SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Queue

sushmakadge@somaiya.edu
swatimali@somaiya.edu

Somaiya TRUST

Queue

- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- Luggage kept on conveyor belts. The bag which was placed first will be the first to come out at the other end.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

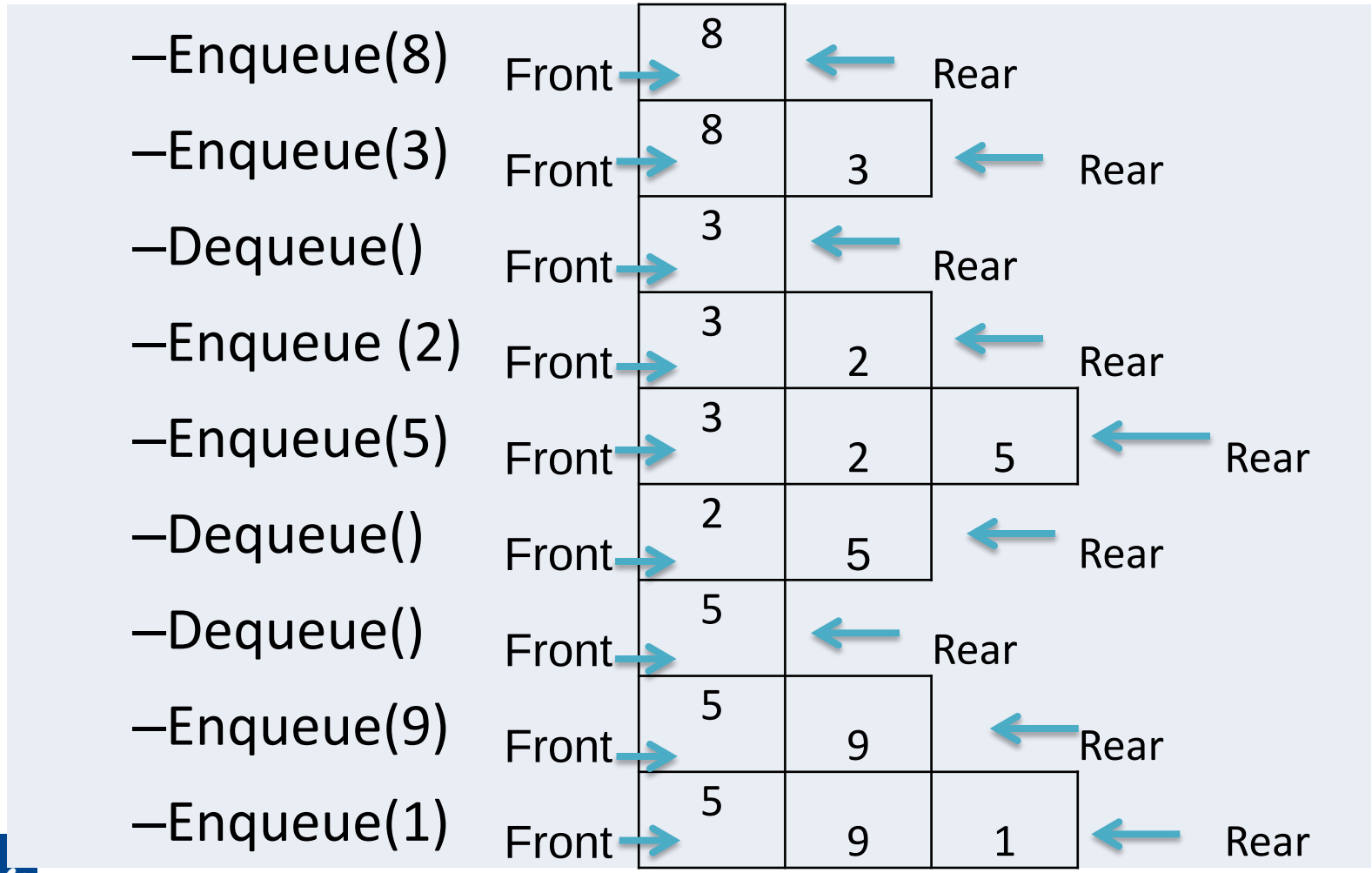
Queue

- First In First Out
- Elements are added at one end called REAR and removed only from the other end called FRONT
- Gives access only to two elements- one at the front and one at the rear end

A Queue

- Definition:
  - An ordered collection of homogenous data items
  - Where elements are added at rear and removed from the front end
- Operations:
  - Create an empty queue
  - check if it is empty and/or full
  - Enqueue: add an element at the rear
  - Dequeue: remove the element in front
  - Destroy : remove all the elements one by one and destroy the data structure

Exercise: Queue



Types of queues

- Simple queue- additions at rear and deletions from front
- Circular queue- last node is connected to first node, deletions at front end while insertions are done at rear end
- Doubly ended queue- deletions and insertions can be done at both the ends, has two pairs of fronts and rears, both
- Priority queue- every element has predefined priority
  - Max priority : element with max priority is removed first
  - min priority: element with min priority is removed first

