SRI MANAKULA VINAYAGAR ENGINEERING COLLEGE
(An Autonomous Institution)
(Approved by AICTE, New Delhi & Affiliated to Pondicherry University)
(Accredited by NBA-AICTE, New Delhi, ISO 9001:2000 Certified Institution & Accredited by NAAC with "A" Grade)
Madagadipet, Puducherry - 605 107

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Subject Name: Data Structures
Subject Code: U20EST356

UNIT – II

Stacks and Queues: ADT Stack and its operations. Applications of Stacks: Expression Conversion and evaluation. ADT Queue and its operations. Types of Queue: Simple Queue – Circular Queue – Priority Queue – Deque

2 MARKS

1. **What is Stack?**
   - Stack is a linear and static data structure.
   - Stack is an ordered collection of elements in which we can insert and delete the elements at one end called Top.
   - Initial condition of the stack Top=-1.
   - It is otherwise called as LIFO (Last In First Out).

2. **What are the various operations that can be performed on stack?**
   In Stack, we can perform two operations namely Push and Pop.
   Push: Push means inserting a new element into the stack. Insertion can be done by incrementing top by 1.
   Pop: Pop means deleting an element from the stack. Deletion can be done by decrementing top by 1.
   Peek: Returns the top element and an error occurs if the stack is empty.

3. **What do you mean by Top in Stack?**
   - Top is the pointer which always points the top element of the Stack.
   - If Top =-1, then Stack is Empty.
   - We can insert a new element into stack by incrementing top by 1.
   - We can delete an element from stack by decrementing top by 1.

4. **What are the applications of Stack?**
   Some of the applications of Stack are:
   i. Matching of nested parenthesis in a mathematical expression.
   ii. Conversion of infix to postfix.
   iii. Evaluation of postfix.

5. **What is ADT in data structure with example?**
   The **Data** Type is basically a type of **data** that can be used in different computer program. The **ADT** is made of with primitive datatypes, but operation logics are hidden. Some **examples** of **ADT** are Stack, Queue, List etc.

6. **Why stack and queue is called ADT?**
   Stack and Queue are referred as abstract datatype because in stack there are, mainly two operations push and pop and in queue there are insertion and deletion. Which are when operated on any set of data then it is free from that which type of data that must be contained by the set.

7. **What are types of Expression?**
   Expression is classified as three types according to position of operator with respect to the operands. They are:
   i. **Infix**: The operator is placed in between two operands. E.g.: A+B.
   ii. **Postfix**: The operator is placed after the operands. E.g.: AB+.
   iii. **Prefix**: The operator is placed before the operands. E.g.: +AB.

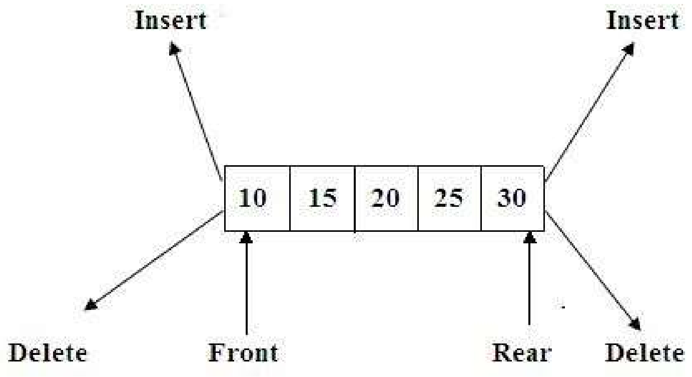
8. **What is Queue?**
   - Queue is a linear and static data structure.
   - Queue is an ordered collection of elements in which we can insert an element at one end called Rear and delete an element at another end called Front.
   - Initial condition of the stack Rear=Front=-1. It is otherwise called as FIFO (First In First Out).
9. **What are the various operations that can be performed on Queue?**
   In Queue, we can perform two operations namely Insertion and Deletion.
   Insertion: Insertion can be done by incrementing Rear by 1.
   Deletion: Deletion can be done by incrementing Front by 1

10. **What do you mean by Queue empty?**
    - If Queue has no data, then it is called as Queue Empty.
    - Queue may be empty on two conditions.
      i. Front==Rear==-1 and
      ii. Front==Rear.

11. **Difference between Stack and Queue.**

Stack	Queue
➢ A Stack is an ordered collection of data items, where all insertion and deletions always are made at the end of the sequence called Top.	➢ A Queue is an ordered collection of items, where all insertions are made at the end of the sequence called Rear and all deletions always are made from the beginning of the sequence called Front.
➢ In Stack, we get data items out in reverse order compared to the order they have been put into the stack.	➢ In Queue, we get data items out in same order compared to the order they have been put into the queue.
➢ Stack is otherwise called as FIFO.	➢ Queue is otherwise called as LIFO.

12. **Define Dequeue?**
    - Dequeue is otherwise called as Double ended queue.
    - In Dequeue, we can insert and delete at both ends either front or rear.



13. **What are the types of Queue?**
    - Simple Queue
    - Circular Queue
    - Priority Queue
    - Double ended Queue

14. **What is difference between Queue and Dequeue?**
    - A queue is designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue.
    - Whereas Dequeue represents a queue where you can insert and remove elements from both ends of the queue.

15. What is Priority Queue with example?

- ➢ A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority.
- ➢ Generally, the value of the element itself is considered for assigning the priority.
- ➢ For example: The element with the highest value is considered as the highest priority element.

16. Define Ascending Priority Queue.

- ➢ In this elements are placed in ascending order.
- ➢ The first smallest element is placed in first position and second smallest element in second position and so on.
- ➢ The new data item is inserted in priority queue without affecting the ascending order of queue.

17. Define Descending Priority Queue.

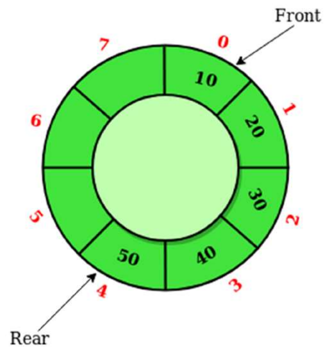
- ➢ In this elements are placed in descending order.
- ➢ The first highest element is placed in first position and second highest element is placed in second position and so on.
- ➢ The new data item is inserted in priority queue without affecting the descending order of queue.

18. What are the application of Priority Queue?

- ➢ Priority queues are used to sort heaps.
- ➢ Priority queues are used in operating system for load balancing and interrupt handling.
- ➢ Priority queues are used in huffman codes for data compression.
- ➢ In traffic light, depending upon the traffic, the colors will be given priority.

19. What is Circular Queue?

- ➢ Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle.
- ➢ The last position is connected back to the first position to make a circle.
- ➢ Circular Queue is used in memory management and scheduling process.
- ➢ It is also called 'Ring Buffer'.



20. How do you determine the size of Circular Queue?

Assuming you are using array of size N for queue implementation, then size of queue would be **size = (N-front + rear) mod N**. This formula work for both liner and circular queues.

21. How a stack is implemented using queue?

A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

- Method 1 (By making push operation costly)
- Method 2 (By making pop operation costly)

22. Write the limitations of stack.

- ➢ Only a small number of operations can be performed on it.
- ➢ It contains only a bounded capacity

23. Write the conditions to test "Queue is empty", "Queue is full" for a linear queue implemented in linear array?

If REAR = N (no. of elements), Queue is full

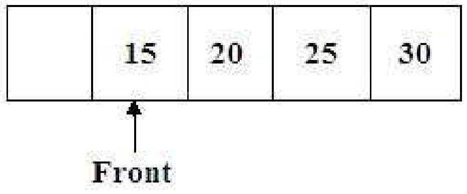
If FRONT =REAR, Queue is empty.

24. How to insert an element at the beginning of the Dequeue.

- First check the Dequeue is full or not.
- We have to check whether the element is first to be inserted if it is true then perform the following steps
  1. Shift the elements from left to right
  2. So that we need the number of elements present in the Dequeue , the position of the last element i.e. the position of the rear.
  3. Now insert any element in zeroth position.

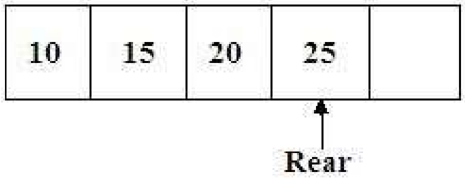
25. How to delete an element at the beginning of Dequeue.

- Delete the first element from the front position of the Dequeue
- The index of next element is stored in front pointer.
- Increment the front pointer by one.



26. How to delete an element at the end of Dequeue.

- The last element is deleted.
- The index of next element is stored in rear pointer.
- Decrement the rear pointer by one.



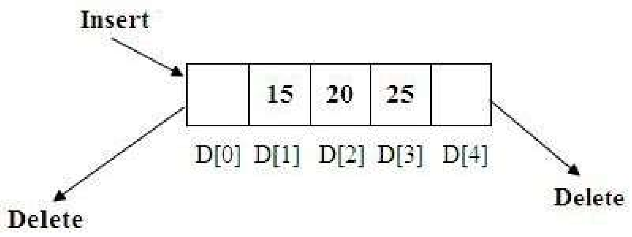
27. What are the types of Dequeue?

Two types of Dequeue are

1. Input Restricted Dequeue
2. Ouput Restricted Dequeue.

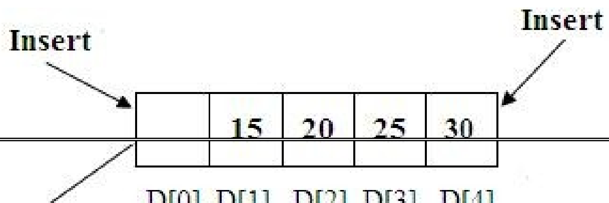
28. Define Input restricted Dequeue.

