Module I – Introduction to Stacks, Queues and Linked Lists

- Introduction to Data Structures: Linear and Non Linear Data Structures, Static and Dynamic Data Structures.
- Concept of Stack and Queue. Array Implementation of Stack and Queue, Circular Queue, Double Ended Queue, Priority Queue.

What is a Stack?

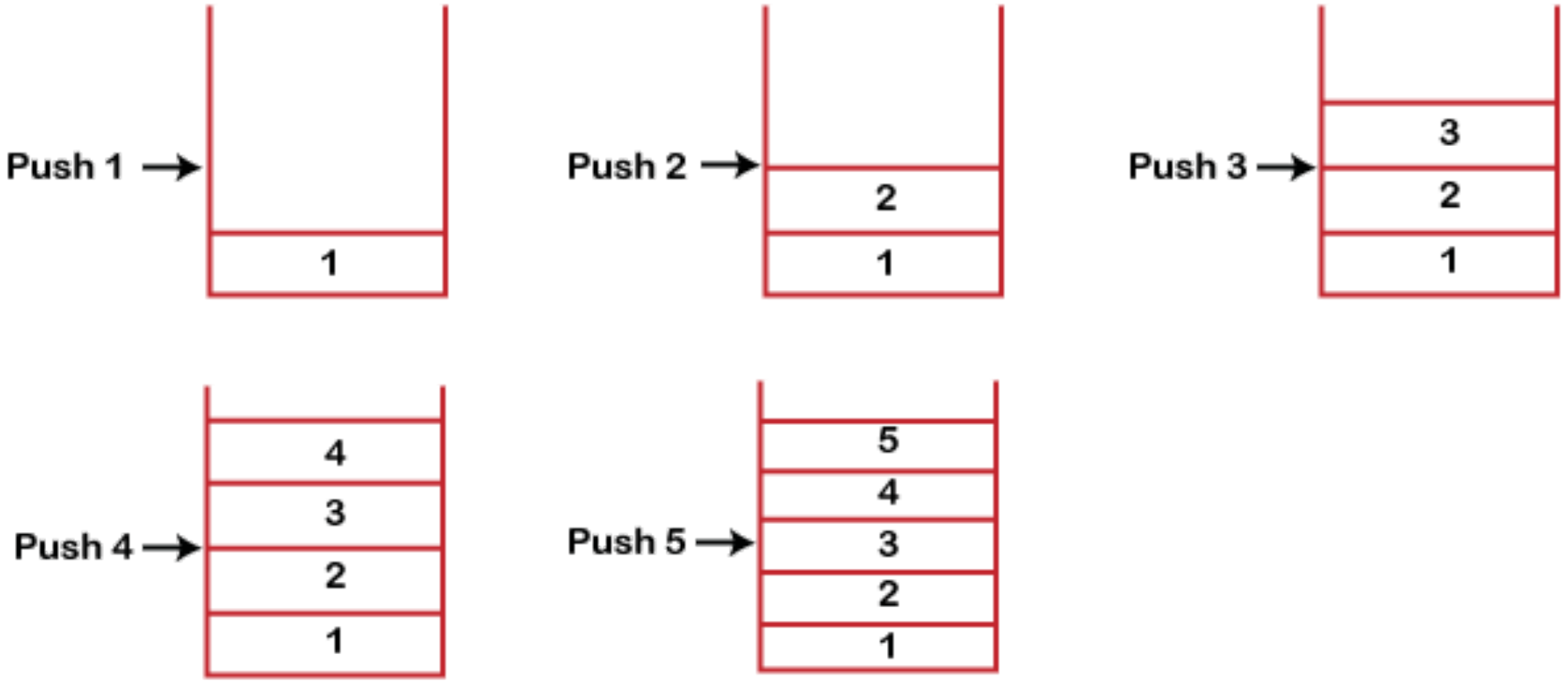
- A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**).
- Stack contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

- Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.
- Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 1 is entered first, so it will be removed only after the deletion of all the other elements.