Simple Queue

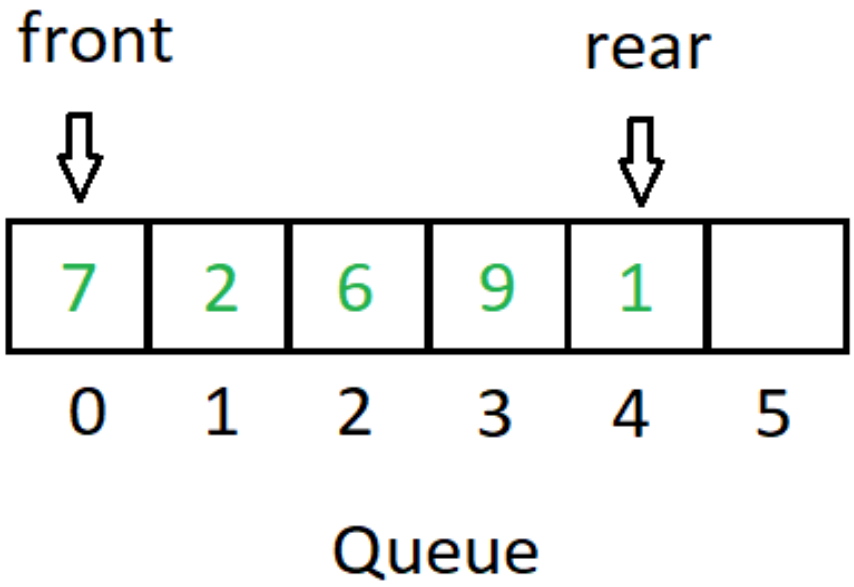


Image courtesy: GeeksforGeeks.org

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Circular Queue

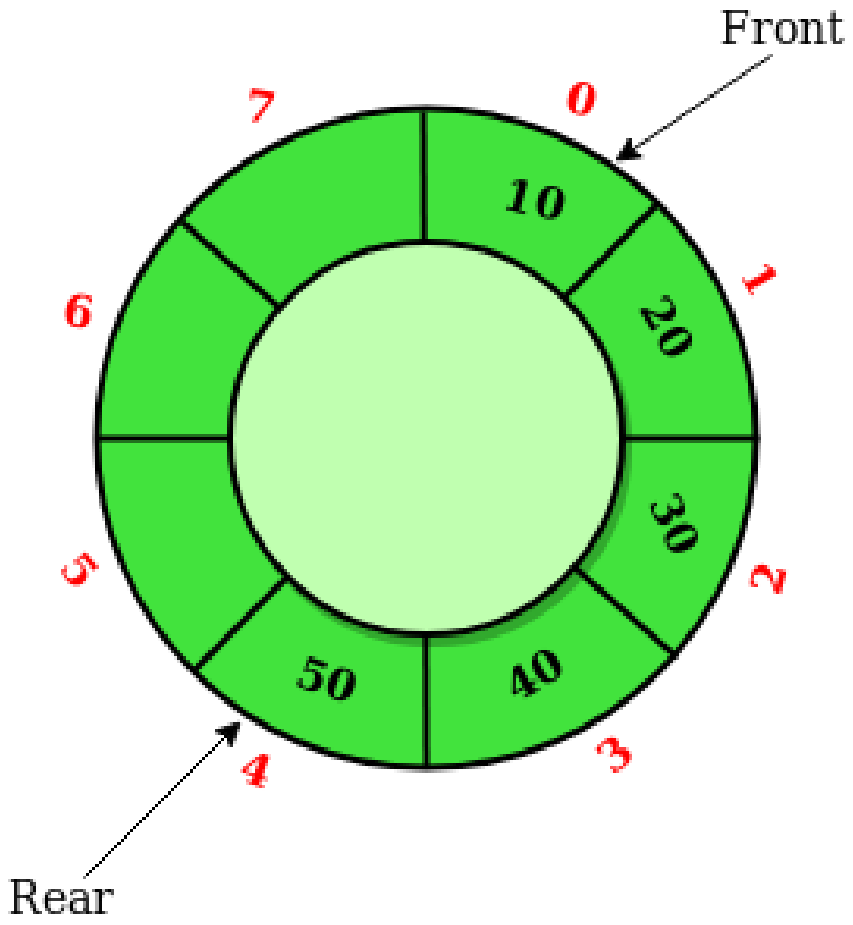


Image courtesy: GeeksforGeeks.org

Somaiya TRUST

Doubly ended Queue

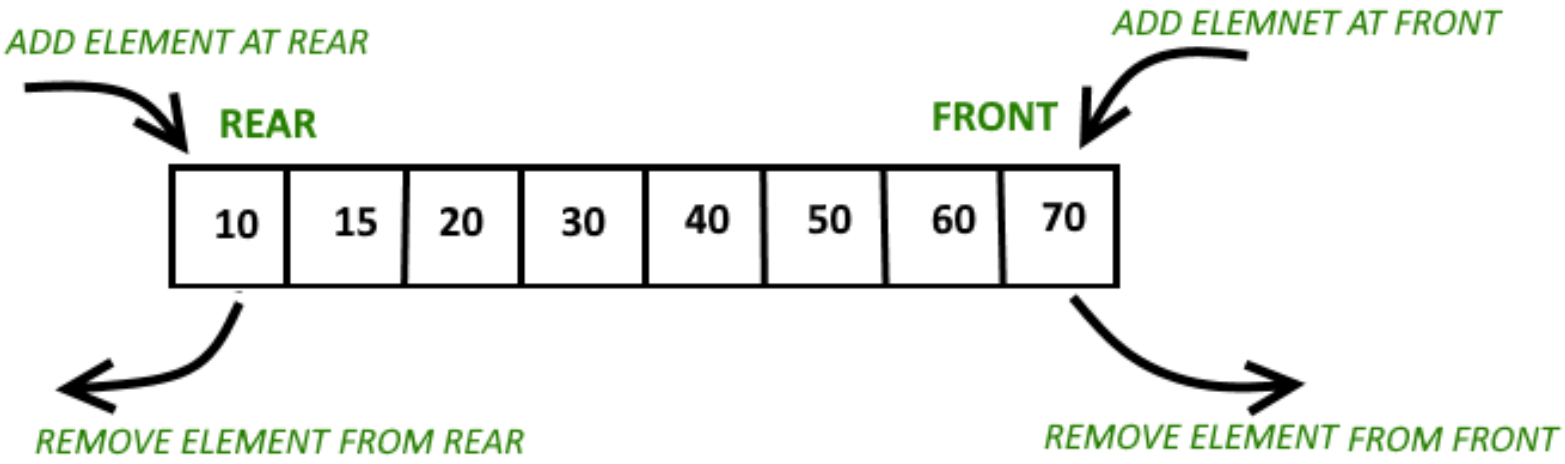


Image courtesy: GeeksforGeeks.org

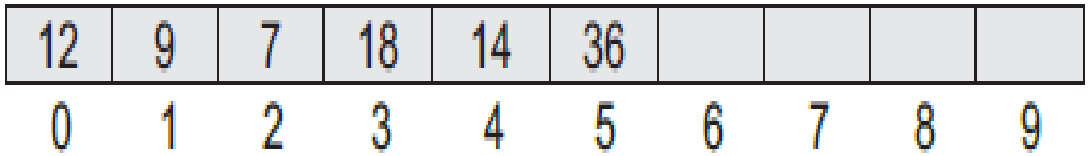
Priority Queue

Index	Front				Rear			
Data	10	5	3	98	12			
Priority	5	4	3	2	1			

Max Priority queue

ARRAY REPRESENTATION OF QUEUEs

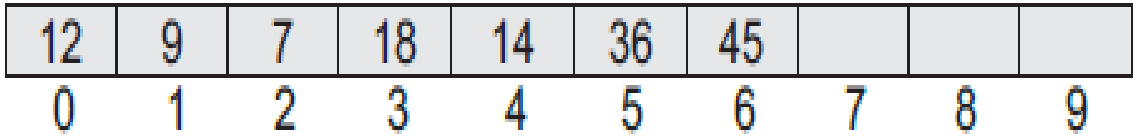
- Queues can be easily represented using linear arrays.
- point to the position from where deletions and insertions can be done, resp.
- The array representation of a queue is



- FRONT = 0 and REAR = 5.

ARRAY REPRESENTATION

- Add another element with value 45
- REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.
- The queue after addition would be as



- Here, FRONT = 0 and REAR = 6.
- Every time a new element has to be added, we repeat the same procedure.

ARRAY REPRESENTATION

- Delete an element from the queue,
- The value of FRONT will be incremented.
- Deletions are done fromthis end of the queue.
- The queue after deletion will be

	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

- Here, FRONT = 1 and REAR = 6.

Implementing Queues: Simple queue with Array

Queue indices:	Front Rear									
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:	8									

- Initally, front=rear=-1 (Empty queue)
- Enqueue(8), Enqueue(3), Dequeue(), Enqueue (2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1)

Implementing Queues: Simple queue with Array

Queue indices:	Front	Rear								
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:	8	3								

- Enqueue(8), Enqueue(3), Dequeue(), Enqueue (2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1)

Implementing Queues: Simple queue with Array

Queue indices:		Front Rear								
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:		3								

- Enqueue(8), Enqueue(3), Dequeue(), Enqueue (2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1)

Implementing Queues: Simple queue with Array

Queue indices:		Front	Rear							
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:		3	2							

- Enqueue(8), Enqueue(3), Dequeue(), Enqueue(2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1)

Implementing Queues: Simple queue with Array

Queue indices:		Front		Rear						
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:		3	2	5						

- Enqueue(8), Enqueue(3), Dequeue(), Enqueue(2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1)

Implementing Queues: Simple queue with Array

Queue indices:			Front	Rear						
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:			2	5						

- Enqueue(8), Enqueue(3), Dequeue(), Enqueue(2), Enqueue(5), Dequeue(), Pop(), Enqueue(9), Enqueue(1)