- It means we can insert the element only at one end and delete the elements at both ends.
- The above diagram shows the input restricted Dequeue.



29. Define Output restricted Dequeue.

- It means we can insert the elements in both ends and delete the elements at only one end.
- The above diagram shows the output restricted Dequeue.



30. State the advantages of using infix notations.
The advantages of using infix notations are,
- ➢ It is the mathematical way of representing the expression.
- ➢ It is easier to see visually which operation is done from first to last.

31. State the advantages of using postfix notations.
The advantages of using infix notations are
- ➢ We need not worry about the rules of precedence.
- ➢ We need to worry about the rules for right to left associatively.
- ➢ We need not parenthesis to override the above rules.

32. Write down the application of queue.
- Printing
- CPU scheduling
- Mail service
- Elevator
- Keyboard buffering

5 Marks

1. What is Stack? Explain with an example.
Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Operations:
- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

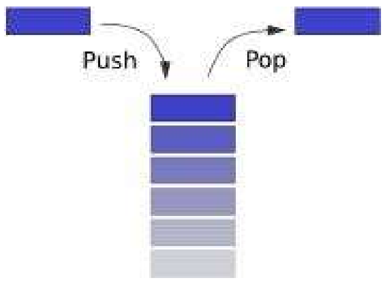


Fig: Simple representation of a Stack

Example for Stack:

Converting a decimal number into a binary number:
To solve this problem, we use a stack. We make use of the *LIFO* property of the stack. Initially we *push* the binary digit formed into the stack, instead of printing it directly. After the entire digit has been converted into the binary form, we *pop* one digit at a time from the stack and print it. Therefore, we get the decimal number is converted into its proper binary form.

Algorithm:
```
      1. Create a stack
      2. Enter a decimal number which has to be converted into its
equivalent binary form.
      3. iteration1 (while number > 0)
            3.1 digit = number % 2
            3.2 Push digit into the stack
            3.3 If the stack is full
                  3.3.1 Print an error
                  3.3.2 Stop the algorithm
            3.4 End the if condition
            3.5 Divide the number by
      4. End iteration1
      5. iteration2 (while stack is not empty
            5.1 Pop digit from the stack
            5.2 Print the digit
      6. End iteration2
      7. STOP
```

2. Explain the push operation on stack.

Stack operations may involve initializing the stack, using it and then de-initalizing it. Appart from these basic stuffs, a stack is used for the following two primary operations –

- **push() –** Pushing (storing) an element on the stack.
- **pop() –** Removing (accessing) an element from the stack.

Push Operation:
The process of putting a new data element onto stack is known as Push Operation. Push operation involves a series of steps –

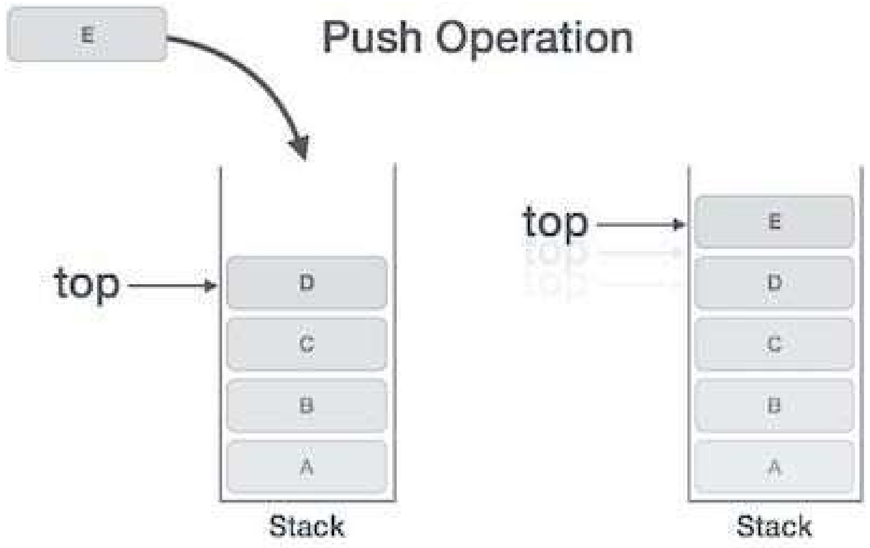
Step 1 - Checks if the stack is full.

Step 2 - If the stack is full, produces an error and exit.

Step 3 - If the stack is not full, increments **top** to point next empty space.

Step 4 - Adds data element to the stack location, where top is pointing.

Step 5 - Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

<u>Algorithm for PUSH Operation:</u>

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data
```

```
        if stack is full
              return null
        endif

        top ← top + 1
        stack[top] ← data

  end procedure
```

Example:

```
void push(int data) {
      if(!isFull()){
            top += 1;
            stack[top] = data;
      } else{
            printf("Could not insert, Stack  is full. \n");
      }
}
```

3. Explain the pop operation on stack.

Stack operations may involve initializing the stack, using it and then de-initalizing it. Appart from these basic stuffs, a stack is used for the following two primary operations –

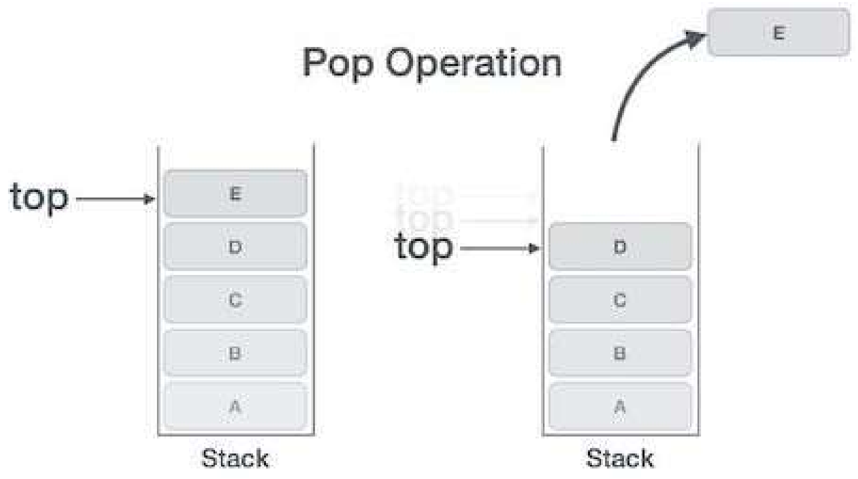
- **push() –** Pushing (storing) an element on the stack.
- **pop() –** Removing (accessing) an element from the stack.

POP Operation:

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation:

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack
     if stack is empty
          return null
     end if

     data ← stack[top]
     top ← top - 1
     return data
```

Implementation of this algorithm in C, is as follows –

Example:

```
int pop(int data){
     if(!isempty()){
          data = stack[top];
          top = top - 1;
          return data;
     }else{
          printf("Could not retrieve, Stack is empty.\n");
     } }
```

4. Explain the conversion of Infix to Postfix expression using Algorithm

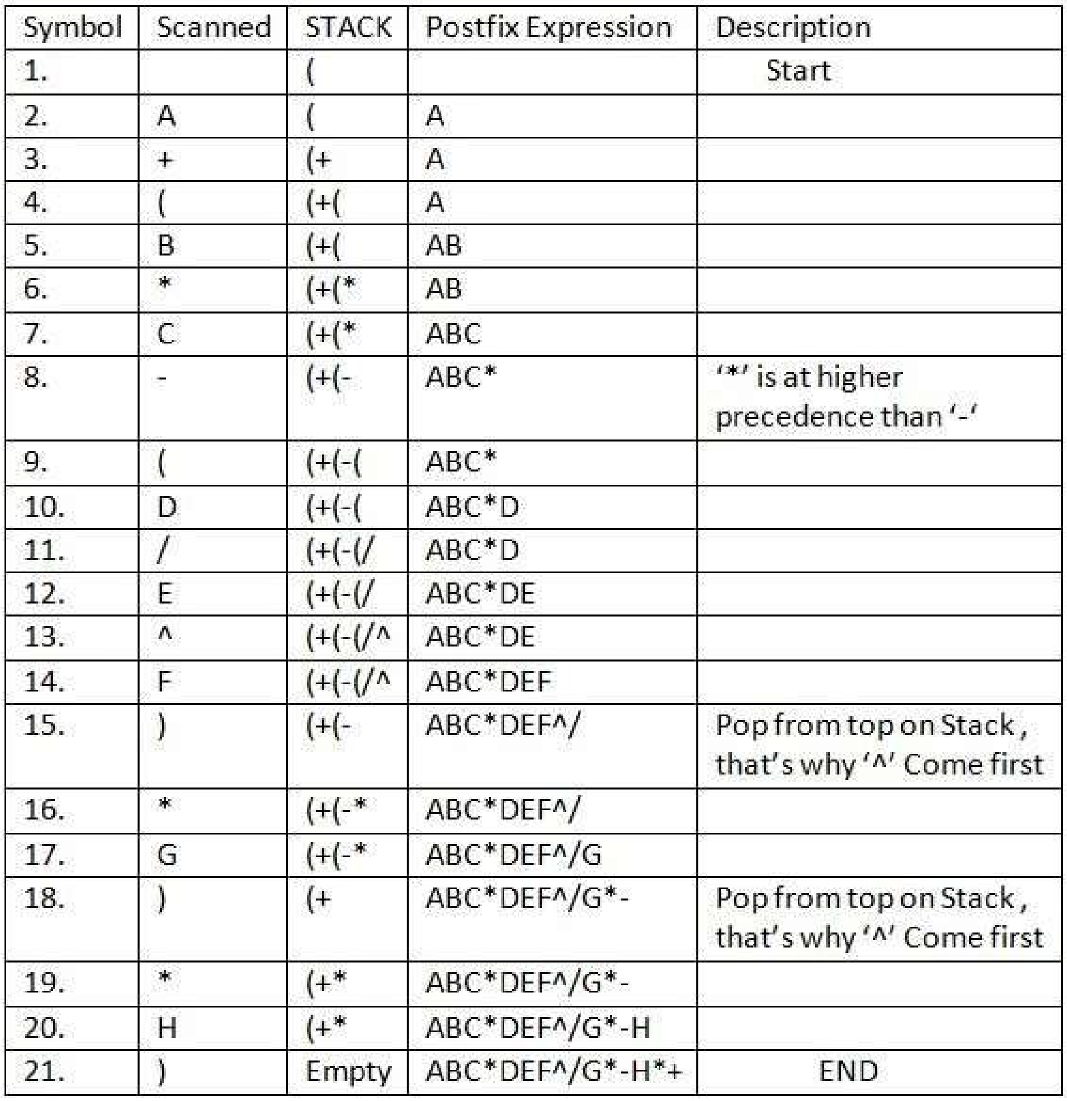
Infix expression: The expression of the form a op b. When an operator in-between every pair of operands.