Functions and Algorithms

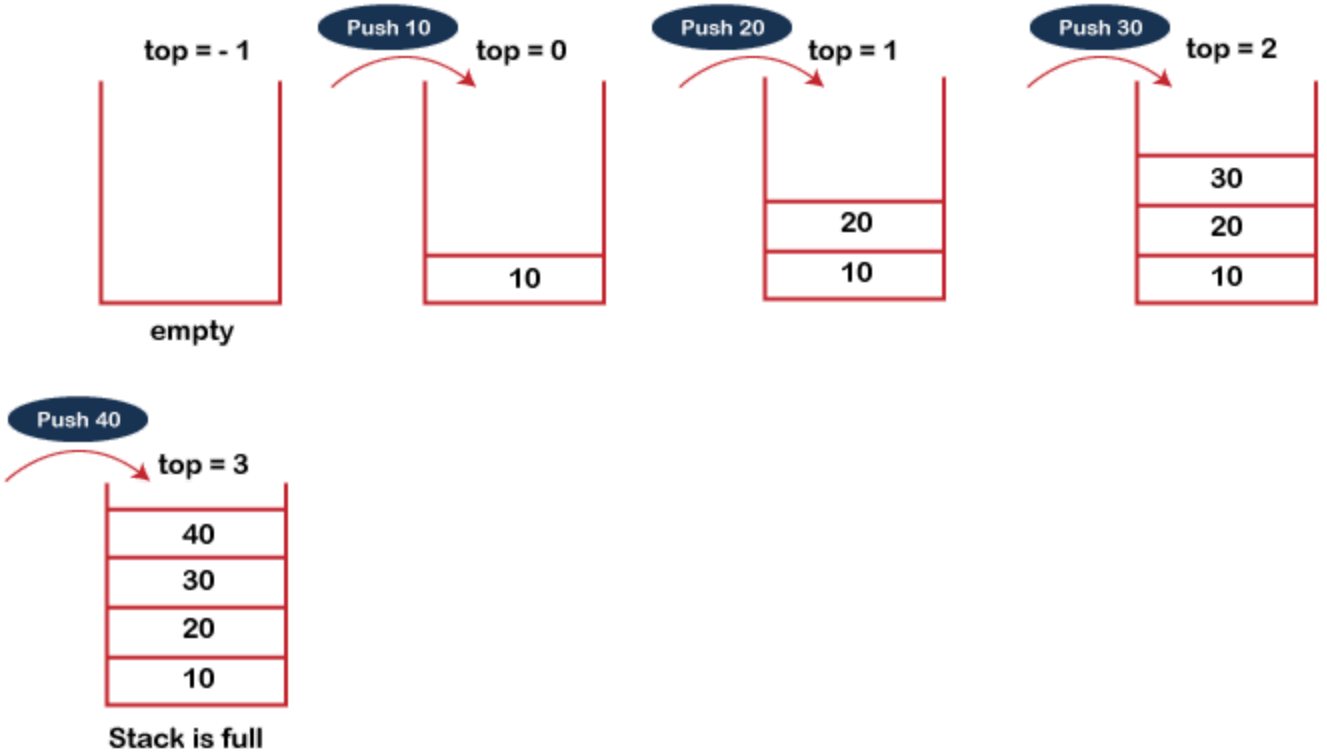
- Initialization of stack.
- Insertion into stack (push operation).
- Deletion from stack (pop operation).
- Check fullness.
- Check emptiness.

Standard Stack Operations

- **The following are some common operations implemented on the stack:**
- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

- **The steps involved in the PUSH operation is given below:**
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.


top = - 1
Push 10
top = 0
10
Push 20
top = 1
20
10
Push 30
top = 2
30
20
10
empty
Push 40
top = 3
40
30
20
10
Stack is full

INIT_STACK (STACK, TOP)

Algorithm to initialize a stack using array.

TOP points to the top-most element of stack.