Implementing Queues: Simple queue with Array

Queue indices:				Front Rear						
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:				5						

- Enqueue(8), Enqueue(3), Dequeue(), Enqueue(2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1)

Implementing Queues: Simple queue with Array

Queue indices:				Front	Rear					
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:				5	9					

- Enqueue(8), Enqueue(3), Dequeue(), Enqueue(2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1)

Implementing Queues: Simple queue with Array

Queue indices:				Front		Rear				
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:				5	9	1				

- Enqueue(8), Enqueue(3), Dequeue(), Enqueue(2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1),

Implementing Queues: Simple queue with Array

Queue indices:						Front Rear				
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:						1				

- Enqueue(8), Enqueue(3), Dequeue(), Enqueue(2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1), Dequeue(), Dequeue(), Dequeue()

Implementing Queues: Simple queue with Array

Queue indices:										
Array Index:	0	1	2	3	4	5	6	7	8	9
Data:										

- **Front=-1, Rear=-1**
- Enqueue(8), Enqueue(3), Dequeue(), Enqueue(2), Enqueue(5), Dequeue(), Dequeue(), Enqueue(9), Enqueue(1), Dequeue(), Dequeue(), Dequeue()

Algorithm to insert an element

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Queue : Array Implementation.

- #define MAX 10
- int queue[MAX];
- int front = -1, rear = -1;
- if(rear == MAX-1)
- printf("\n OVERFLOW");
- else if(front == -1 && rear == -1)
- front = rear = 0;
- else
- rear++;
- queue[rear] = num;

Algorithm to Delete an element

```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

Queue : Array Implementation.

```
 if(front == -1 || front>rear)
{
printf("\n UNDERFLOW");
return -1;
}
else
{
val = queue[front];
front++;
}
```

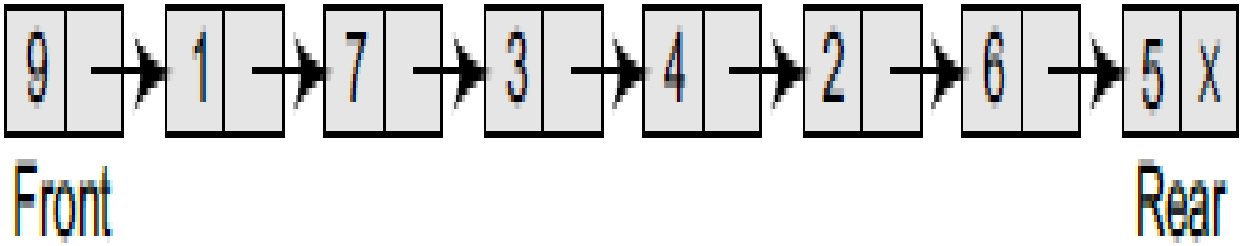
LINKED REPRESENTATION OF QUEUEs

- The queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation.
- But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.
- The storage requirement of linked representation of a queue with n elements is O(n) and the typical time requirement for operations is O(1).

LINKED REPRESENTATION OF QUEUEs

- In a linked queue, element has two parts, one that stores the data and another that stores the address of the next element.
- The START pointer of the linked list is used as FRONT. Also use another pointer called REAR, which will store the address of the last element in the queue.
- All insertions will be done at the rear end and all the deletions will be done at the front end.
- If FRONT = REAR = NULL, then the queue is empty.

LINKED REPRESENTATION OF QUEUEs



Operations on Linked Queues

- Insert *Operation*
- Delete *Operation*
- Peek *Operation*

Algorithm to insert an element in a linked queue

