATIONS

Let Q be the array of some specified size say SIZE

INSERTING AN ELEMENT INTO THE QUEUE

```
1. Initialize front=0 rear = –1
2. Input the value to be inserted and assign to variable "data"
3. If (rear >= SIZE)
(a) Display "Queue overflow"
(b) Exit
4. Else
(a) Rear = rear +1
5. Q[rear] = data
6. Exit
```

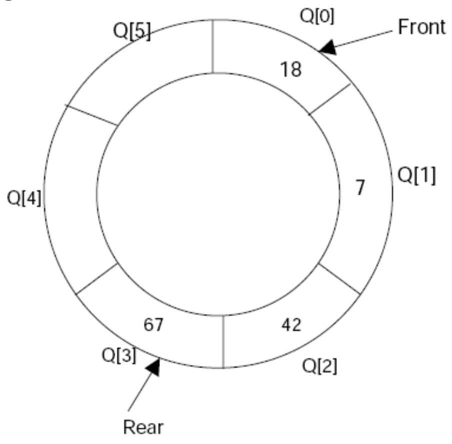
DELETING AN ELEMENT FROM QUEUE

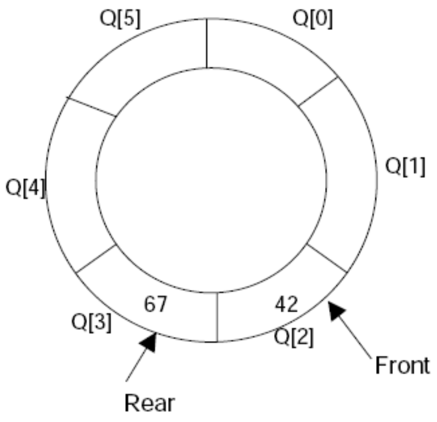
```
1. If (rear< front)
(a) Front = 0, rear = –1
(b) Display "The queue is empty"
(c) Exit
```

```
2. Else
(a) Data = Q[front]
3. Front = front +1
4. Exit
```

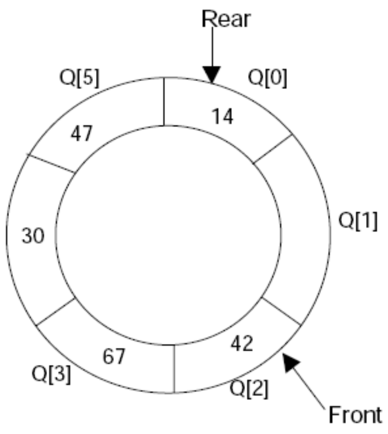
Circular queue

Ans. In circular queues the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.
Suppose Q is a queue array of 6 elements. Push and pop operation can be performed on circular. The following figures will illustrate the same.





After inserting an element at last location Q[5], the next element will be inserted at the very first location (*i.e.*, Q[0]) that is circular queue is one in which the first element comes just after the last element.



At any time the position of the element to be inserted will be calculated by the relation Rear = (Rear + 1) % SIZE
After deleting an element from circular queue the position of the front end is calculated by the relation Front= (Front + 1) % SIZE
After locating the position of the new element to be inserted, *rear*, compare it with *front*. If (rear = front), the queue is full and cannot be inserted anymore

ALGORITHMS

Let Q be the array of some specified size say SIZE. FRONT and REAR are two pointers where the elements are deleted and inserted at two ends of the circular queue. DATA is the element to be inserted.

Inserting an element to circular Queue

```
1. Initialize FRONT = – 1; REAR = 1
2. REAR = (REAR + 1) % SIZE
3. If (FRONT is equal to REAR)
    (a) Display “Queue is full”
    (b) Exit
4. Else
    (a) Input the value to be inserted and assign to variable “DATA”
5. If (FRONT is equal to – 1)
    (a) FRONT = 0
    (b) REAR = 0
6. Q[REAR] = DATA
7. Repeat steps 2 to 5 if we want to insert more elements
8. Exit
```

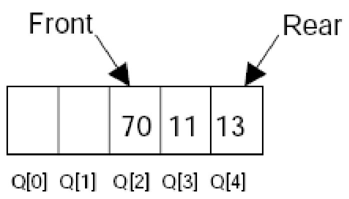
Deleting an element from a circular queue

```
1. If (FRONT is equal to – 1)
    (a) Display "Queue is empty"
    (b) Exit
2. Else
    (a) DATA = Q[FRONT]
3. If (REAR is equal to FRONT)
    (a) FRONT = –1
    (b) REAR = –1
4. Else
    (a) FRONT = (FRONT +1) % SIZE
5. Repeat the steps 1, 2 and 3 if we want to delete more elements
6. Exit
```

Diffrence

In Linear queue the last node address field contains NULL value but in circular queue the last node address points to first node.

Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elements popped.



Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed. Because in a queue, elements are always inserted at the *rear* end and hence *rear* points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty. This limitation can be overcome if we use circular queue.