1) TOP: = 0;
2) Exit

Push operation is used to insert an element into stack.

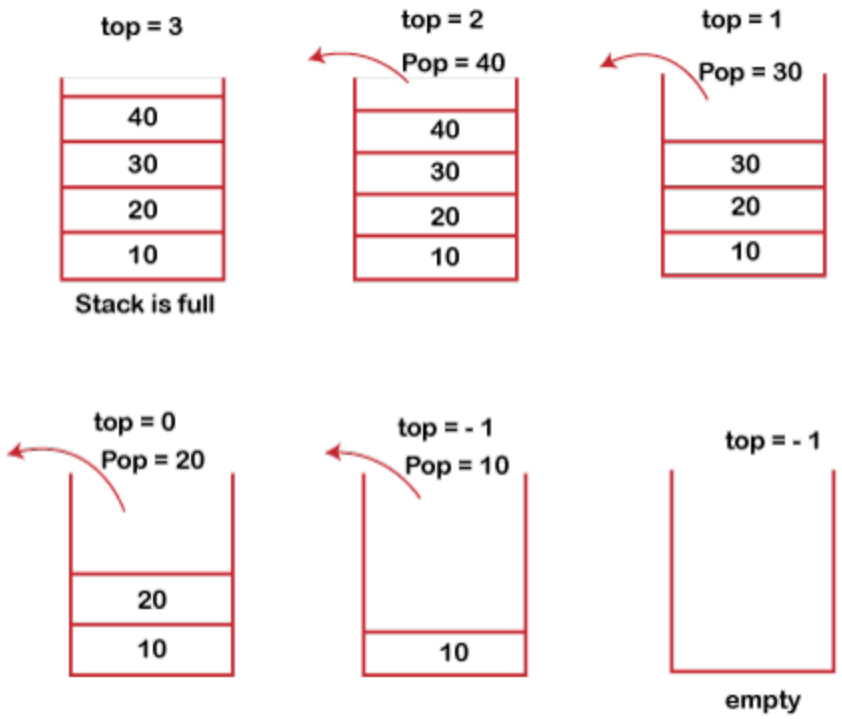
- PUSH_STACK(STACK,TOP,MAX,ITEM)

Algorithm to push an item into stack.

1) IF TOP = MAX then
   Print “Stack is full”;
   Exit;
2) Otherwise
   TOP: = TOP + 1; /*increment TOP*/
   STACK (TOP):= ITEM;
   3) End of IF
   4) Exit

POP operation

- **The steps involved in the POP operation is given below:**
- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



POP_STACK(STACK,TOP,ITEM) Algorithm to pop an element from stack.

1) IF TOP = 0 then Print “Stack is empty”; Exit;
2) Otherwise ITEM: =STACK (TOP); TOP:=TOP – 1;
3) End of IF
4) Exit

IS_FULL

```
IS_FULL(STACK,TOP,MAX,STATUS)
Algorithm to check stack is
full or not. STATUS contains
the result status.
1) IF TOP = MAX then STATUS:=true;
2) Otherwise STATUS:=false;
3) End of IF
4) Exit
```

IS_EMPTY

IS_EMPTY(STACK,TOP,MAX,STATUS) Algorithm to check stack is empty or not. STATUS contains the result status.

1) IF TOP = 0 then STATUS:=true;
2) Otherwise STATUS:=false;
3) End of IF
4) Exit

Applications of Stack

- **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:

```
int main()
{
   cout<<"Hello";
  cout<<"javaTpoint";
}
```

As we know, each program has *an opening* and *closing* braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax error occurs in a program.

- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack. First, we push all the characters of the string in a stack until we reach the null character.
  After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.