```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

Implementation: linked queue

```
struct queue *insert(struct queue *q,int val){
struct node *ptr;
ptr = (struct node*)malloc(sizeof(struct node));
ptr -> data = val;
if(q -> front == NULL){
q -> front = ptr;
q -> rear = ptr;
q -> front -> next = q -> rear -> next = NULL;}
else{
q -> rear -> next = ptr;
q -> rear = ptr;
q -> rear -> next = NULL;}
return q;
}
```

Algorithm to delete an element in a linked queue

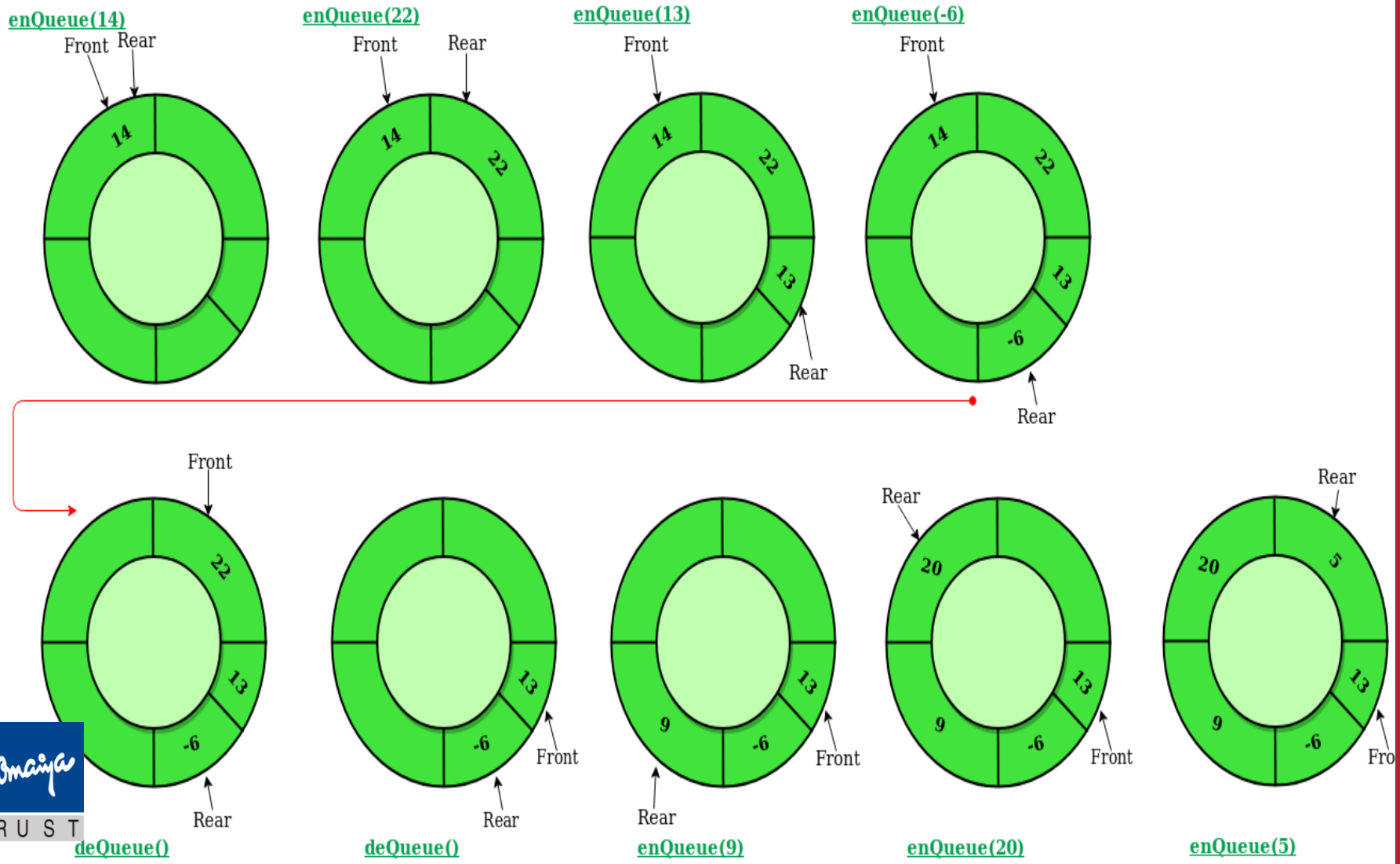
```
Step 1: IF FRONT = NULL
            Write "Underflow"
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Implementing Circular Queue

Somaiya TRUST

Implementing Queues: Simple queue with Array



Algorithm to insert an element in a circular queue

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW"
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Algorithm to delete an element in a circular queue

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

Circular Queue: Array Implementation

1. Enqueue
   - Insertion in full queue
   - Insertion in initially empty queue
   - General case
2. Dequeue

   -deletion from empty queue

   -deleting the last remained value in the queue

   - General case

Circular Queue: Array Implementation

1. Algorithm QueueType CreateCQueue()

//This Algorithm returns an empty Queue

{ front =-1;

Rear=-1

Return queue;

}

Circular Queue: Array Implementation

2. Algorithm QueueType CEnqueue(QueueType CQueue, ElementType Element)

// This algorithm accepts a QueueType Queue and ElementType Element as input and adds 'Element' at the rear of 'Queue'. Front and rear are the integer indices those point to the front and rear elements in the queue. Array CQueue[0:Size-1] is an array that stores queue elements.

```
{
        if NotFull(CQueue)= True
        {if (rear == SIZE – 1 && front != 0)
                rear=0;
        else rear= rear+1;
        CQueue[rear]= Element // add the element at rear
        if (front==-1) then front =0; // insertion of first element
        }
        Else "Error Message"
}
```

Circular Queue: Array Implementation

3. Algorithm ElementType Dequeue(QueueType CQueue)

// This algorithm accepts a queue as input and returns 'Element' at the front of 'queue'. Temp is a temporary variable used to hold the value being deleted. Array CQueue[0:Size-1] is an array that stores queue elements.

