Subject Outlines:

Subject title : Data Structures

- **Text book:** An Introduction To Data Structures With Application
  By Tremblay, J.P., P.G. Sorenson. Second Edition.

- **Ref. book:** Data Structures Using C and C++
  By Langsam, Augenstein, Tenenbaum

Teaching Scheme:

Theory : 3 Lectures/week

Practical: 2 hrs / Batch/week

Examination Scheme:

External theory : 70 Marks , Internal Exam: 30 Marks = 100

Practical + Termwork : 50 (25+25) Total=150 Marks

Data Structure

- a **data structure** is a particular way of storing and organizing data so that it can be used efficiently.
- Data structure usually consists of two things:
- The type of structure that we are going to use to store data and how efficiently we perform operations onto that data stored in structure so that it results in less execution time and memory space require.
- Different kinds of data structures are suited to different kinds of applications

For example :

- An array data structure stores a number of elements of the same type in a specific order. Arrays may be fixed-length or expandable.
- Record : Records are among the simplest data structures. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names.

Data Structure

Data Structure describes,

- how to store a collection of objects in memory,
- what operations we can perform on that data,
- the algorithms for those operations, and
- how time and space efficient those algorithms are.

- A logical relationship among data elements that support specific data manipulation functions.

- Some data structures are linear in fashion. some are non-linear.
- In linear data structure processing of data items is done in linear fashion. Linear adjacency is exist among the data.

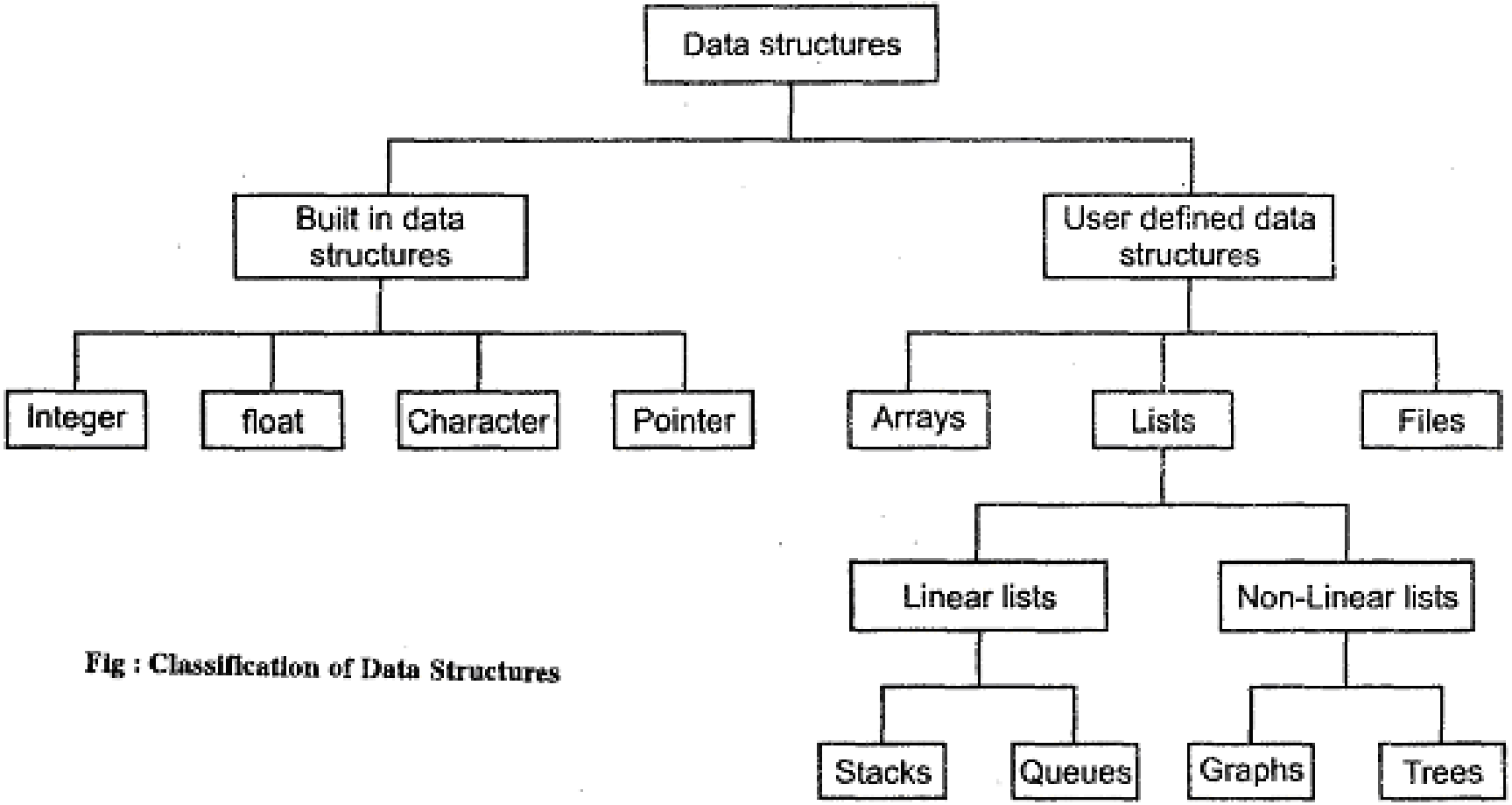


Fig : Classification of Data Structures

- **Primitive data structure**:
  - These are data structures that can be manipulated directly by machine instructions.
  - In C language, the different primitive data structures are int, float, char, double.
  - Primitive data are only single values, they have not special capabilities.
- **Non primitive data structures**:
  - These are data structures that can not be manipulated directly by machine instructions. Arrays, linked lists, files etc., are some of non-primitive data structures and are classified into *linear data structures* and *non-linear data structures*.
  - The non-primitive data types are used to store the group of values.

- Linear data structure:
- In linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory.
- Some commonly used linear data structures are arrays, linked lists, stacks and queues.

- Non-linear data structure:
- In nonlinear data structures, data elements are not organized in a sequential fashion.
- they might be difficult to be implemented in computer's linear memory .
- Trees, graphs and files are non-linear data structures.

QUEUE Data Structure

- A Queue is a special kind of data structure in which elements are added at one end (called the **rear**), and elements are removed from the other end (called the **front**).

- You come across a number of examples of a queue in real life situations.

- For example, consider a line of students at a fee counter. Whenever a student enters the queue, he stands at the end of the queue.
- Every time the student at the front of the queue deposits the fee, he leaves the queue.
- The student who comes first in the queue is the one who leaves the queue first.

- Since the first item inserted is the first item removed, a queue is commonly called a first-in-first-out or a FIFO data structure.

QUEUE Data Structure

- Closely related to Stacks
- It is a linear Data Structure
- Queues are a first in first out data structure
  - FIFO
- Add items to the end of the queue
- Access and remove from the front
- Used extensively in operating systems
  - Queues of processes, I/O requests, and much more

QUEUE Data Structure

How do we keep track which element to be removed and where the new element to be added to queue???????

We require to maintain Two pointer in order to perform the insertion and deletion operations in queue.

The Rear pointer is used to insert an elements in a queue.