Postfix expression: The expression of the form **a b op.** When an operator is followed for every pair of operands.

EVALUATION OF POSTFIX EXPRESSION USING STACK:

1. initialize empty stack with top=-1.
2. scan the given postfix expression from left to right till the last character of the expression.
3. if the character is operand than push it into the stack.
4. if the operator is unary operator then pop up one operand from the stack.
5. if the operator is binary operator then pop two operator from the stack and perform the operation and store the result in the stack.
6. repeat 3&4 till the last character of the postfix expression .

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where ^ is an exponential operator.



Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

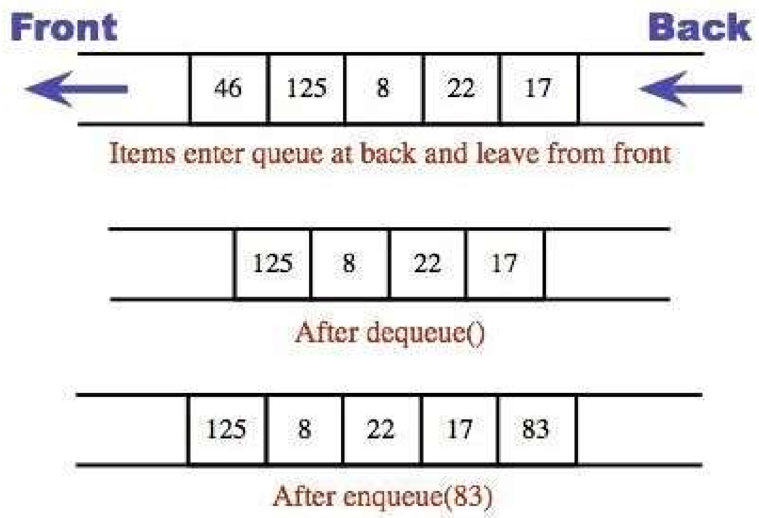
Resultant Postfix Expression: ABC*DEF^/G*-H*+

5. Show how the fundamental operation of a queue can be implemented.

QUEUE:
A queue is an ordered collection of items from which items may be deleted at one end called the front and the items may be inserted at the other end called rear of the queue.

PRINCIPLE:
The first element inserted into a queue is the first element to be removed. Queue is called First In First Out (FIFO) list.

BASIC OPERATIONS INVOLVED IN A QUEUE:



1. Create a queue
2. Check whether a queue is empty or full
3. Add an item at the rear end
4. Remove an item at the front end
5. Read the front of the queue
6. Print the entire queue

INSERTION OPERATION

An attempt to push an item onto a queue, when the queue is full, causes an overflow.

1. Check whether the queue is full before attempting to insert another element.
2. Increment the rear pointer & 3. Insert the element at the rear pointer of the queue.

ALGORITHM:

Rear – Rear end pointer, Q – Queue, N – Total number of elements & Item – The element to be inserted

```
1. if(Rear=N)
   [Overflow?] Then
   Call QUEUE_FULL
   Return
2. Rear<-Rear+1 [Increment rear pointer]
3. Q[Rear]<-Item [Insert element]
 End INSERT
```

DELETION OPERATION:

An attempt to remove an element from the queue when the queue is empty causes an underflow.

Deletion operation involves:

1. Check whether the queue is empty.
2. Increment the front pointer.
3. Remove the element.

ALGORITHM:

Q – Queue, Front – Front end pointer , Rear – Rear end pointer & Item – The element to be deleted.

```
1.if(Front=Rear) [Underflow?]
   Then Call QUEUE_EMPTY
2. Front<-Front+1   [Incrementation]
3. Item<-Q [Front]  [Delete element]
```

Thus queue is a dynamic structure that is constantly allowed to grow and shrink and thus changes its size, when implemented using linked list.

6. Explain the different types of Queue.

QUEUE:

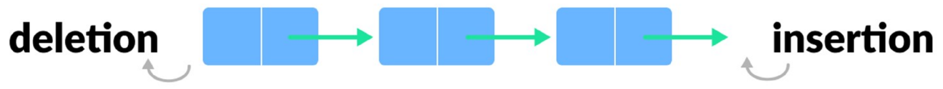
A queue is an ordered collection of items from which items may be deleted at one end called the front and the items may be inserted at the other end called rear of the queue.

There are four different types of queue in data structure.

- **Simple queue**
- **Circular queue**
- **Priority queue**
- **Double Ended queue or Dequeue**

Simple Queue:

In a simple queue, insertion takes palce at the rear and removal occurs at the front. It strictly follows FIFO rule.



Queue Specifications

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations.

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue

- **IsEmpty**: Check if the queue is empty.
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

Circular Queue:

In a circular queue, the last element points to the first element making a circular link.

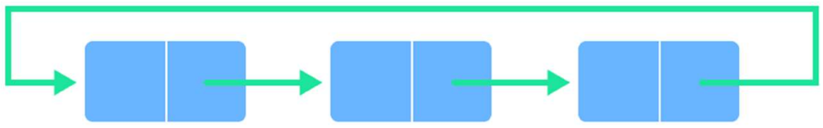


Fig: Circular Queue

The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty then, an element can be inserted in the first position. This action is not possible in a simple queue.

Priority Queue:

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

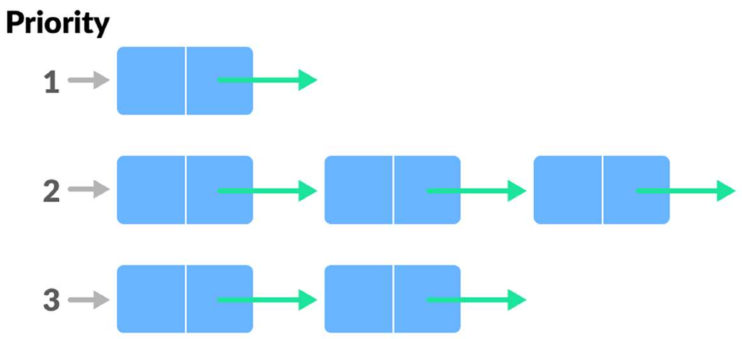


Fig: Priority Queue

Insertion occurs based on the arrival of the values and removal occurs based on priority.

Double Ended queue or Dequeue:

Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).

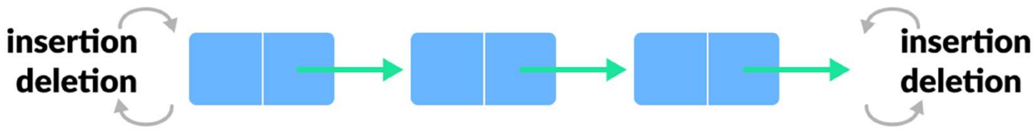
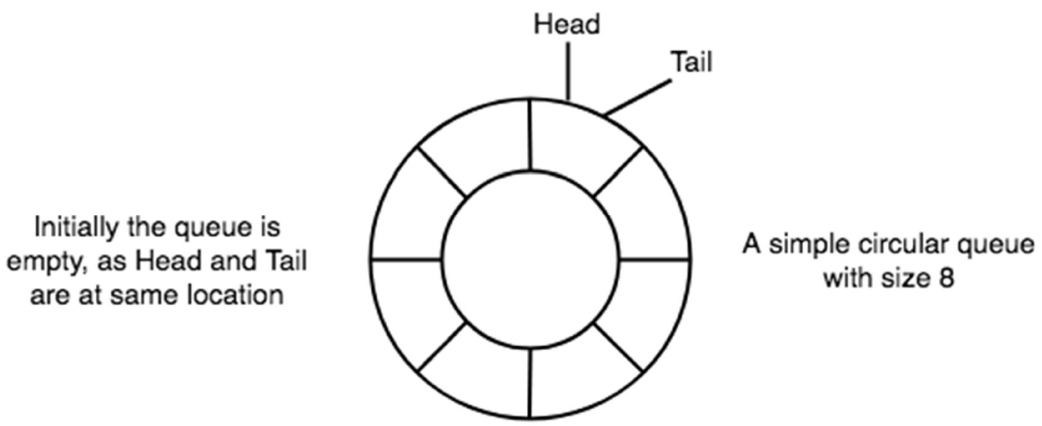


Fig: Dequeue

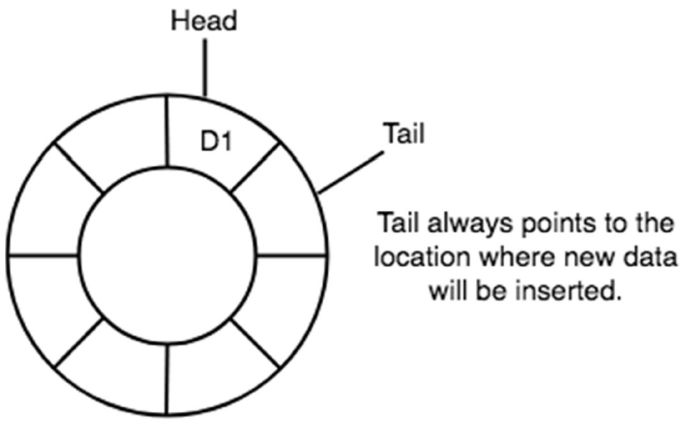
7. Explain Circular Queue with an example.