```
{  if NotEmpty(CQueue)= True
     {temp= CQueue[front];
          if (front==rear) then front=rear=-1; //deletion of last element
          else if (front==size-1) then front=0;//front was pointing last
                                                location
          Else front++; // general case
          return(temp)
     }
Else  print "Error Message"
}
```

Circular Queue: Array Implementation

4. Abstract DestroyQueue(QueueType CQueue)

//This algorithm returns all the elements from Queue in FIFO order and destroys the data structure

```
{ if NotEmpty(CQueue) = true
     while(NotEmpty(CQueue))
          print Dequeue(CQueue)
   else print “Error Message”
}
```

Circular Queue: Array Implementation

5. Abstract Boolean NotFull(QueueType CQueue)

// This algorithm returns true if the Queue is not full, false otherwise. Array CQueue[0:Size-1] is an array that stores queue elements. Rear and front are the indices those point to first and last element in circular queue, respectively.

```
{ if  ((rear == SIZE-1 && front == 0) || (rear == front-1))
            return False
  else
            return True
}
```

6. Abstract Boolean NotEmpty(QueueType CQueue)

```
// This algorithm returns true if the Queue is not empty, false otherwise.
{ if (front != -1)
            return True
  else
            return False
```

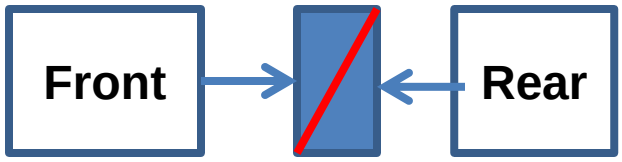
Implementing Circular Queue: Linked List

```
Struct NodeType{
                ElementType Element;
                NodeType Next;
        }
```

1. Algorithm QueueType CreateQueue()

//This Algorithm creates and returns an empty Queue, pointed by two pointers- front and rear

```
{ createNode(front);
createNode(rear);
Front=rear=NULL;
}
```



Implementing Queue: Linked List

2. QueueType Enqueue(QueueType CQueue, NodeType NewNode)

// This Algorithm adds a NewNode at the rear of 'queue'. rear is a pointer that points to the last node in the queue

```
{
        If(front==rear==NULL)
                Front=rear=newnode // insertion of first element
                rear->next=newnode //circular queue definition
        else //general case
                temp=front;
                while(temp!=rear) {
                        temp=temp->next;
                        temp->next = newnode;
                        newnode->next = rear->next;
                        rear=newnode;
                }//while
}//enqueue
```

Enqueue another algorithm

2. QueueType Enqueue(QueueType CQueue, NodeType NewNode)

// This Algorithm adds a NewNode at the rear of 'queue'. rear is a pointer that points to the last node in the queue

```
{
        If(front==rear==NULL)
                Front=rear=newnode // insertion of first element
                rear->next=newnode //circular queue definition
        else //general case
                rear->next= newnode;
                rear=newnode;
                newnode->next=front;

}//enqueue
```

Implementing Queue: Linked List

3. Algorithm ElementType DeQueue(QueueType CQueue)

//This algorithm returns value of ElementType stored at the front of queue. Temp is a temporary node used in the dequeuer process.

```
{ if (front==rear==NULL)
            Print “Underflow”
            exit;
 Else if (front==rear)
            { temp= front;
              front=rear=NULL;
              return(temp->data);
            }
Else {
            temp=front;
            front=front->next;
            rear->next= front;
            return(temp->data);
            }
}//Dequeue
```

Implementing Stacks: Linked List

4. Abstract DestroyQueue(QueueType CQueue)

//This algorithm returns values stored in data structure and free the memory used in data structure implementation.

```
{ if front==NULL
        Print “Underflow”
        exit;
 Else {   createNode(Temp);
        while(NotEmpty(CQueue))
        {
        return(Dequeue(CQueue));
        }
        }//else
}
```

Implementing Queue: Linked List

6. Abstract DisplayQueue(QueueType Queue)

//This algorithm Prints all the Elements stored in stack. Temp purpose?