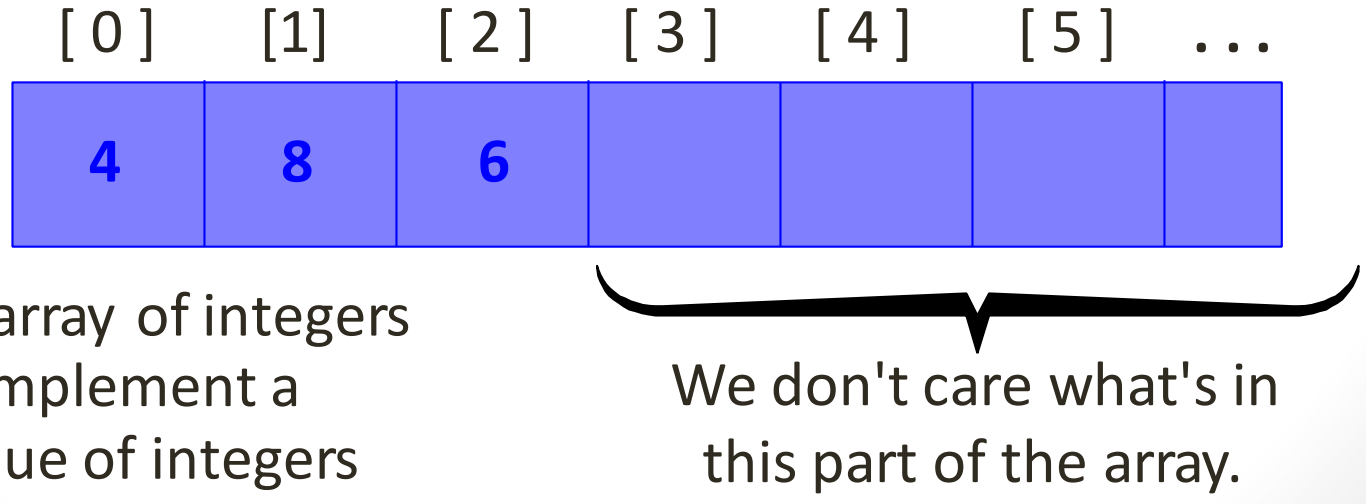
The Front pointer is used to delete an elements from the queue.

Array Implementation of QUEUE:

1. **Array Declaration**
2. **Front pointer**
3. **Rear Pointer**

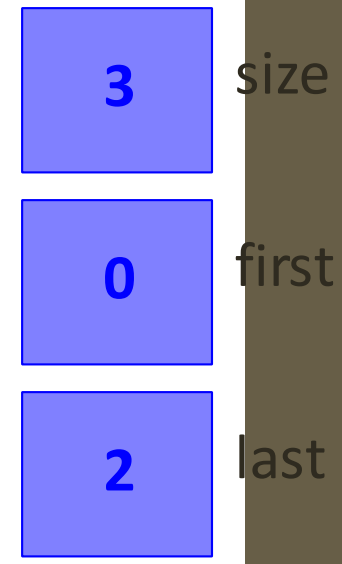
Array Implementation

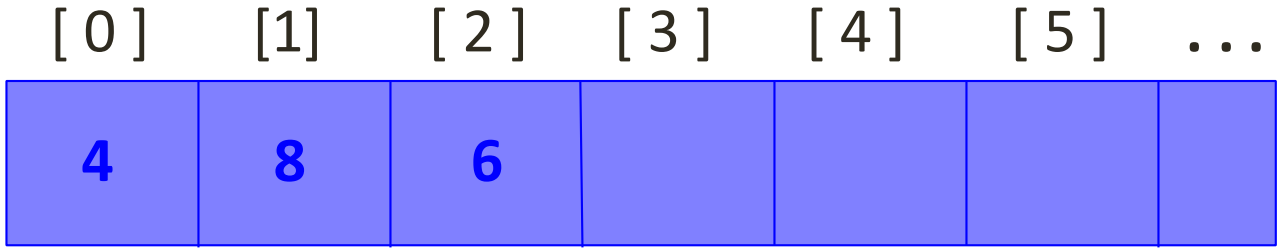
- A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).



Array Implementation

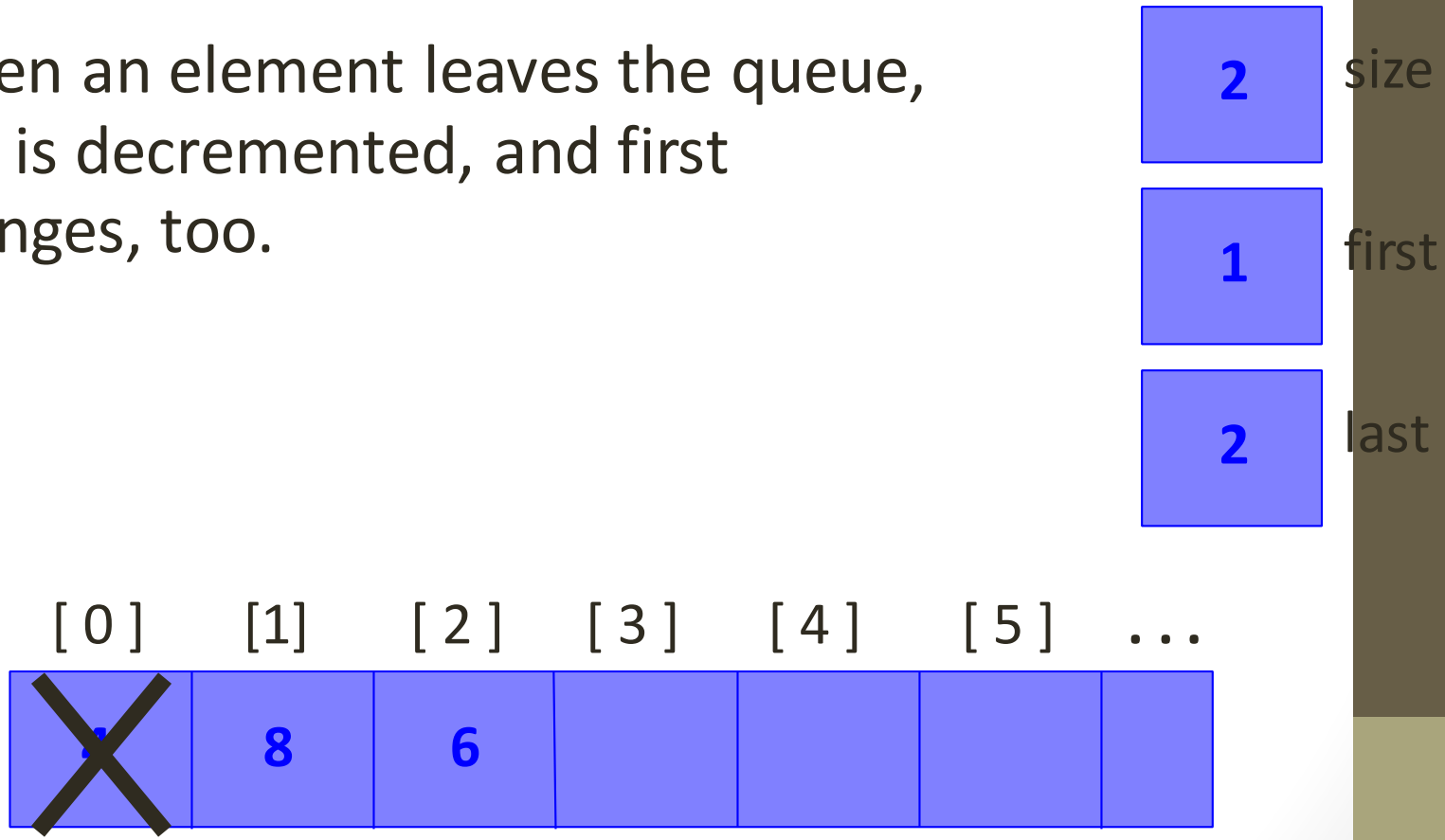
- The easiest implementation also keeps track of the number of items in the queue and the index of the first element (at the front of the queue), the last element (at the rear).