Circular Queue is also a linear data structure, which follows the principle of **FIFO** (First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

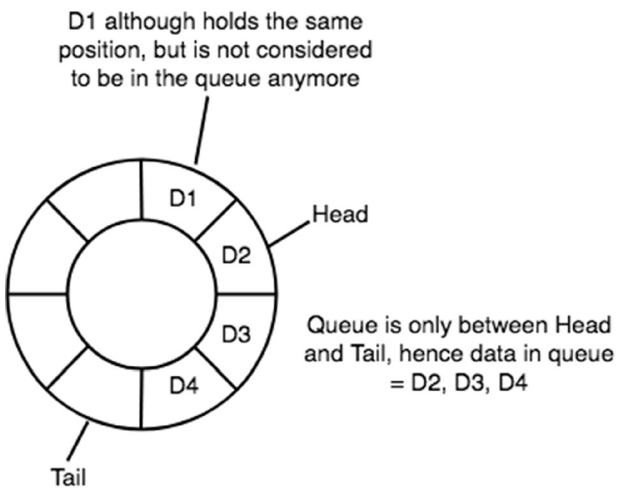
- In case of a circular queue, `head` pointer will always point to the front of the queue, and `tail` pointer will always point to the end of the queue.
- Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.


Head
Tail
Initially the queue is empty, as Head and Tail are at same location
A simple circular queue with size 8

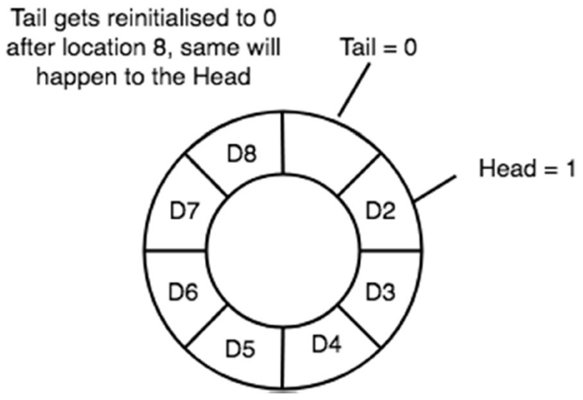
- New data is always added to the location pointed by the `tail` pointer, and once the data is added, `tail` pointer is incremented to point to the next available location.


Head
D1
Tail
Tail always points to the location where new data will be inserted.

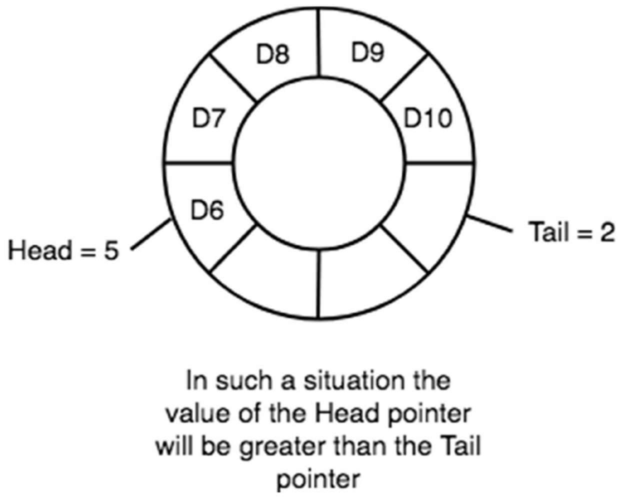
- In a circular queue, data is not actually removed from the queue. Only the `head` pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between `head` and `tail`, hence the data left outside is not a part of the queue anymore, hence removed.


D1 although holds the same position, but is not considered to be in the queue anymore
D1
Head
D2
D3
Queue is only between Head and Tail, hence data in queue = D2, D3, D4
D4
Tail

- The `head` and the `tail` pointer will get reinitialised to **0** every time they reach the end of the queue.


Tail gets reinitialised to 0 after location 8, same will happen to the Head
Tail = 0
D8
Head = 1
D7
D2
D6
D3
D5
D4

- Also, the `head` and the `tail` pointers can cross each other. In other words, `head` pointer can be greater than the `tail`. Sounds odd? This will happen when we dequeue the queue a couple of times and the `tail` pointer gets reinitialised upon reaching the end of the queue.


D8
D9
D7
D10
D6
Head = 5
Tail = 2
In such a situation the value of the Head pointer will be greater than the Tail pointer

Implementation:

Below we have the implementation of a circular queue:

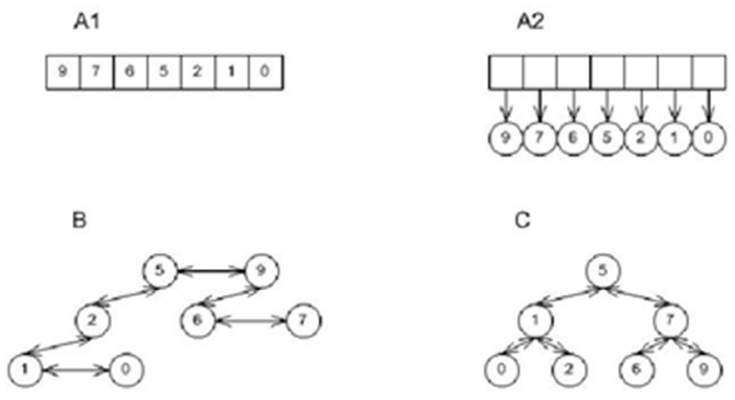
1. Initialize the queue, with size of the queue defined (maxSize), and head and tail pointers.
2. enqueue: Check if the number of elements is equal to maxSize - 1:
    - If **Yes**, then return **Queue is full**.
    - If **No**, then add the new data element to the location of tail pointer and increment the tail pointer.
3. dequeue: Check if the number of elements in the queue is zero:
    - If **Yes**, then return **Queue is empty**.
    - If **No**, then increment the head pointer.
4. Finding the size:
    - If, **tail >= head**, size = (tail - head) + 1
    - But if, **head > tail**, then size = maxSize - (head - tail) + 1

8. Explain Priority Queue with an example.

PRIORITY QUEUES

Priority queues are a kind of queue in which the elements are dequeued in priority order.

A priority queue is a collection of elements where each element has an associated priority. Elements are added and removed from the list in a manner such that the element with the highest (or lowest) priority is always the next to be removed. When using a heap mechanism, a priority queue is quite efficient for this type of operation.



- Each element has a priority, an element of a totally ordered set (usually a number).
- More important things come out first, even if they were added later.
- Three types of priority. Low priority [10], Normal Priority [5] and High Priority [1].
- There is no (fast) operation to find out whether an arbitrary element is in the queue.

ALGORITHM:

Priority Queue - Algorithms - Adjust

```
Adjust(i)
left = 2i, right = 2i + 1

if left <= H.size and H[left] > H[i]
```

```
     then largest = left
     else largest = i
if right <= H.size and H[right] > H[largest]
     then largest = right
if largest != i
     then swap H[largest] with H[i]
          Adjust(largest)
```

Explanation:

Adjust works recursively to guarantee the heap property. It compares the current node with its children finding which, if either, has a greater priority. If one of them does, it will swap array locations with the largest child. Adjust is then run again on the current node in its new location.

Priority Queue - Algorithms - Insert

```
Insert(Key)
        H.size = H.size + 1
        i = H.size
        while i > 1 and H[i/2] < Key
                H[i] = H[i/2]
                i = i/2
        end while
        H[i] = key
```

Explanation

The insert algorithm works by first inserting the new element (key) at the end of the array. This element is then compared with its parent for highest priority. If the parent has a lower priority, the two elements are swapped. This process is repeated until the new element has found its place.

Priority Queue - Algorithms - Extract

```
Extract()
    Max = H[1] H[1] = H[H.size]
    H.size = H.size – 1
    Adjust(1)  Return
    Max
```

Explanation

Extract works by removing and returning the first array element, the one of highest priority, and then promoting the last array element to the first. Adjust is then run on the first element so that the heap property is maintained.