```
{ if front==NULL
        Print “Error Message”
  Else {
Student Assignment


}
```

Priority Queues

- A priority queue is a data structure in which each element is assigned a priority.
- The priority of the element will be used to determine the order in which the elements will be processed.
- The general rules of processing the elements of a priority queue are

  - An element with higher priority is processed before an element with a lower priority.

  -Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

Priority Queues

- A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first.
- The priority of the element can be set based on various factors.
- Widely used in operating systems to execute the highest priority process first.
- The priority of the process may be set based on the CPU time it requires to get executed completely.
- For eg, 3 processes, 1st process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns,
- Then the second process will have the highest priority and will thus be the first to be executed.
- However, CPU time is not the only factor that determines the priority, rather it is just one among several factors.

Priority Queues

- Another factor is the importance of one process over another. In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

Implementation Priority Queues

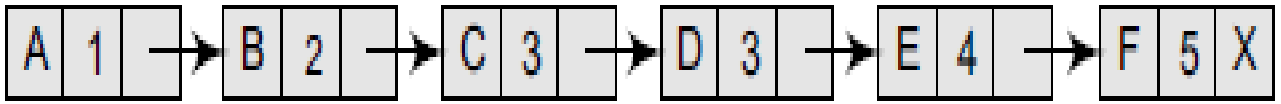
- Two ways to implement a priority queue.
- We can either use a sorted list to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority
- or we can use an unsorted list so that insertions are always done at the end of the list. Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed.
- While a sorted list takes O(n) time to insert an element in the list, it takes only O(1) time to delete an element.
- On the contrary, an unsorted list will take O(1) time to insert an element and O(n) time to delete an element from the list.

Representation of a Priority Queue

- In the computer memory, a priority queue can be represented using arrays or linked lists.
- When a priority queue is implemented using a linked list, then every node of the list will have three parts:
- (a) the information or data part,
- (b) the priority number of the element, and
- (c) the address of the next element.
- If we are using a sorted linked list, then the element with the higher priority will precede the element with the lower priority.

Representation of a Priority Queue

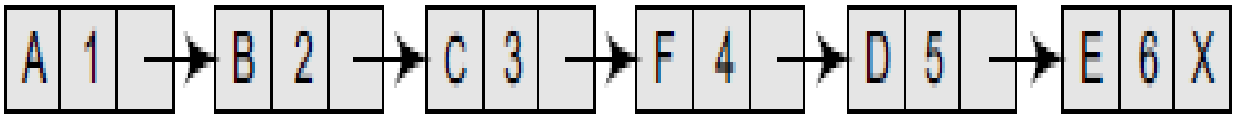
Lower priority number means higher priority. For example, if there are two elements A and B, where A has a priority number 1 and B has a priority number 5, then A will be processed before B as it has higher priority than B.

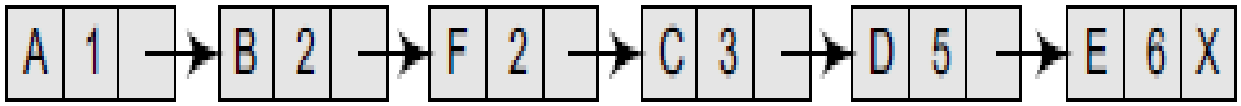


Representation of a Priority Queue

- ***Insertion***

- Priority queue after insertion of a new node





Representation of a Priority Queue

- ***Deletion***

  Deletion is a very simple process in this case. The first node of the list will be deleted and the data of that node will be processed first.

Array Representation of a Priority Queue

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained.
- Each of these queues will be implemented using circular arrays or circular queues.
- Every individual queue will have its own FRONT and REAR pointers.
- We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space.

Array Representation of a Priority Queue

FRONT	REAR
3	3
1	3
4	5
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3				E	F
4	I			G	H

Array Representation of a Priority Queue

- ***Insertion*** To insert a new element with priority K in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element.
- For example, if we have to insert an element R with priority number 3, then the priority queue will be given

FRONT	REAR
3	3
1	3
4	1
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3	R			E	F
4	I			G	H

Array Representation of a Priority Queue

- ***Deletion*** To delete an element, we find the first nonempty queue and then process the front element of the first non-empty queue. In our priority queue, the first non-empty queue is the one with priority number 1 and the front element is A, so A will be deleted and processed first.

- In technical terms, find the element with the smallest K, such that FRONT[K] != NULL.

Priority Queue: Array Implementation

1. Enqueue
   - Insertion in full queue
   - Insertion in initially empty queue
   - General case
2. Dequeue

   -deletion from empty queue

   -deleting the last remained value in the queue

   - General case

Priority Queue: Array Implementation

```
Struct PriQueue{       int data;
                       int priority
                };
Struct PriQueue PQ[MaxSize];
```

1. Algorithm QueueType CreatePQueue()

//This Algorithm returns an empty Queue

```
{ front =-1;
Rear=-1
}
```

Priority Queue: Array Implementation

2. Algorithm QueueType PEnqueue(QueueType PQueue, ElementType Element, int p)

// This algorithm accepts a QueueType Pqueue, ElementType Element and its associated priority 'p' as input and adds 'Element' at the rear of 'Queue'. Front and rear are the integer indices those point to the front and rear elements in the queue. Array PQueue[0:MaxSize-1] is an array that stores queue elements.

```
{
            if(rear==MaxSize-1) then  overflow; exit; //PQueue is full
else        if (front==rear==-1) // inserting first element
            { front=rear=0;
             PQ[0].data= element;
            PQ[0].priority = p;
            }
}else if { rear++// increment rear to accommodate new element
            PQ[rear].data=element;
             PQ[rear].priority=p;
      //find a proper place for new element as per its priority using insertion sort logic
           key=PQ[rear]
            j=rear-1;
            while(j>=0 && PQ[j].priority < key.priority)
            { PQ[j+1]=PQ[j];
              j--;
            }
            PQ[j+1]= key; //assign both data value and priority
}
```

Priority Queue: Array Implementation

3. Algorithm ElementType Dequeue(QueueType PQueue)

// This algorithm accepts a queue as input and returns 'Element' at the front of 'queue'. Temp is a temporary variable used to hold the value being deleted. Array CQueue[0:Size] is an array that stores queue elements.

```
{ if (front=-1) then underflow; exit; // deleting from empty data structure?
   if(front==rear) { // only element in PQueue
            temp=PQ[front] ;
            front=rear=-1;
            }//if
      else  { // General case
            temp=PQque[front]
            front++;
            }//else
return(temp)
```

Priority Queue: Array Implementation

4. Abstract DestroyQueue(QueueType PQueue)

//This algorithm returns all the elements from Queue in FIFO order and destroys the data structure

```
{ if NotEmpty(PQueue) = true
     while(NotEmpty(PQueue))
          print Dequeue(PQueue)
   else print “Error Message”
}
```

Priority Queue: Array Implementation

5. Abstract Boolean NotFull(QueueType PQueue)

Student assignment

6. Abstract Boolean NotEmpty(QueueType PQueue)

Student assignment

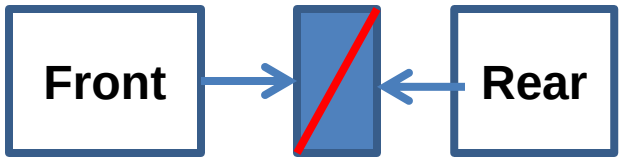
Implementing Priority: Linked List

```
Struct NodeType{
                ElementType Element;
                integer priority;
                NodeType Next;
        }
```

1. Algorithm QueueType CreateQueue()

//This Algorithm creates and returns an empty Queue, pointed by two pointers- front and rear

```
{ createNode(front);
createNode(rear);
Front=rear=NULL;
}
```



Implementing Priority Queue: Linked List

2. QueueType Enqueue(QueueType PQueue, NodeType NewNode, int p)

```
// This Algorithm adds a NewNode at the rear of 'queue'. rear is a pointer that points to the last node in the queue
{  if(rear==Null) //if inserting first element?
                front=rear=NewNode;
   else if(front.priority >  NewNode->priority) //insertion before the first node
              { NewNode->next= front;
              front= NewNode;
              }
   else   { temp = front; current=NULL;
              while(temp->priority<=NewNode->priority && temp->next!=Null)
                          current=temp; temp=temp->next;
              if(temp->priority > NewNode->Priority) //insertion in between
                          Newnode->next= temp;
                          current-> next= NewNode;
              if(temp->next==NULL) // insertion after rear
                          temp->next=NewNode;
                          rear=NewNode;
              }
}//enqueue
```

Implementing Priority Queue: Linked List

3. Algorithm ElementType DeQueue(QueueType PQueue)

//This algorithm returns value of ElementType stored at the front of queue. Temp is a temporary node used in the dequeuer process.