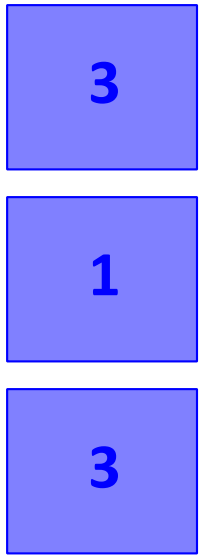
A Dequeue Operation

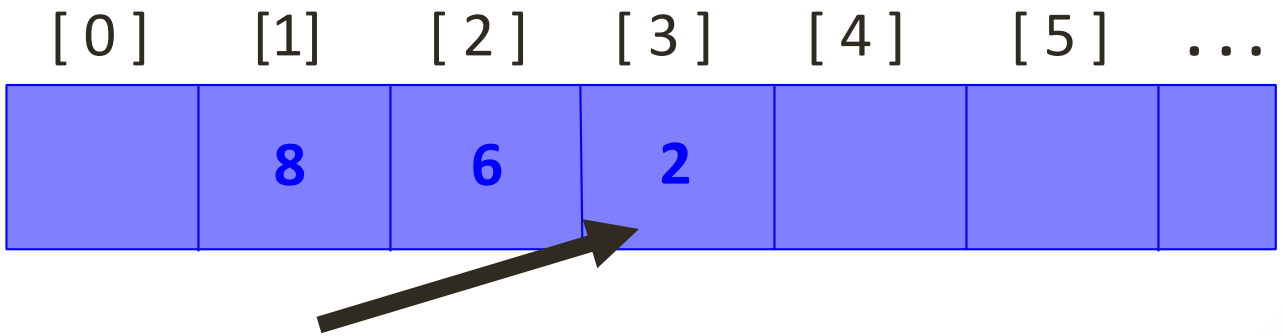
- When an element leaves the queue, size is decremented, and first changes, too.



An Enqueue Operation

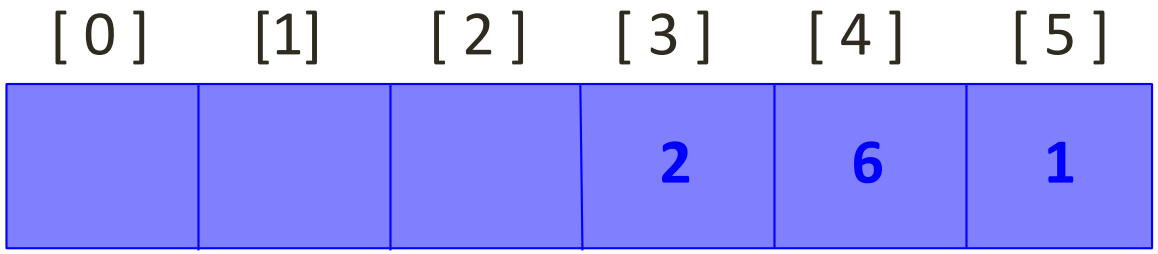
- When an element enters the queue, size is incremented, and last changes, too.

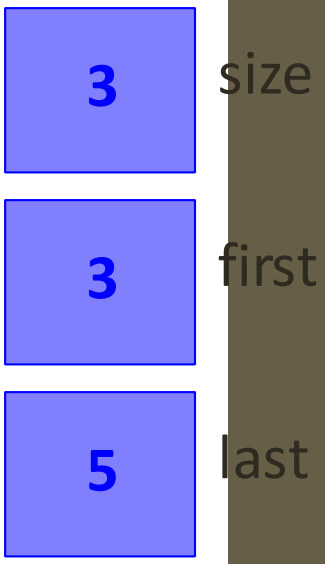




At the End of the Array

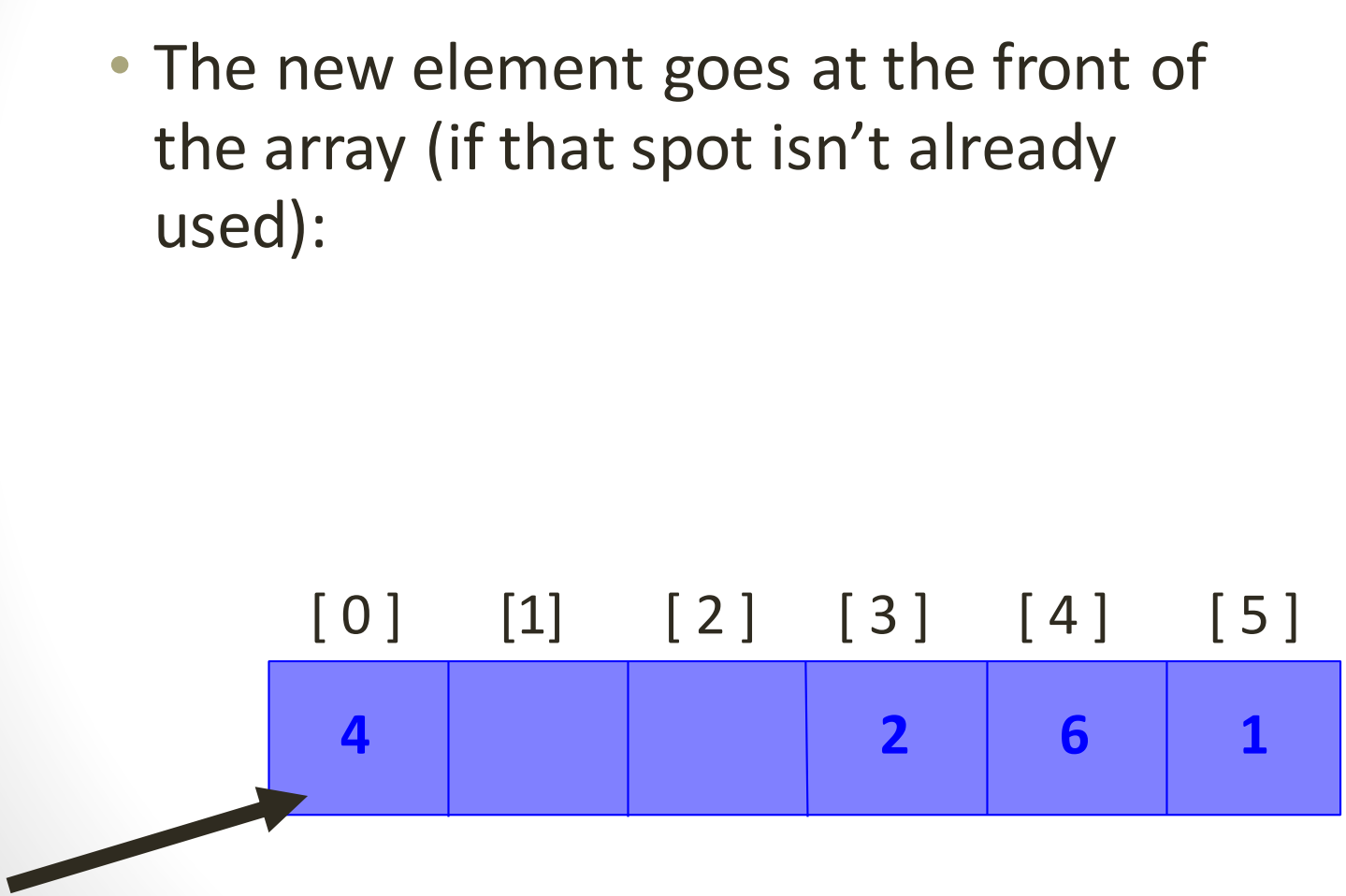
- There is special behavior at the end of the array. For example, suppose we want to add a new element to this queue, where the last index is [5]:





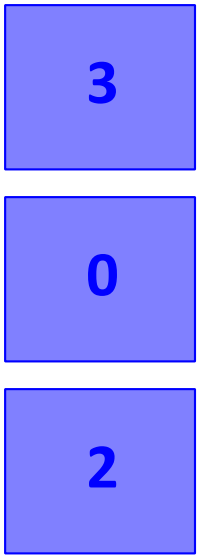
At the End of the Array

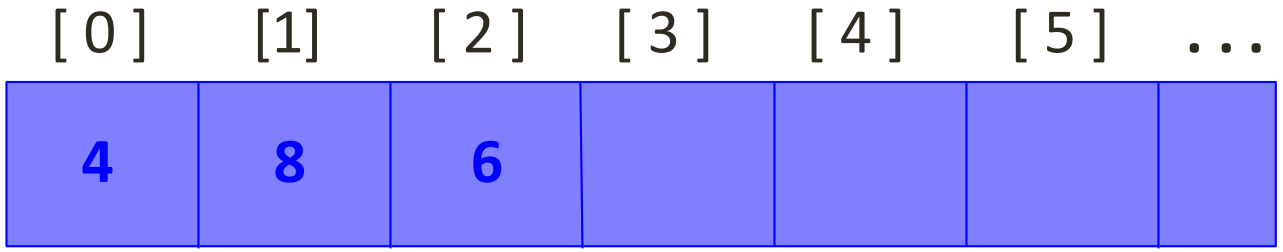
- The new element goes at the front of the array (if that spot isn't already used):



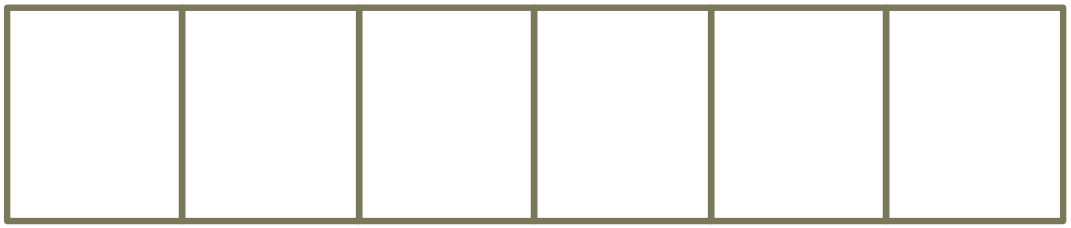
Array Implementation

- Easy to implement
- But it has a limited capacity with a fixed array
- Or you must use a dynamic array for an unbounded capacity
- Special behavior is needed when the rear reaches the end of the array.





QUEUE Data Structure



Basic Queue Operations

- Creating and Initializing the Queue.
- Inserting an element into Queue.
- Deleting an element from the Queue.
- Is the Queue empty?
- Is the Queue full?

Queues in computer science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send

- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order

- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Creating Queue

- There are Two way to implement the Queue Data structure

- **Static Implementation of queue**
- **Dynamic Creation of queue.**

- **The first approach is implementing Queue using ARRAY.**

- **The Second approach is implementing Queue using Linked List(pointer).**

Array Implementation of Queue

Array implementation of queue require followings:

- Declare an array of appropriate size.
- Create two index variable which will work as a front and rear of the queue.

- During queue operations queue may grow or shrink within the space reserved for it.

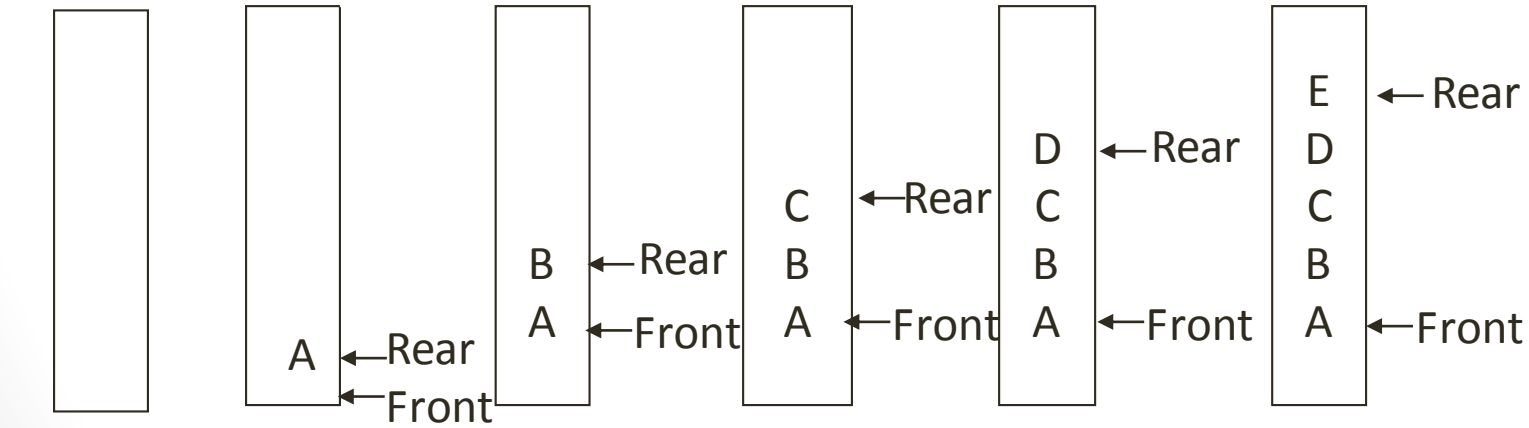
- When queue is empty ,front and rear variable has value -1 .when queue has one element ,front and rear variable has value 0,when queue has 2 elements then front variable has value 0 and rear variable has value 1 ,so on called queue is growing.

Array Implementation of Queue

- When queue has **Size-1** elements then we can say queue is full. Next insert operation put the queue in overflow condition.
- In short, every insert operation increment the value of rear variable by 1.
- Similarly every delete operation remove front element from the queue and increment the front variable by 1.
- When front and rear variable point to the same position means queue is empty and both initialization with -1.

Array Implementation of Queue

- https://www.cs.usfca.edu/~galles/visualization/QueueArray.html



QUEUE Data Structure

QINSERT(Q, F, R,N,X):- This algorithm is used to insert all N elements in queue Q. N is integer type variable indicate there are N elements in queue Q. F and R pointers to the front and rear elements of a queue and insert X element at the rear of the queue.

1. [check Queue is Overflow or not]

if R >= N-1 then

write(" Queue Overflow");

return

2. [increment rear pointer by 1]

R <- R +1

3. [insert element x]

Q[R] <- X

3. [Increment Front pointer if it is first insert operation]

if F = -1 then

F <- 0

4.[finish]

Exit

QUEUE Data Structure

QDELETE(Q, F, R):- This algorithm is used to delete an element from queue Q. F and R pointers to the front and rear elements of a queue. X is a temporary variable.

1. [check Queue is Underflow or not?]

if F = -1 then

write(" Queue Underflow");

return

2. [Delete the element from the location pointed by F]

X <- Q[F]

3. [Check after deletion, queue is empty?]

if F = R then

F⟵ R ⟵ -1

else

F <- F+1

4.[finish]

Return X

Disadvantages of a Queue

- The operations such as inserting an element and deleting an element from queue works perfectly until the rear index reaches the end of the array.
- If some items are deleted from the front, there will be some empty spaces in the beginning of the queue. Since the rear index points to the end of the array, the queue is thought to be full and no more insertions are possible.
- This disadvantage can be eliminated by removing the item in the front end of the queue and then shifting the data from the second locations onwards to their respective previous locations.
- Another option is to use a circular queue.