Run Time Complexity

Θ(lg *n*) time for Insert and worst case for Extract (where *n* is number of elements)

Θ(lg *n*) average time for Insert

Can construct a Heap from a list in Θ(lg *n*) where a Binary Search Tree takes Θ(*n* lg *n*)

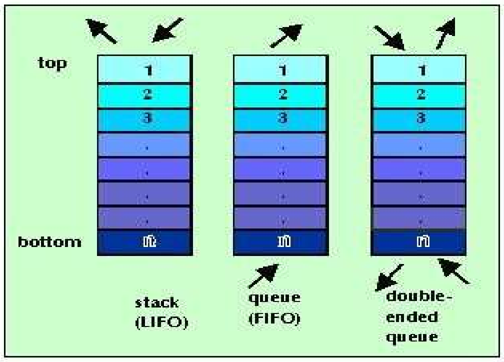
Space Requirements

All operations are done in place - space requirement at any given time is the number of elements, *n*.

9. **Write an algorithm for a doubly ended queue for insertion and deletion.**

Double Ended Queue

A **deque** (short for *double-ended queue*) is an abstract data structure for which elements can be added to or removed from the front or back(both end). This differs from a normal queue, where elements can only be added to one end and removed from the other. Both queues and stacks can be considered specializations of deques, and can be implemented using deques.



double-ended queue

INSERT NEW ELEMENT INTO DEQUEUE:

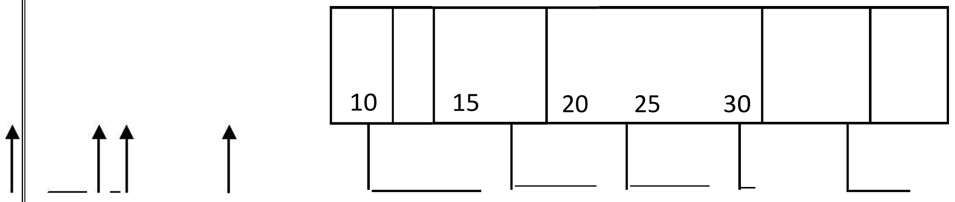
AT THE END OF THE DEQUEUE:

- First check the dequeue is full or not.
- We have to check whether the element is first to be added or inserted if it is true assign front and rear is 0.
- Insert a new element

AT THE BEGINNING OF THE DEQUEUE:
First check the dequeue is full or not.

➢ We have to check whether the element is first to be inserted if it is true then perform the following steps

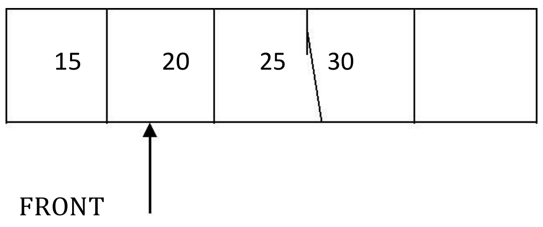
1. shift the elements from left to right
2. so that we need the number of elements present in the dequeue ,the position of the last element ie the position of the rear
3. now insert any element in zeroth position



DELETING THE ELEMENT FROM DEQUEUE:

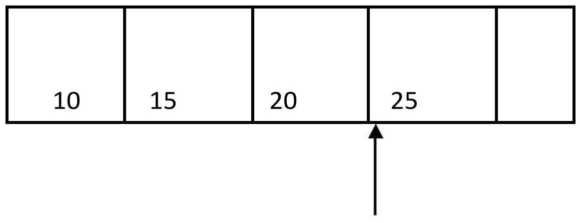
AT THE BEGINNING OF DEQUEUE:

- delete the first element from the front position of the dequeue
- the index of next elem
- .ent is stored in front pointer.
- Increment the front pointer by one .



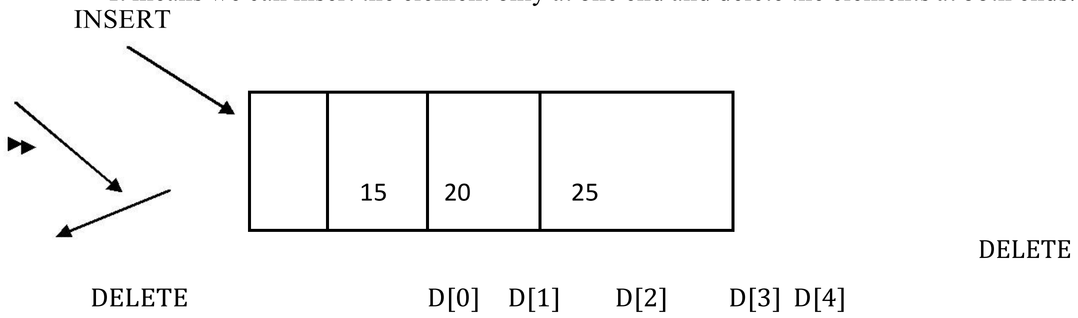
AT THE END OF DEQUEUE:

- □ The last element is deleted .
- □ The index of next element is stored in rear pointer
- ➢ Decrement the rear pointer by one.



INPUT RESTRICTED DEQUEUE: REAR

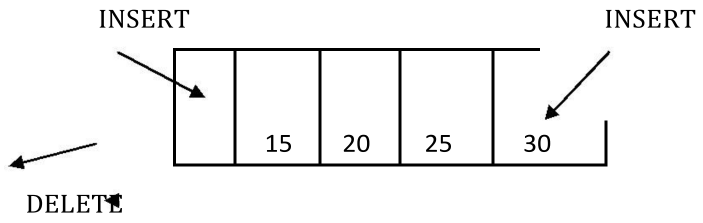
It means we can insert the element only at one end and delete the elements at both ends.



➢ The above diagram shows the input restricted dequeue.

OUTPUT RESTRICTED DEQUEUE:

It means we can insert the elements in both ends and delete the elements at only one end.



The above diagram shows the output restricted dequeue.

10. Write an algorithm for peek(), isFull() and isEmpty() operations of queue.

The supportive functions are required to make the a queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
Being procedure isfull
        If rear equals to MAZSIZE
                return true
        else
                return false
        endif
end procedure
```

isempty()

Algorithm of isempty() function −

Algorithm

```
Being procedure isfull
        If front is less than MIN or front is greater than rear
                return true
        else
                return false
        endif
end procedure
```

11. Difference between Stack and queue.

STACKS	QUEUES
Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.

STACKS	QUEUES
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.
Stack is used in solving problems works on recursion.	Queue is used in solving problems having sequential processing.

12. Comparison between Linear Queue and Circular Queue.

LINEAR QUEUE	CIRCULAR QUEUE
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.
Tasks are executed in order they were placed before (FIFO).	Order of executing a task may change.
The new element is added from the rear end and removed from the front.	Insertion and deletion can be done at any position.

LINEAR QUEUE	CIRCULAR QUEUE
The linear queue is an ordered list in which data elements are organized in the sequential order	In contrast, circular queue stores the data in the circular fashion.
Linear queue follows the FIFO order for executing the task (the element added in the first position is going to be deleted in the first position)	Conversely, in the circular queue, the order of operations performed on an element may change.
Inefficient	Works better than the linear queue.

13. Write note on Applications of Stack.

The Stack is Last In First Out (LIFO) data structure. This data structure has some important applications in different aspect. These are like below −

- Expression Handling −
    - Infix to Postfix or Infix to Prefix Conversion −

      The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions. These expressions are not so much familiar to the infix expression, but they have some great advantages also. We do not need to maintain operator ordering, and parenthesis.
    - Postfix or Prefix Evaluation −

      After converting into prefix or postfix notations, we have to evaluate the expression to get the result. For that purpose, also we need the help of stack data structure.
- Backtracking Procedure −

  Backtracking is one of the algorithm designing technique. For that purpose, we dive into some way, if that way is not efficient, we come back to the previous state and go into some other paths. To get back from current state, we need to store the previous state. For that purpose, we need stack. Some examples of backtracking is finding the solution for Knight Tour problem or N-Queen Problem etc.

- Another great use of stack is during the function call and return process. When we call a function from one other function, that function call statement may not be the first statement. After calling the function, we also have to come back from the function area to the place, where we have left our control. So we want to resume our task, not restart. For that reason, we store the address of the program counter into the stack, then go to the function body to execute it. After completion of the execution, it pops out the address from stack and assign it into the program counter to resume the task again.

14. Write an example program for Stack using array.

```
// C program for array implementation of stack
#include <limits.h>
#include <stdio.h>
#include  <stdlib.h>

// A structure to represent a stack

 struct Stack {
    int top;
    unsigned capacity;
    int* array;
 };

 // function to create a stack of given capacity. It initializes size of
 // stack as 0
 struct Stack* createStack(unsigned capacity)
 {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
 }

 // Stack is full when top is equal to the last index
 int isFull(struct Stack* stack)
 {
    return stack->top == stack->capacity - 1;
 }

 // Stack is empty when top is equal to -1
 int isEmpty(struct Stack* stack)
 {
    return stack->top == -1;
 }

 // Function to add an item to stack.  It increases top by 1
 void push(struct Stack* stack, int item)
 {
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
 }

 // Function to remove an item from stack.  It decreases top by 1
 int pop(struct Stack* stack)
 {
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
 }

 // Function to return the top from stack without removing it
 int peek(struct Stack* stack)
 {
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));

    return 0;
}
```

Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 popped from stack
```

15. What is Stack? Write the time complexities of stack operations and applications of Stack.

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Applications of stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver
- In Graph Algorithms like Topological Sorting and Strongly Connected Components

10 MARKS

1. **What is stack? Write the algorithm with an example program.**

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Operations:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

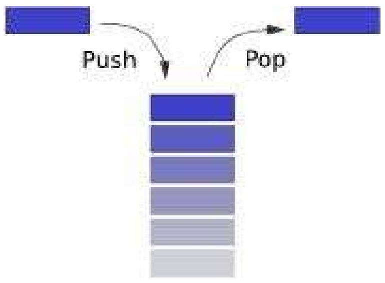


Fig: Simple representation of a Stack

Example for Stack:

Converting a decimal number into a binary number:

To solve this problem, we use a stack. We make use of the *LIFO* property of the stack. Initially we *push* the binary digit formed into the stack, instead of printing it directly. After the entire digit has been converted into the binary form, we *pop* one digit at a time from the stack and print it. Therefore, we get the decimal number is converted into its proper binary form.