```
{ if (front==NULL)
        Print "Underflow"
        exit;
 Else if (front==rear) // deleting the last remaining node in the PQueue
        { temp= front;
          front=rear=NULL;
          return(temp->data);
        }
Else                    // general case
        {
        temp=front;
        front=front->next;
        return(temp->data);
        }

}//Dequeue
```

Implementing Dqueue: Linked List

4. Abstract DestroyQueue(QueueType PQueue)

//This algorithm returns values stored in data structure and free the memory used in data structure implementation.

```
{ if (front==NULL)
        Print “Underflow”
        exit;
 Else {   createNode(Temp);
        while(NotEmpty(PQueue))
        {
        return(Dequeue(PQueue));
        }
        }//else
}
```

Implementing Queue: Linked List

6. Abstract DisplayQueue(QueueType DQueue)

//This algorithm Prints all the Elements stored in stack. Temp purpose?

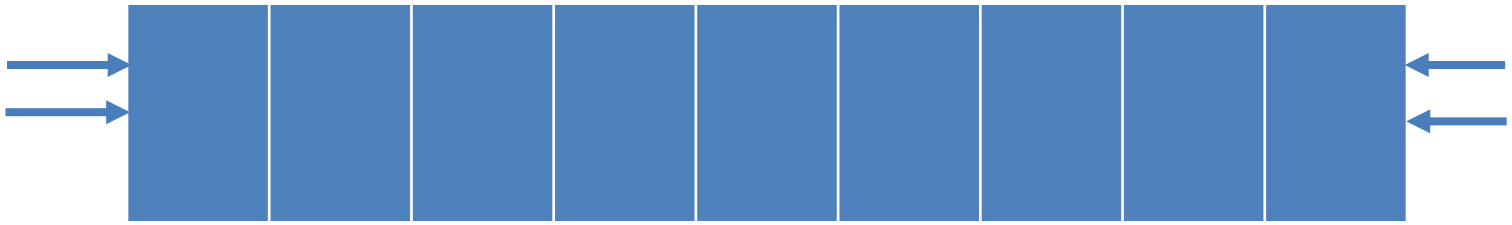
```
{ if front==NULL
        Print “Error Message”
  Else {
```

Student Assignment

```
}
```

Doubly ended queue(Deque)

- A deque is pronounced as ‘deck’ or ‘dequeue’ is a list in which the elements can be inserted or deleted at either end.



- Definition: queue has two pairs of fronts and rears on either end.

Doubly ended queue(Deque)

- Also known as a *head-tail linked list* because elements can be added to or removed from either the front (head) or the back (tail) end.
- However, no element can be added and deleted from the middle.
- In the computer's memory, a deque is implemented using either a circular array or a circular doubly linked list.
- In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque.
- The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N–1] is followed by Dequeue[0].

Doubly ended queue(Deque)

- There are two variants of a double-ended queue. They include
- *Input restricted deque*

  In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.

- *Output restricted deque*

  In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

Input restricted deque

- 1.Insert at right
- 2.Delete from left
- 3.Delete from right

Input restricted deque

```
Insert at right
void insert_right()
{
int val;
printf("\n Enter the value to be added:");
scanf("%d", &val);
if((left == 0 && right == MAX–1) || (left == right+1))
{
printf("\n OVERFLOW");
return;
}
if (left == –1) /* if queue is initially empty */
{
left = 0;
right = 0;
}
else
{
if(right == MAX–1) /*right is at last position of queue */
right = 0;
else
right = right+1;
}
deque[right] = val ;
}
```

Output restricted deque

- 1.Insert at right
- 2.Insert at left
- 3.Delete from left

DQue: Array Implementation

1. Algorithm QueueType CreateDQueue()

//This Algorithm returns an empty Queue

{ front1 =-1;

Rear1=-1;

Front2=-1;

Rear2=-1;

Return dqueue;

}

DQue: Array Implementation

2. Algorithm QueueType DEnqueue(QueueType DQueue, ElementType Element, int end)

// This algorithm accepts a QueueType DQueue and ElementType Element as input and adds 'Element' at the rear of 'Queue'. Front and rear are the integer indices those point to the front and rear elements in the queue. Array DQueue[0:Size-1] is an array that stores queue elements. The integer variable end defines where the element is to be added; 1=right end and 2=left end.

```
{
        if(end==2 && rear2==0)  then LeftEnd=Full; exit;
        if(end==1 && rear1==maxsize-1) then RightEnd=Full; exit;
        if(rear1=-1) //insertion of first element
        { front1=front2=rear1=rear2=MaxSize/2; //set indices in such a way that queue has scope to grow in both directions
        deque[rear1]=element;
        }
        else if(end==1) //insertion in right end using rear1, general case
                deque[rear1++]=element
                front2=rear2
        else if(end==2) ) //insertion in left end using rear2, general case
                deque[rear2--]=element;
                front1=rear2
```

Dque Queue: Array Implementation

3. Algorithm ElementType Dequeue(QueueType Dqueue, int end)

// This algorithm accepts a queue as input and returns 'Element' at the front of 'queue'. Temp is a temporary variable used to hold the value being deleted. Array CQueue[0:Size] is an array that stores queue elements. The integer variable end defines from where the element is to be deleted; 1=left end and 2=right end

```
{ if (front1==-1) then underflow; exit; // deleting from empty data structure?
   if(front1==front2==rear1==rear2) { // only element in deque
                temp=Deque[front1]
                front1=front2=rear1=rear2=-1
                }//if
     else if(end==1) { // deletion in left end with front1?
                temp=Deque[front1]
                front1++; rear2++;
                }//else if
     else if(end==2) { // deletion in right end with front2?
                temp=temp=Deque[front2]
                front2--; rea1--;
                } //else if
  return(temp)
}
```

Deque Queue: Array Implementation

4. Abstract DestroyQueue(QueueType DQueue)

//This algorithm returns all the elements from Queue in FIFO order and destroys the data structure