Implement Static Queue

```
void qinsert(int q[],int n,int *f,int *r,int x)
        {
                if(*r>=(n-1))
                        {
                                printf("Queue overflow");
                                return;
                         }
                *r = *r + 1;
                q[*r]=x;
                if(*f==-1)
                        *f=0;
        }
```

int qdelete (int q[],int *f,int *r)

```
{
        int a;
        if(*f==-1)
                return 0;
        a=q[*f];
        if(*f==*r)
          {
                *f=-1;
                *r=-1;
           }
       else
                *f=*f+1;
        return(a);
}
```

```
int isempty(int *f)
        {
                if(*f==-1)
                        return 1;
                else
                        return 0;
        }
int isfull(int *f,int *r,int n)
        {
                if( (*r >=(n-1))
                        return 1;
                else
                        return 0;
        }
```

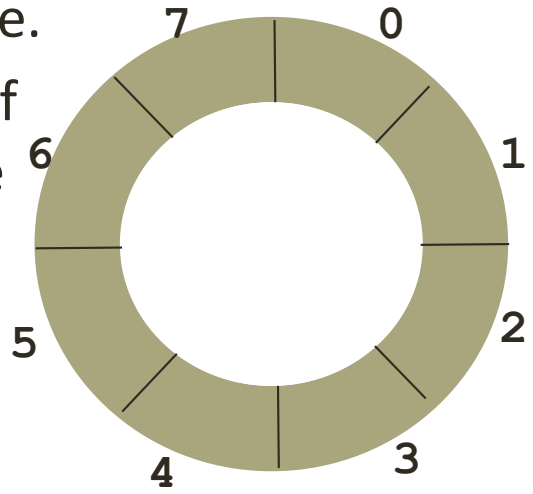
```
void display(int q[],int *f,int *r)
    {
        int i;
        if(*f==-1)
            printf("Queue is empty");
        else
            {
                printf("\nThe queue contains:\n\n");
                for(i=*f;i<=*r;i++)
                    printf("%d  ",q[i]);
            }
        getch();
    }
```

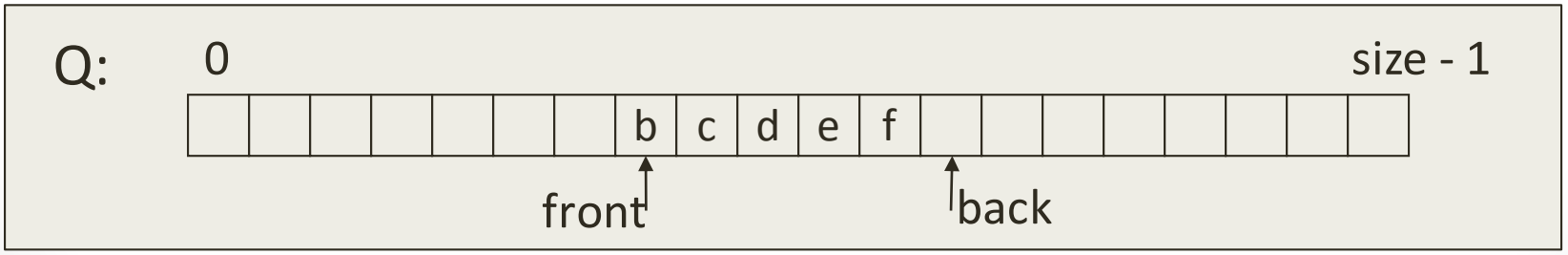
Initialize

```
void main()
        {
                int n=5,*f,*r,i,j,k;
                int q[5];
                *f=-1;
                *r=-1;
```

Implementing Queue ADT: Circular Array Queue

- **Neat trick**: use a ***circular array*** to insert and remove items from a queue in constant time.
- The idea of a circular array is that the end of the array "wraps around" to the start of the array.





http://www.csanimated.com/animation.php?t=Circular_buffer

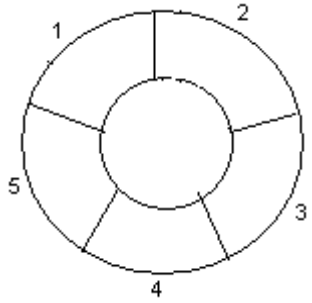
Implementation

- Two pointers called FRONT and REAR are used to keep track of the first and last elements in the queue.
- When initializing the queue, we set the value of FRONT and REAR to -1.
- On enqueing an element, we circularly increase the value of REAR index and place the new element in the position pointed to by REAR.
- On dequeueing an element, we return the value pointed to by FRONT and circularly increase the FRONT index.
- Before enqueing, we check if queue is already full.
- Before dequeuing, we check if queue is already empty.
- When enqueing the first element, we set the value of FRONT to 0.
- When dequeing the last element, we reset the values of FRONT and REARto -1.

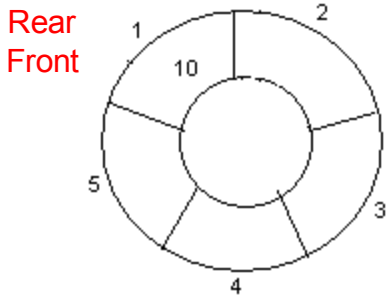
- However, the check for full queue has a new additional case:
- Case 1: FRONT = 0 && REAR == SIZE - 1
- Case 2: FRONT = REAR + 1
- The second case happens when REAR starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.

Example: Consider the following circular queue with N = 5.

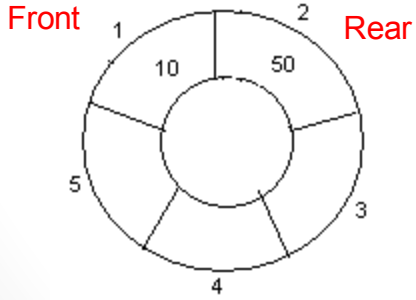
1. Initially, Rear = 0, Front = 0.



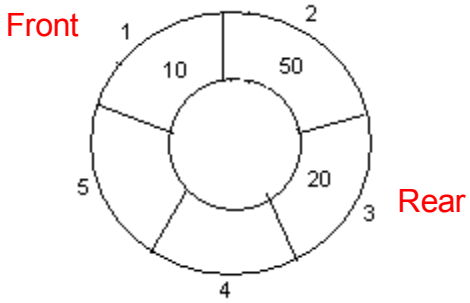
2. Insert 10, Rear = 1, Front = 1.



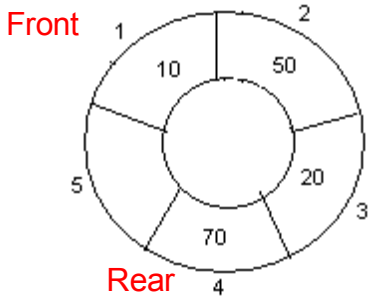
3. Insert 50, Rear = 2, Front = 1.



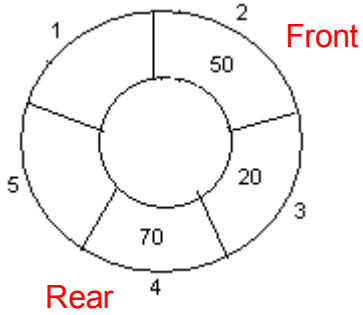
4. Insert 20, Rear = 3, Front = 0.



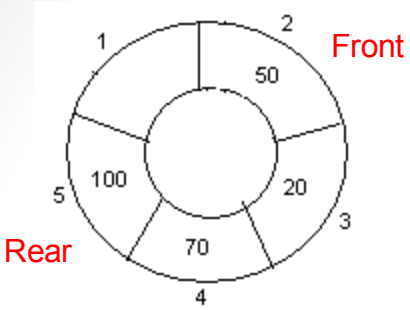
5. Insert 70, Rear = 4, Front = 1.



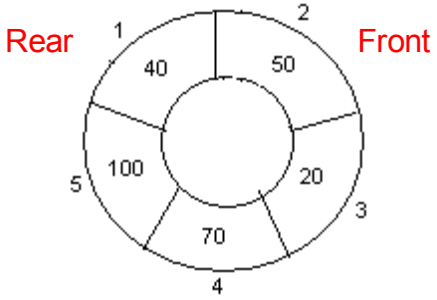
6. Delete front, Rear = 4, Front = 2.