Algorithm:

```
      1. Create a stack
      2. Enter a decimal number which has to be converted into its
equivalent binary form.
      3. iteration1 (while number > 0)
            3.1 digit = number % 2
            3.2 Push digit into the stack
            3.3 If the stack is full
                  3.3.1 Print an error
                  3.3.2 Stop the algorithm
            3.4 End the if condition
            3.5 Divide the number by
      4. End iteration1
      5. iteration2 (while stack is not empty
            5.1 Pop digit from the stack
            5.2 Print the digit
      6. End iteration2
      7. STOP
```

```
// C program for array implementation of stack
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a stack

 struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
    return stack->top == stack->capacity - 1;
}


// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}

// Function to add an item to stack.  It increases top by 1
void push(struct Stack* stack, int item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

// Driver program to test above functions
int main()
{
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);
    printf("%d popped from stack\n", pop(stack));

    return 0;
}
```

Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 popped from stack
```

2. Explain both push and pop operation on stack.

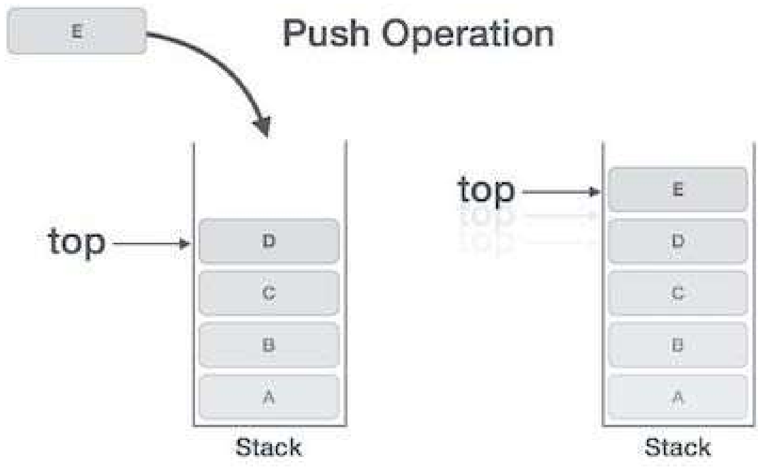
Stack operations may involve initializing the stack, using it and then de-initalizing it. Appart from these basic stuffs, a stack is used for the following two primary operations –

- **push() –** Pushing (storing) an element on the stack.
- **pop() –** Removing (accessing) an element from the stack.

Push Operation:

The process of putting a new data element onto stack is known as Push Operation. Push operation involves a series of steps –

Step 1 - Checks if the stack is full.
Step 2 - If the stack is full, produces an error and exit.
Step 3 - If the stack is not full, increments **top** to point next empty space.
Step 4 - Adds data element to the stack location, where top is pointing.
Step 5 - Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation:

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

    if stack is full
        return null
    endif

    top ← top + 1
    stack[top] ← data

end procedure
```

Example:

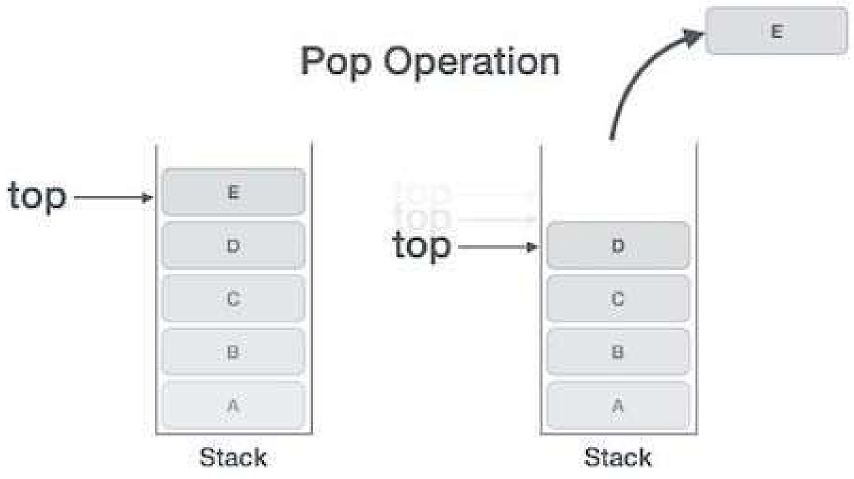
```
void push(int data) {
      if(!isFull()){
            top += 1;
            stack[top] = data;
      } else{
            printf("Could not insert, Stack  is full. \n");
      }
}
```

POP Operation:

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation:

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack
      if stack is empty
            return null
      end if

      data ← stack[top]
```

```
        top ← top - 1
        return data
```

Implementation of this algorithm in C, is as follows –

Example:

```
int pop(int data){
     if(!isempty()){
          data = stack[top];
          top = top - 1;
          return data;
     }else{
          printf("Could not retrieve, Stack is empty.\n");
     } }
```

3. **How the fundamental operations of a queue can be implemented and explain the types of queue in detail.**

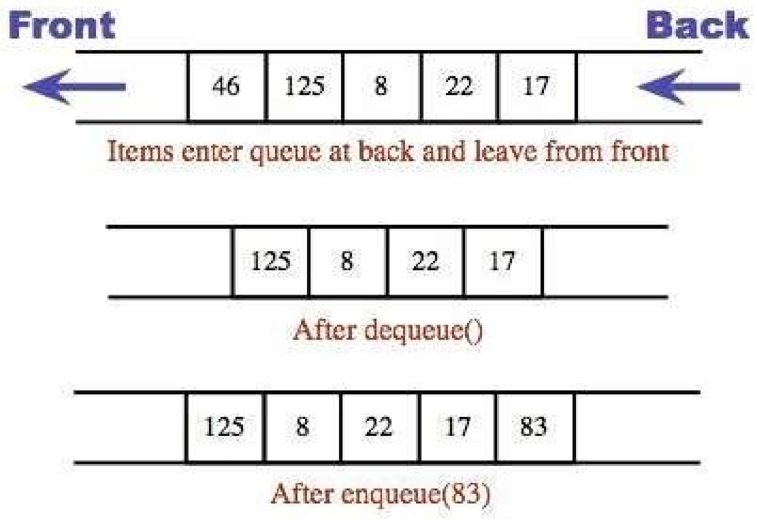
QUEUE:

A queue is an ordered collection of items from which items may be deleted at one end called the front and the items may be inserted at the other end called rear of the queue.

PRINCIPLE:

The first element inserted into a queue is the first element to be removed. Queue is called First In First Out (FIFO) list.

BASIC OPERATIONS INVOLVED IN A QUEUE:



1. Create a queue
2. Check whether a queue is empty or full
3. Add an item at the rear end
4. Remove an item at the front end
5. Read the front of the queue

6. Print the entire queue

INSERTION OPERATION

An attempt to push an item onto a queue, when the queue is full, causes an overflow.

3. Check whether the queue is full before attempting to insert another element.
4. Increment the rear pointer & 3. Insert the element at the rear pointer of the queue.

ALGORITHM:

Rear – Rear end pointer, Q – Queue, N – Total number of elements & Item – The element to be inserted

```
4. if(Rear=N)
   [Overflow?] Then
   Call QUEUE_FULL
   Return
5. Rear<-Rear+1 [Increment rear pointer]
6. Q[Rear]<-Item [Insert element]
  End INSERT
```

DELETION OPERATION:

An attempt to remove an element from the queue when the queue is empty causes an underflow.

Deletion operation involves:

4. Check whether the queue is empty.
5. Increment the front pointer.
6. Remove the element.

ALGORITHM:

Q – Queue, Front – Front end pointer , Rear – Rear end pointer & Item – The element to be deleted.