```
{ if NotEmpty(DQueue) = true
     while(NotEmpty(DQueue))
          print Dequeue(DQueue)
   else print “Error Message”
}
```

Deque Queue: Array Implementation

5. Abstract Boolean NotFull(QueueType CQueue)

Student assignment

6. Abstract Boolean NotEmpty(QueueType CQueue)

Student assignment

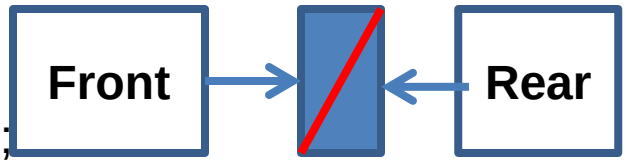
Implementing Deque: Linked List

```
Struct NodeType{
                ElementType Element;
                NodeType Next;
        }
```

1. Algorithm QueueType CreateQueue()

//This Algorithm creates and returns an empty Queue, pointed by two pointers- front and rear

```
{ createNode(front1);
createNode(rear1);
 createNode(front2);
createNode(rear2);
Front1=rear1=front2=rear2=NULL;
}
```



Implementing DQue: Linked List

2. QueueType Enqueue(QueueType CQueue, NodeType NewNode, int end)

// This Algorithm adds a NewNode at the rear of 'queue'. rear is a pointer that points to the last node in the queue

```
{  if(rear1==Null) //if inserting first element?
                front1=rear1=front2=rear2=NewNode;
   else if(end==1)
                { rear1->next= NewNode;
                front2= NewNode;
                rear1= NewNode;
                }
   else if(end==2)
                { NewNode->next= rear2;
                rear2=NewNode;
                front1=NewNode;
                }
}//enqueue
```

Implementing DQue: Linked List

3. Algorithm ElementType DeQueue(QueueType Dqueue, int end)

//This algorithm returns value of ElementType stored at the front of queue. Temp is a temporary node used in the dequeuer process.

```
{ if (front1==NULL)
        Print "Underflow"
        exit;
 Else if (front1==rear1) //last node in the data structure
        { temp= front1;
          front1=rear1=front2=rear2=NULL;
          return(temp->data);
        }
Else if (end==1)  //deleting the left end element at front1
        {
        temp=front1;
        front1=front1->next;
        rear2= front1;  or rear2= rear2->next;
        return(temp->data);
        }

Else if (end==2) //deleting the right end element at front2
        { temp=front2;
        temp2=front1;
        while(temp2->next!=front2)
                temp2=temp2->next; //While loop
        rear1= temp2;
        front2= temp2;
        rear1->next = NULL
        return(temp->data);
        }
}//Dequeue
```

Implementing Dqueue: Linked List

4. Abstract DestroyQueue(QueueType DQueue)

//This algorithm returns values stored in data structure and free the memory used in data structure implementation.

```
{ if (front1==NULL)
        Print “Underflow”
        exit;
 Else {   createNode(Temp);
        while(NotEmpty(Dqueue))
        {
        return(Dequeue(Dqueue,1));
        }
        }//else
}
```

Implementing Queue: Linked List

6. Abstract DisplayQueue(QueueType DQueue)

//This algorithm Prints all the Elements stored in stack. Temp purpose?

```
{ if front==NULL
        Print “Error Message”
  Else {
```

Student Assignment

```
}
```

APPLICATIONs OF QUEUES

- As waiting lists for a single shared resource like printer, disk, CPU.
- To transfer data asynchronously between two processes.
- As buffers on MP3 players and portable CD players, iPod playlist.
- Playlist for jukebox to add songs to the end, play from the front of the list.
- Operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job.
- If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.

Joseph's Problem

- Let us see how queues can be used for finding a solution to the Josephus problem.
- In Josephus problem, n people stand in a circle waiting to be executed.
- The counting starts at some point in the circle and proceeds in a specific direction around the circle.
- In each step, a certain number of people are skipped and the next person is executed (or eliminated).
- The elimination of people makes the circle smaller and smaller. At the last step, only one person remains who is declared the 'winner'.

Joseph's Problem

- If there are n number of people and a number k which indicates that k–1 people are skipped and kth person in the circle is eliminated. K=2

- Try the same process with n = 7 and k =3.
- The elimination goes in the sequence of 3, 6, 2, 7, 5 and 1.

Joseph's Problem

- Given the people = {Arya, Jon, Robb, Catelyn, Rose, Bran, Tyrion, Cersei, Sansa, Brienne}

k=4, Figure out name of the surviving person assuming that they are standing in the same sequence as given in the set. Show the solution step by step.

The Queue ADT: Value definition

Abstract typedef QueueType(ElementType ele)

Condition: none

Queue ADT: Operator definition

1. Abstract QueueType CreateQueue()

Precondition: none

Postcondition: Empty Queue is created

2. Abstract QueueType Enqueue(QueueType Queue, ElementType Element)

Precondition: Queue not full or NotFull(Queue)= True

Postcondition: Queue = Queue' + Element at the rear

Or Queue = original queue with new Element at the rear

Queue ADT: Operator definition

3. Abstract ElementType dequeue(QueueType Queue)

Precondition: Queue not empty or NotEmpty(Queue)= True

Postcondition: Dequeue= element at the front

Queue= Queue - Element at the front

Or Queue = original queue with front element deleted

4. Abstract DestroyQueue(QueueType Queue)

Precondition: Queue not empty or NotEmpty(Queue)= True

Postcondition: Element from the Queue are removed one by one starting from front to rear.

NotEmpty(Queue)= False

Queue ADT: Operator definition

5. Abstract Boolean NotFull(QueueType Queue)

Precondition: none

Postcondition: NotFull(Queue)= true if Queue is not full

NotFull(Queue)= False if Queue is full.

6. Abstract Boolean NotEmpty(QueueType Queue)

Precondition: none

Postcondition: NotEmpty(Queue)= true if queue is not empty

NotEmpty(Queue)= False if Queue is empty.

Thank you!