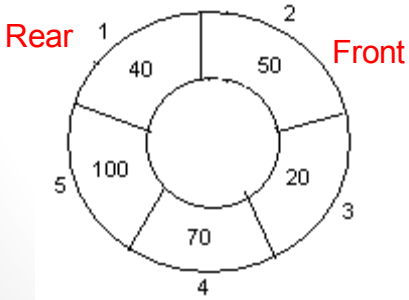
7. Insert 100, Rear = 5, Front = 2.



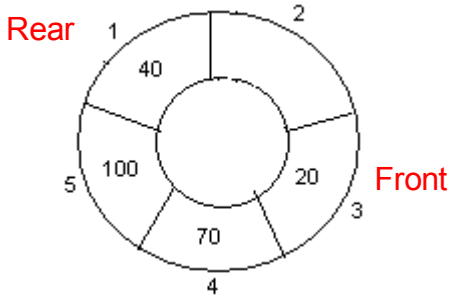
8. Insert 40, Rear = 1, Front = 2.



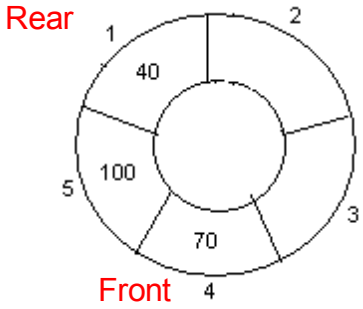
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.



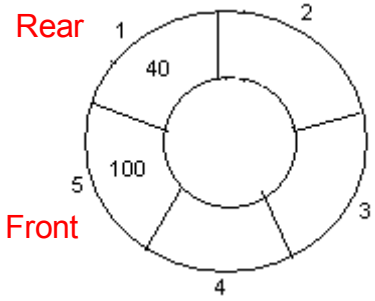
10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.



QUEUE Data Structure

CQINSERT(Q, F, R,X,N):- This algorithm is used to insert all N elements in circular queue Q. N is integer type variable indicate there are N elements in circular queue Q. F and R pointers to the front and rear elements of a circular queue and insert X element at the rear of the queue.

1. [check Queue is Overflow or not]

if (F=0 and R = Size-1) or (F= R+1)

write(" Queue Overflow");

return /*if((F == R+1) || (F ==0 && R == N -1))*/

2. [increment rear pointer by 1]

if R= size then

R <- 1

else

R <- R +1 /* R = (R + 1) % N */

3. [insert element x]

Q[R] <- X

3. [Increment Front pointer if it is first insert operation]

if F = -1then

F <- 0

4.[finish] Exit

QUEUE Data Structure

CQDELETE(Q, F, R,N):- This algorithm is used to delete an element from queue Q. F and R pointers to the front and rear elements of a queue. X is a temporary variable.

1. [check CQueue is Underflow or not?]

if R=F=-1 then

write(" Queue Underflow");

return

2. [Delete the element from the location pointed by F]

X <- Q[F]

3. [Check after deletion, queue is empty?]

if F = R then

F ⟵ R ⟵ -1

Return X

4.[Increment the front pointer]

if F = N then F <- 1

else

F <- F+1 /* F = (F + 1) %N */

5.[finish]

Return X

Circular Queue-

```
int isFull()
 {
if( (front == rear + 1) || (front == 0 && rear == SIZE-1))
 return 1;
 return 0;
 }
int isEmpty()
 {
if(front == -1)
 return 1;
 return 0; }
```

Circular Queue-Enqueue

```
void enQueue(int element)
{
 if(isFull())
 printf("\n Queue is full!! \n");
 else
{
 if(front == -1)
 front = 0;
rear = (rear + 1) % SIZE;
items[rear] = element;
printf("\n Inserted -> %d", element);
 }
 }
```

- int deQueue()
- { int element;
- if(isEmpty())
- { printf("\n Queue is empty !! \n");
- return(-1); }
- else {
- element = items[front];
- if (front == rear)
- { front = -1; rear = -1; } /* Q has only one element, so we reset the queue after dequeing it. ? */
- else { front = (front + 1) % SIZE; }
- printf("\n Deleted element -> %d \n", element);
  return(element);
- }
- }

Drawback of Circular Queue

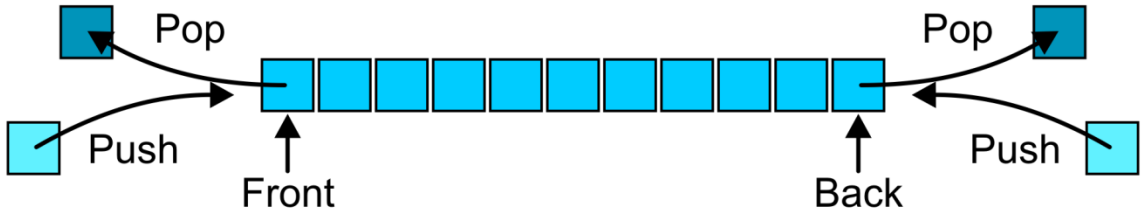
- The drawback of circular queue is , difficult to distinguished the full and empty cases. It is also known as **boundary case problem**.
- In circular queue it is necessary that:
- Before insertion, fullness of Queue must be checked (for overflow).
- Before deletion, emptiness of Queue must be checked (for underflow).

Difference

FOR COMPARISON	LINEAR QUEUE	CIRCULAR QUEUE
Basic	Organizes the data elements and instructions in a sequential order one after the other.	Arranges the data in the circular pattern where the last element is connected to the first element.
Order of task execution	Tasks are executed in order they were placed before (FIFO).	Order of executing a task may change.
Insertion and deletion	The new element is added from the rear end and removed from the front.	Insertion and deletion can be done at any position.
Performance	Inefficient	Works better than the linear queue.

Deque(Double-Ended Queue)

- A deque, also known as a double-ended queue, is an ordered collection of items similar to the queue.
- Allows elements to be added or removed on either the ends.



TYPES OF DEQUE

- DeQueue can be represented in two ways :

- **Input restricted Deque**

- Elements can be inserted only at one end.
- Elements can be removed from both the ends.

- **Output restricted Deque**

- Elements can be removed only at one end.
- Elements can be inserted from both the ends.

Deque as Stack and Queue

As STACK

- When insertion and deletion is made at the same side.

As Queue

- When items are inserted at one end and removed at the other end.