```
1.if(Front=Rear) [Underflow?]
   Then Call QUEUE_EMPTY
4. Front<-Front+1   [Incrementation]
5. Item<-Q [Front]  [Delete element]
```

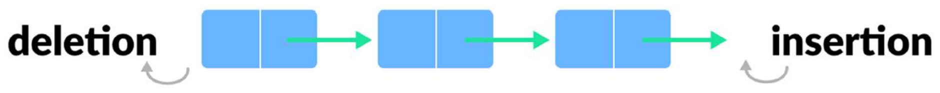
Thus queue is a dynamic structure that is constantly allowed to grow and shrink and thus changes its size, when implemented using linked list.

There are four different types of queue in data structure.

- **Simple queue**
- **Circular queue**
- **Priority queue**
- **Double Ended queue or Dequeue**

<u>Simple Queue:</u>

In a simple queue, insertion takes palce at the rear and removal occurs at the front. It strictly follows FIFO rule.



Queue Specifications

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations.

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty.
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

Circular Queue:

In a circular queue, the last element points to the first element making a circular link.

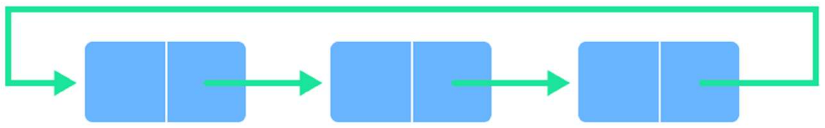


Fig: Circular Queue

The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty then, an element can be inserted in the first position. This action is not possible in a simple queue.

Priority Queue:

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

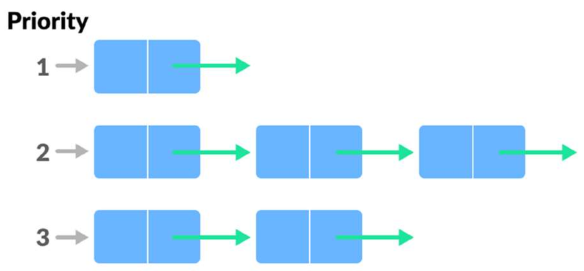


Fig: Priority Queue

Insertion occurs based on the arrival of the values and removal occurs based on priority.

Double Ended queue or Dequeue:

Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).

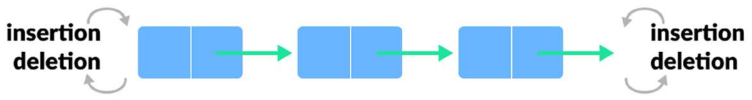
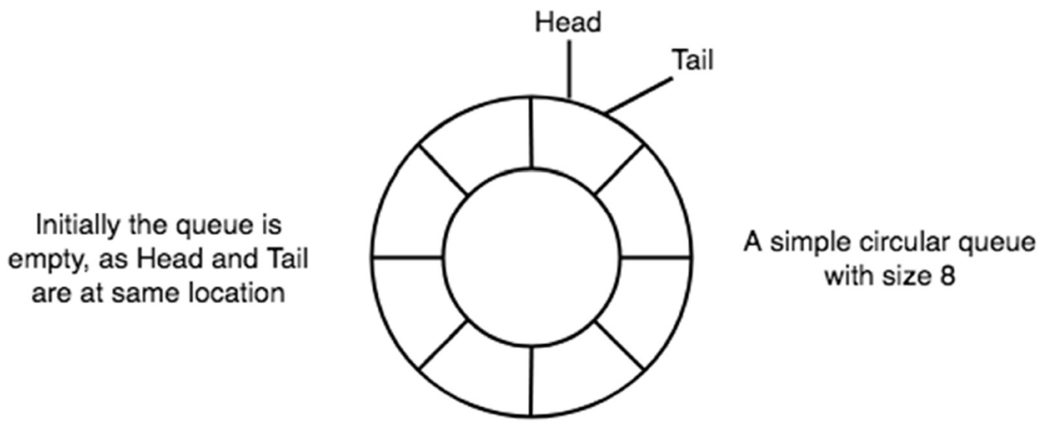


Fig: Dequeue

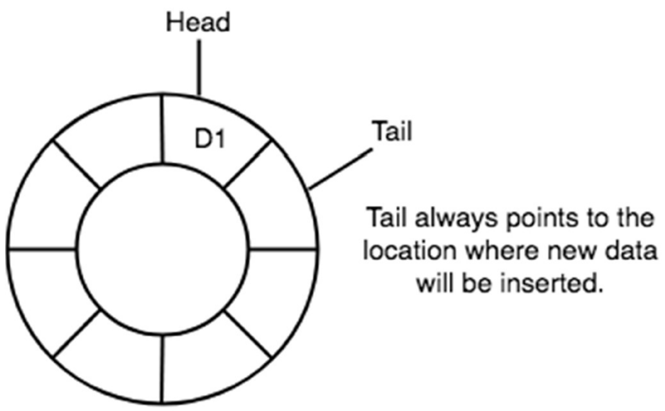
4. Explain Circular queue and Priority queue in detail.

Circular Queue is also a linear data structure, which follows the principle of **FIFO** (First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

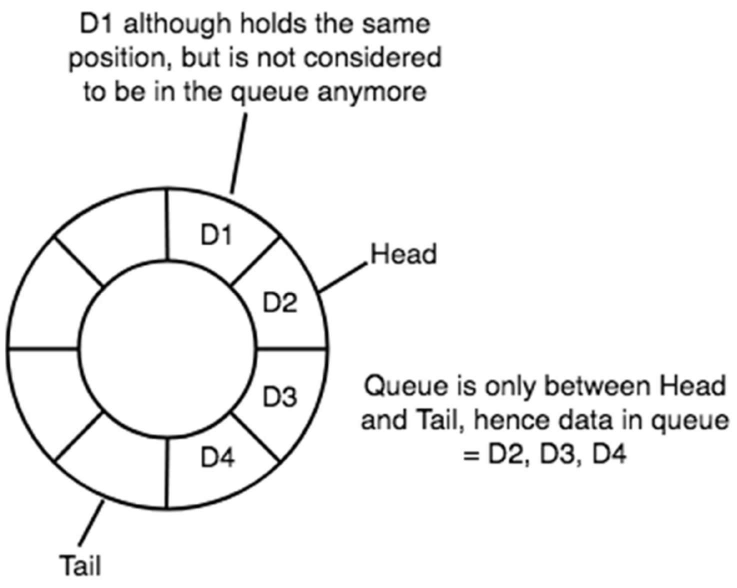
- In case of a circular queue, `head` pointer will always point to the front of the queue, and `tail` pointer will always point to the end of the queue.
- Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.


Head
Tail
Initially the queue is empty, as Head and Tail are at same location
A simple circular queue with size 8

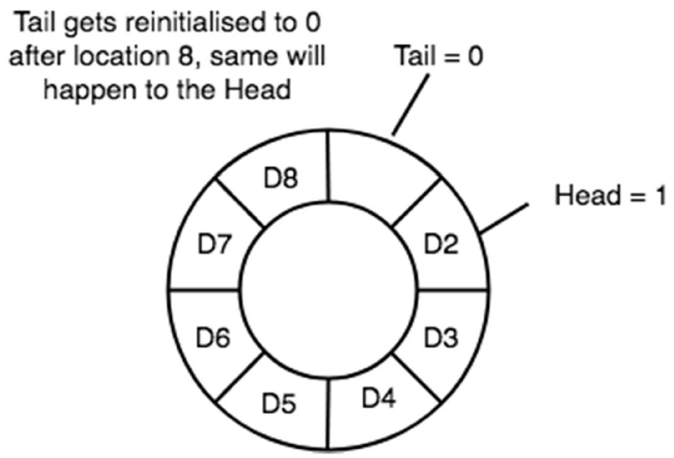
- New data is always added to the location pointed by the `tail` pointer, and once the data is added, `tail` pointer is incremented to point to the next available location.


Head
D1
Tail
Tail always points to the location where new data will be inserted.

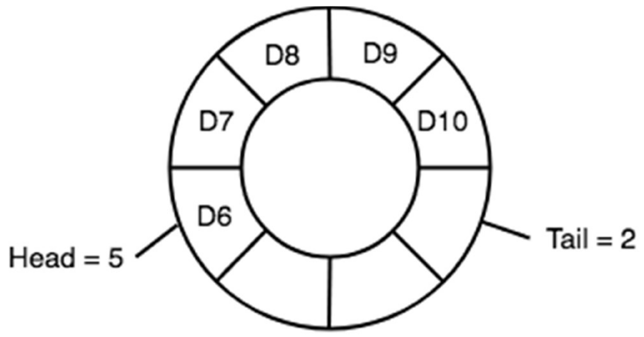
- In a circular queue, data is not actually removed from the queue. Only the `head` pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between `head` and `tail`, hence the data left outside is not a part of the queue anymore, hence removed.


D1 although holds the same position, but is not considered to be in the queue anymore
D1
D2
D3
D4
Head
Tail
Queue is only between Head and Tail, hence data in queue = D2, D3, D4

- The `head` and the `tail` pointer will get reinitialised to **0** every time they reach the end of the queue.


Tail gets reinitialised to 0 after location 8, same will happen to the Head
Tail = 0
Head = 1
D8
D2
D7
D3
D6
D5
D4

- Also, the `head` and the `tail` pointers can cross each other. In other words, `head` pointer can be greater than the `tail`. Sounds odd? This will happen when we dequeue the queue a couple of times and the `tail` pointer gets reinitialised upon reaching the end of the queue.



In such a situation the value of the Head pointer will be greater than the Tail pointer

Implementation:

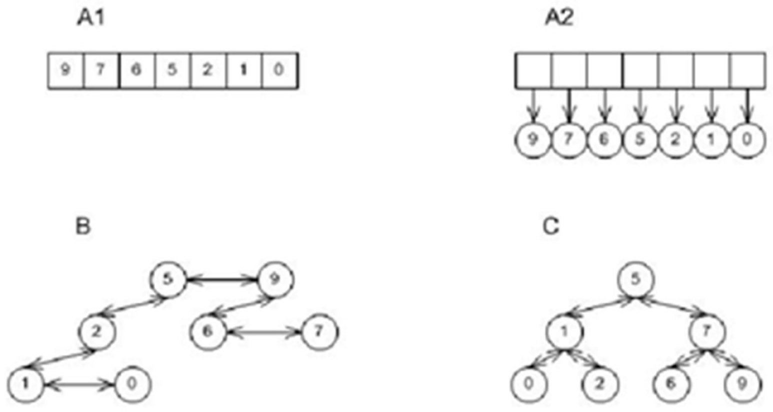
Below we have the implementation of a circular queue:

5. Initialize the queue, with size of the queue defined (maxSize), and head and tail pointers.
6. enqueue: Check if the number of elements is equal to maxSize - 1:
    - If **Yes**, then return **Queue is full**.
    - If **No**, then add the new data element to the location of tail pointer and increment the tail pointer.
7. dequeue: Check if the number of elements in the queue is zero:
    - If **Yes**, then return **Queue is empty**.
    - If **No**, then increment the head pointer.
8. Finding the size:
    - If, **tail >= head**, size = (tail - head) + 1
    - But if, **head > tail**, then size = maxSize - (head - tail) + 1

PRIORITY QUEUES

Priority queues are a kind of queue in which the elements are dequeued in priority order.

A priority queue is a collection of elements where each element has an associated priority. Elements are added and removed from the list in a manner such that the element with the highest (or lowest) priority is always the next to be removed. When using a heap mechanism, a priority queue is quite efficient for this type of operation.



- Each element has a priority, an element of a totally ordered set (usually a number).
- More important things come out first, even if they were added later.
- Three types of priority. Low priority [10], Normal Priority [5] and High Priority [1].
- There is no (fast) operation to find out whether an arbitrary element is in the queue.

ALGORITHM:

Priority Queue - Algorithms - Adjust

```
Adjust(i)
left = 2i, right = 2i + 1

if left <= H.size and H[left] > H[i]
    then largest = left
    else largest = i
if right <= H.size and H[right] > H[largest]
    then largest = right
if largest != i
    then swap H[largest] with H[i]
        Adjust(largest)
```

Explanation:

Adjust works recursively to guarantee the heap property. It compares the current node with its children finding which, if either, has a greater priority. If one of them does, it will swap array locations with the largest child. Adjust is then run again on the current node in its new location.

Priority Queue - Algorithms - Insert

```
Insert(Key)
        H.size = H.size + 1
        i = H.size
        while i > 1 and H[i/2] < Key
                H[i] = H[i/2]
                i = i/2
        end while
        H[i] = key
```

Explanation

The insert algorithm works by first inserting the new element (key) at the end of the array. This element is then compared with its parent for highest priority. If the parent has a lower priority, the two elements are swapped. This process is repeated until the new element has found its place.

Priority Queue - Algorithms - Extract

```
Extract()
    Max = H[1] H[1] = H[H.size]
```

```
H.size = H.size – 1
Adjust(1)  Return
Max
```

Explanation

Extract works by removing and returning the first array element, the one of highest priority, and then promoting the last array element to the first. Adjust is then run on the first element so that the heap property is maintained.

Run Time Complexity

Θ(lg n) time for Insert and worst case for Extract (where n is number of elements)

Θ(lg n) average time for Insert

Can construct a Heap from a list in Θ(lg n) where a Binary Search Tree takes Θ(n lg n)

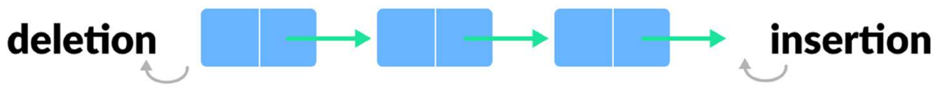
Space Requirements

All operations are done in place - space requirement at any given time is the number of elements, n.

5. Explain simple queue and double ended queue in detail.

Simple Queue:

In a simple queue, insertion takes palce at the rear and removal occurs at the front. It strictly follows FIFO rule.



Queue Specifications

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations.

- **Enqueue**: Add an element to the end of the queue
- **Dequeue**: Remove an element from the front of the queue
- **IsEmpty**: Check if the queue is empty.
- **IsFull**: Check if the queue is full
- **Peek**: Get the value of the front of the queue without removing it

Double Ended Queue

A **deque** (short for *double-ended queue*) is an abstract data structure for which elements can be added to or removed from the front or back(both end). This differs from a normal queue, where elements can only be added to one end and removed from the other. Both queues and stacks can be considered specializations of deques, and can be implemented using deques.

INSERT NEW ELEMENT INTO DEQUEUE:

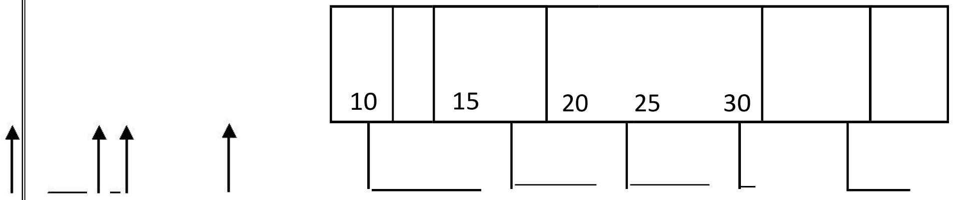
AT THE END OF THE DEQUEUE:

- First check the dequeue is full or not.
- We have to check whether the element is first to be added or inserted if it is true assign front and rear is 0.
- Insert a new element

AT THE BEGINNING OF THE DEQUEUE:
First check the dequeue is full or not.

➢ We have to check whether the element is first to be inserted if it is true then perform the following steps

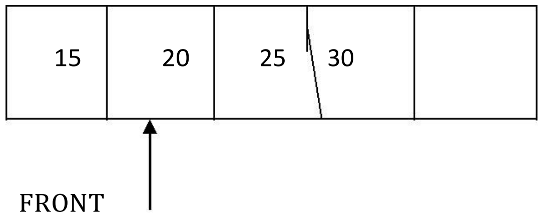
1. shift the elements from left to right
2. so that we need the number of elements present in the dequeue ,the position of the last element ie the position of the rear
3. now insert any element in zeroth position



DELETING THE ELEMENT FROM DEQUEUE:

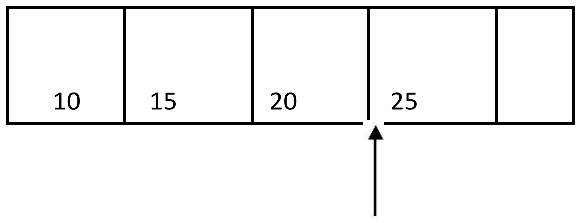
AT THE BEGINNING OF DEQUEUE:

- delete the first element from the front position of the dequeue
- the index of next elem
- .ent is stored in front pointer.
- Increment the front pointer by one .



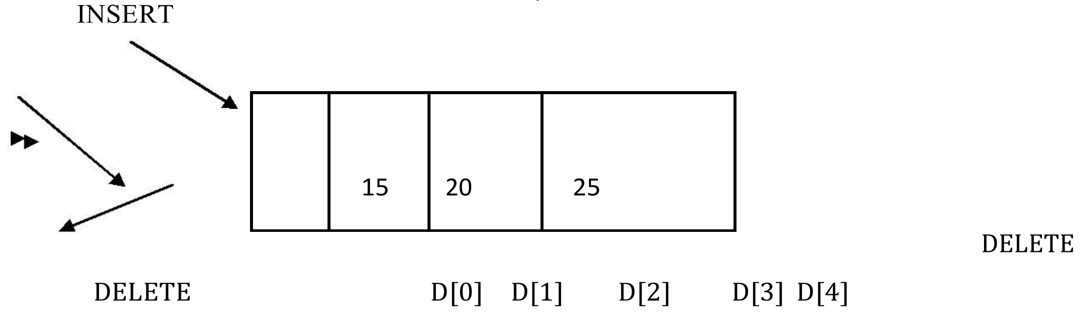
AT THE END OF DEQUEUE:

- The last element is deleted .
- The index of next element is stored in rear pointer
- ➢ Decrement the rear pointer by one.



INPUT RESTRICTED DEQUEUE:

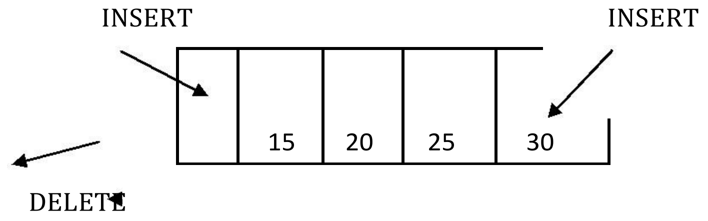
It means we can insert the element only at one end and delete the elements at both ends.



➢ The above diagram shows the input restricted dequeue.

OUTPUT RESTRICTED DEQUEUE:

It means we can insert the elements in both ends and delete the elements at only one end.



The above diagram shows the output restricted dequeue.

6. **What is stack? Write its time complexities along with its application in detail.**

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

The Stack is Last In First Out (LIFO) data structure. This data structure has some important applications in different aspect. These are like below −

- Expression Handling −
  - Infix to Postfix or Infix to Prefix Conversion −

    The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions. These expressions are not so much familiar to the infix expression, but they have some great advantages also. We do not need to maintain operator ordering, and parenthesis.
  - Postfix or Prefix Evaluation −

After converting into prefix or postfix notations, we have to evaluate the expression to get the result. For that purpose, also we need the help of stack data structure.

- Backtracking Procedure −

  Backtracking is one of the algorithm designing technique. For that purpose, we dive into some way, if that way is not efficient, we come back to the previous state and go into some other paths. To get back from current state, we need to store the previous state. For that purpose, we need stack. Some examples of backtracking is finding the solution for Knight Tour problem or N-Queen Problem etc.

- Another great use of stack is during the function call and return process. When we call a function from one other function, that function call statement may not be the first statement. After calling the function, we also have to come back from the function area to the place, where we have left our control. So we want to resume our task, not restart. For that reason, we store the address of the program counter into the stack, then go to the function body to execute it. After completion of the execution, it pops out the address from stack and assign it into the program counter to resume the task again.