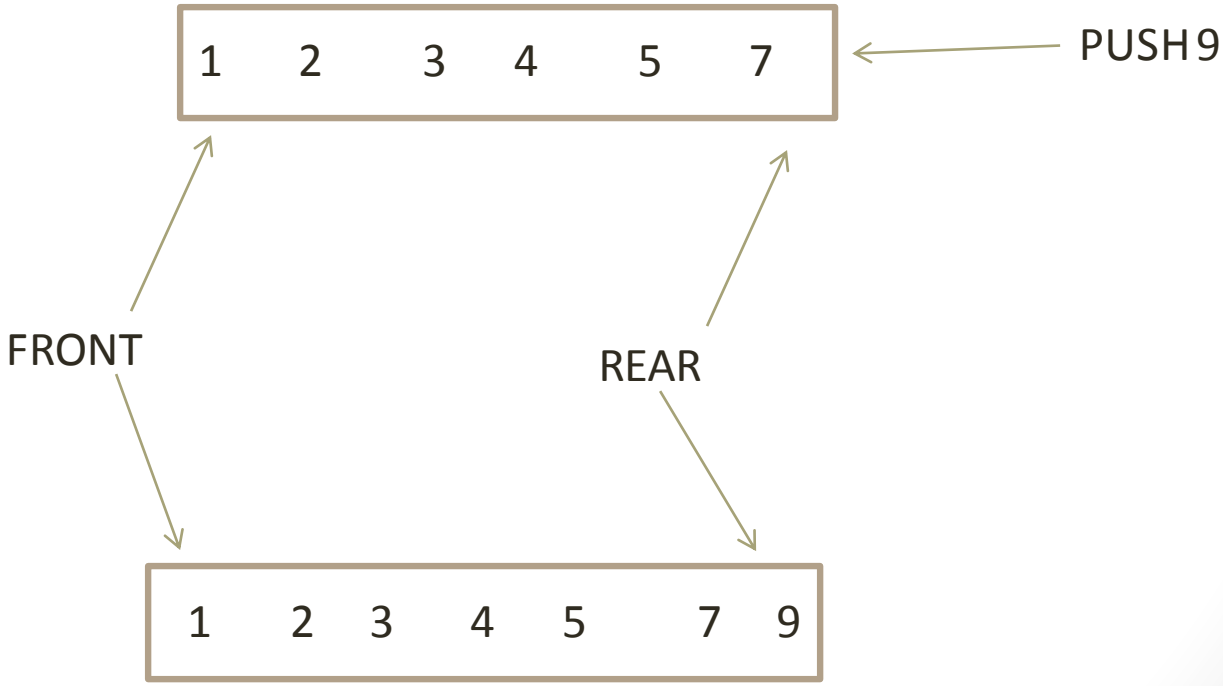
- This hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

OPERATIONS IN DEQUE

- Insert element at right
- Insert element at left
- Remove element at left
- Remove element at right

Insert_right

- Insert_right() is a operation used to insert an element at the right of a *Deque*.



Alogrithm insert_right

QINSERT(Q, F, R,N,X):- This algorithm is used to insert all N elements in queue Q. N is integer type variable indicate there are N elements in queue Q. F and R pointers to the front and rear elements of a queue and insert X element at the rear of the queue.

1. [check Queue is Overflow or not]

if R >= N-1 then

write(" Queue Overflow");

return

2. [increment rear pointer by 1]

R <- R +1

3. [insert element x]

Q[R] <- X

3. [Increment Front pointer if it is first insert operation]

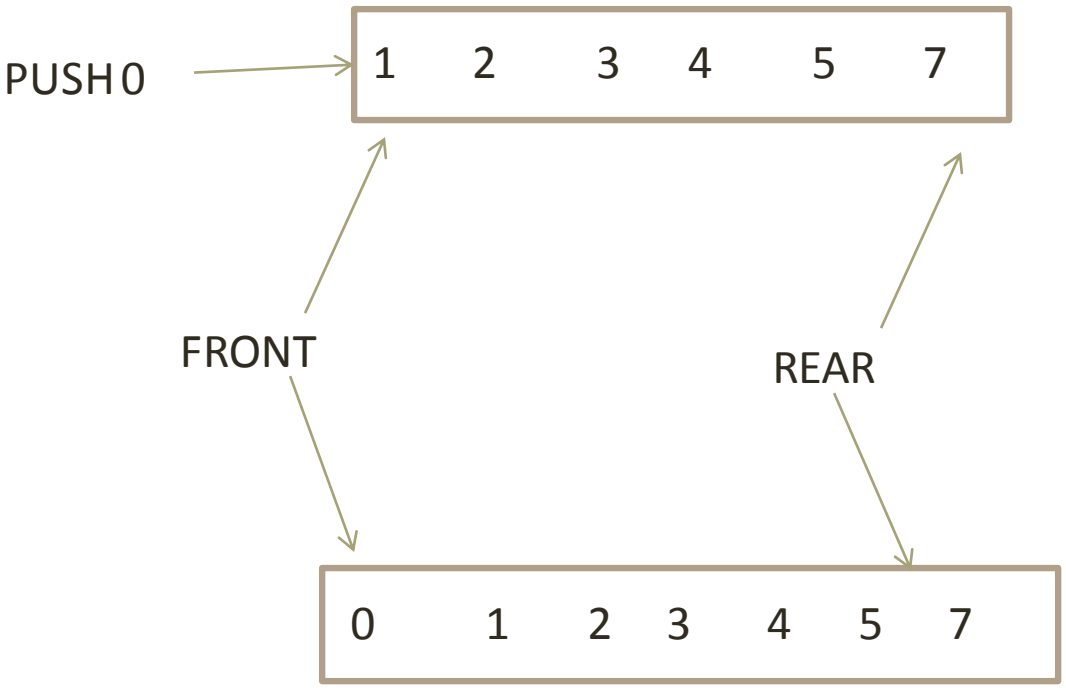
if F = -1 then

F <- 0

4.[finish] Exit

Insert_left

- insert_left() is a operation used to insert an element into the left of the *Deque*.



Algorithm Insert_left

Qinsert_Left(Q, F, X):- This algorithm is used to insert an element from left of the queue Q. F pointers to the front elements of a queue. X is a temporary variable.

1. [check Queue is Underflow or not?]

if F ==-1 then

write(" Queue Underflow");

return

2. [decrement front pointer by 1]

F <- F -1

3. [insert element x]

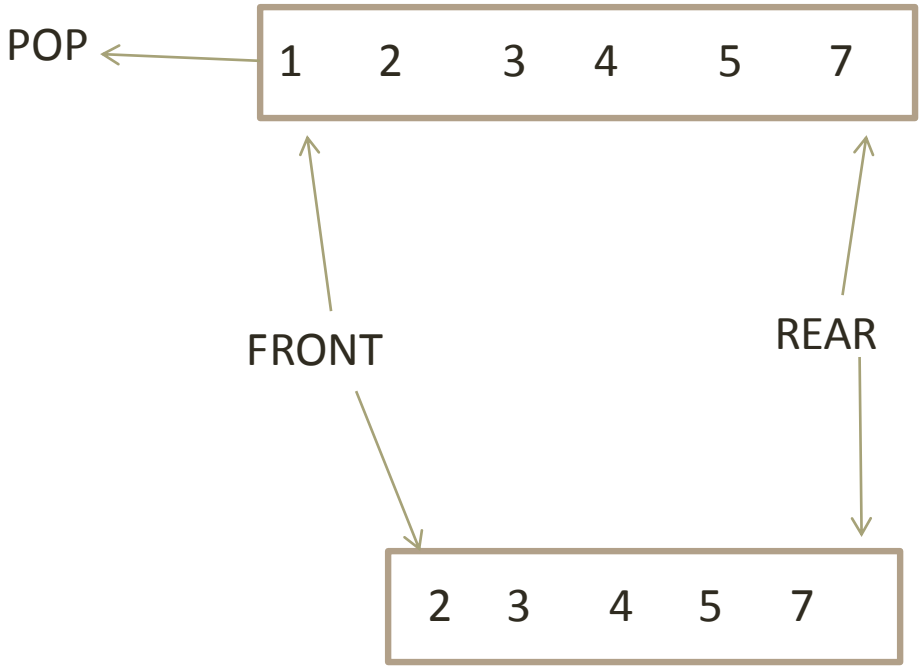
Q[F] <- X

4.[finish]

exit

Remove_left

- remove_left() is a operation used to pop an element on front of the *Deque*.



Remove_left

QDELETE_Left(Q, F, R):- This algorithm is used to delete an element from queue Q. F and R pointers to the front and rear elements of a queue. X is a temporary variable.

1. [check Queue is Underflow or not?]

```
if   F == -1  then
write(" Queue Underflow");
return
```

2. [Delete the element from the location pointed by F]

```
X <- Q[F]
```

3. [Check after deletion, queue is empty?]

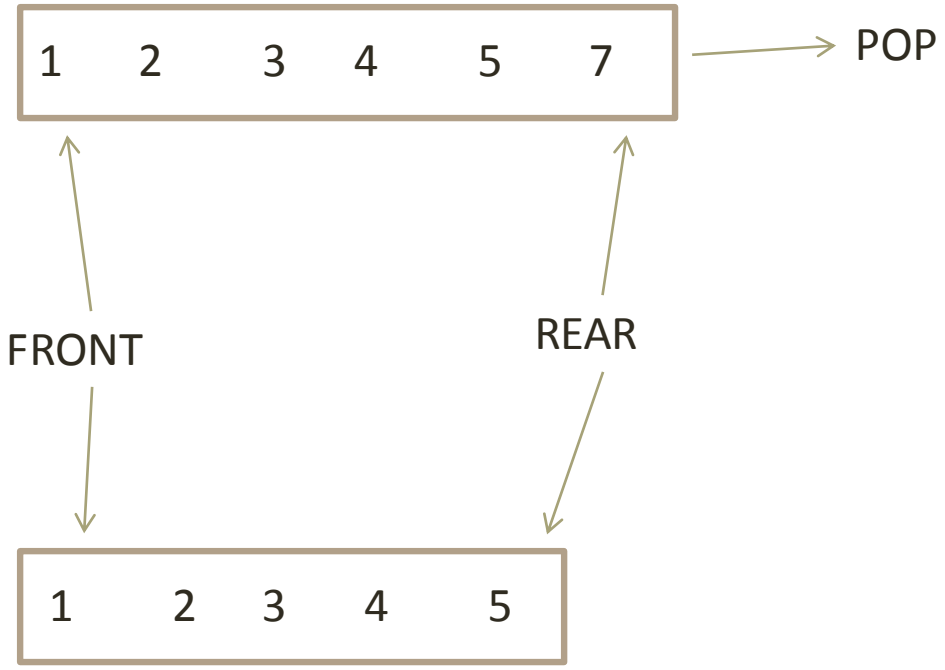
```
if F =  R then
    F <-    R <-      -1
else
    F  <-  F+1
```

4.[finish]

```
Return  X
```

Remove_back

- remove_front() is a operation used to pop an element on front of the *Deque*.



Alogrithm Remove_right

QDELETE(Q, F, R):- This algorithm is used to delete an element from queue Q. F and R pointers to the front and rear elements of a queue. X is a temporary variable.

1. [check Queue is Underflow or not?]

```
if   R == -1  then
write(" Queue Underflow");
return
```

2. [Delete the element from the location pointed by F]

```
X <- Q[R]
```

3. [Check after deletion, queue is empty?]

```
if F =  R then
    F <-    R <-      -1
else
    R  <-  R-1
```

4.[finish]

```
Return  X
```

Priority Queue

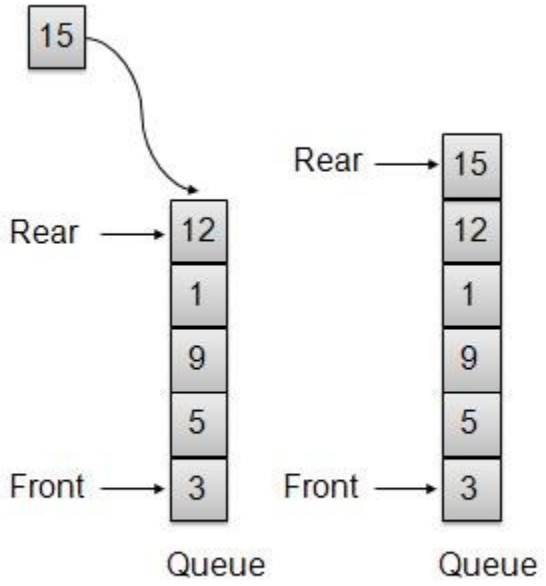
- Queues are a standard mechanism for ordering tasks on a first-come, first-served basis
- However, some tasks may be more important or timely than others.
- Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference.
- In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

- Basic Operations
- insert / enqueue – add an item to the rear of the queue.

- remove / dequeue – remove an item from the front of the queue.

Insert / Enqueue Operation

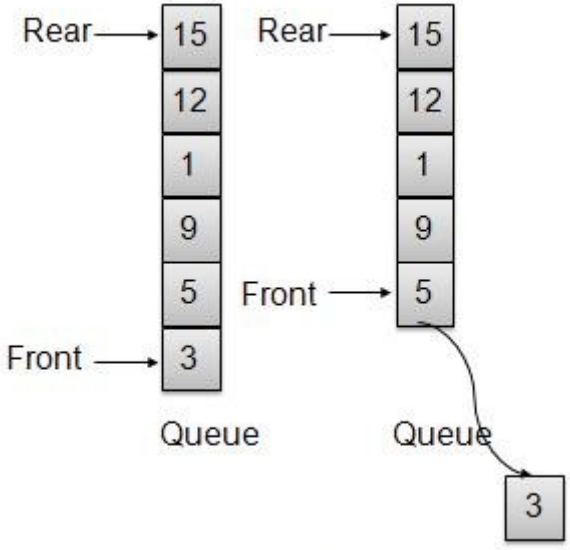
- Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



One item inserted at rear end

Remove / Dequeue Operation

- Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.
-



One Item removed from front

PQ={}

- insert(5,A) PQ={(5,A)}
- insert(9,C) PQ={(5,A), (9,C)}
- insert(3,B) PQ={(5,A), (9,C), (3,B)}
- insert(7,D) PQ={(5,A), (7,D), (9,C), (3,B)}
- min() return (3,B)
- removeMin() PQ = {(5,A), (7,D), (9,C)}
- size() return 3
- removeMin() return (5,A) PQ={(7,D), (9,C)}
- removeMin() return (7,D) PQ={(9,C)}
- removeMin() return (9,C) PQ={}

- **Strengths:**
- **Quickly access the highest-priority item.** Priority queues allow you to peek at the top item in O(1)$O(1)$ while keeping other operations relatively cheap (O(lg(n))$O(lg(n))$).
- **Weaknesses:**
- **Slow enqueues and dequeues.** Both operations take O(\lg(n))$O(\lg(n))$ time with priority queues. With normal first-in, first-out queues, these operations are O(1)$O(1)$ time.

Applications of Priority Queue

- Any time you want to handle things with different priority levels: triaging patients in a hospital, locating the closest available taxi, or just a to-do list.
- **Operating system schedulers** may use priority queues to select the next process to run, ensuring high-priority tasks run before low-priority ones.
- Certain **foundational algorithms** rely on priority queues:
  - Dijkstra's shortest-path
  - A* search (a graph traversal algorithm like BFS)
  - Huffman codes (an encoding for data compression)

Applications of Queues

- Queues are also very useful in a time-sharing multi-user operating system where many users share the CPU simultaneously.

- Whenever a user requests the CPU to run a particular program, the operating system adds the request (by first of all converting the program into a process that is a running instance of the program, and assigning the process an ID).

- **This process ID is then added at the end of the queue of jobs waiting to be executed.**

- Whenever the CPU is free, it executes the job that is at the front of the job queue.

Applications of Queues

- Similarly, there are queues for shared I/O devices. Each device maintains its own queue of requests.
- An example is a print queue on a network printer, which queues up print jobs issued by various users on the network.
- The first print request is the first one to be processed. New print requests are added at the end of the queue.

Difference

#	STACK	QUEUE
1	Objects are inserted and removed at the same end.	Objects are inserted and removed from different ends.
2	In stacks only one pointer is used. It points to the top of the stack.	In queues, two different pointers are used for front and rear ends.
3	In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
4	Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
5	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
6	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
7	Collection of dinner plates at a wedding reception is an example of stack.	People standing in a file to board a bus is an example of queue.