- **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below: Infix to prefix, Infix to postfix, Prefix to infix , Prefix to postfix , Postfix to infix
- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Adding an element onto the stack (push operation)

- Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.
- Increment the variable Top so that it can now refere to the next memory location.
- Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.
- Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition

Adding an element onto the stack (push operation)

- **Algorithm:**

```
begin
   if top = n then stack full
   top = top + 1
   stack (top) : = item;
end
```

Time Complexity : o(1)

implementation of push algorithm in C language

```c
void push (int val,int n) //
n is size of the stack
{
    if (top == n )
    printf("\n Overflow");
    else
    {
    top = top +1;
    stack[top] = val;
    }
}
```

Deletion of an element from a stack (Pop operation)

- Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.
- The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
   if top = 0 then stack empt
y;
   item := stack(top);
   top = top - 1;
end;
```

Time Complexity : o(1)

Implementation of POP algorithm using C language

```
int pop ()
{
   if(top == -1)
   {
      printf("Underflow");
      return 0;
   }
   else
   {
      return stack[top - - ];
   }
}
```

Visiting each element of the stack (Peek operation)

- Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

```
Begin
   if top = -1 then stack empty
   item = stack[top]
   return item
End
```

Time complexity: o(n)

Implementation of Peek algorithm in C language

```
int peek()
{
   if (top == -1)
   {
      printf("Underflow");
      return 0;
   }
   else
   {
      return stack [top];
   }
}
```

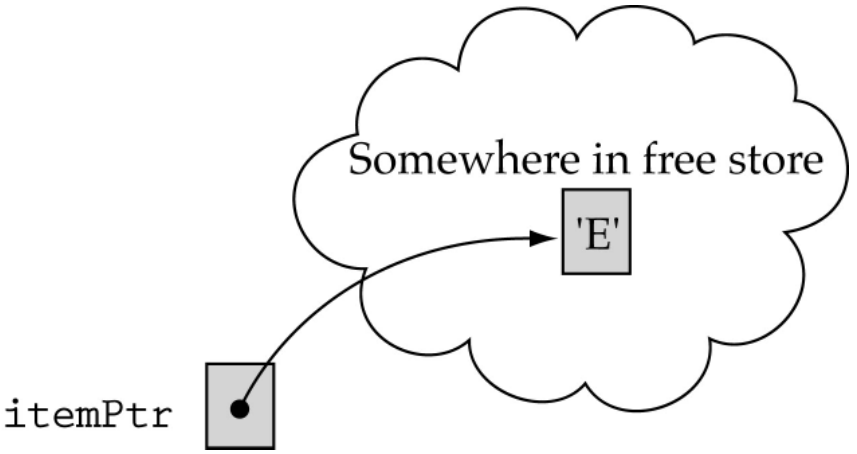
Stack using array

- Simple implementation
- The size of the stack must be determined when a stack object is declared
- Space is wasted if we use less elements
- We cannot "push" more elements than the array can hold

Dynamic allocation of each stack element

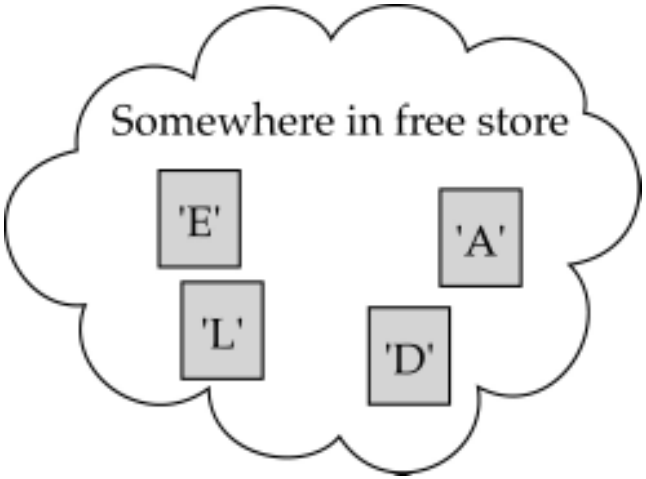
Allocate memory for each new element dynamically

```
ItemType* itemPtr;
 ...
itemPtr = new ItemType;
*itemPtr = newItem;
```



Dynamic allocation of each stack element

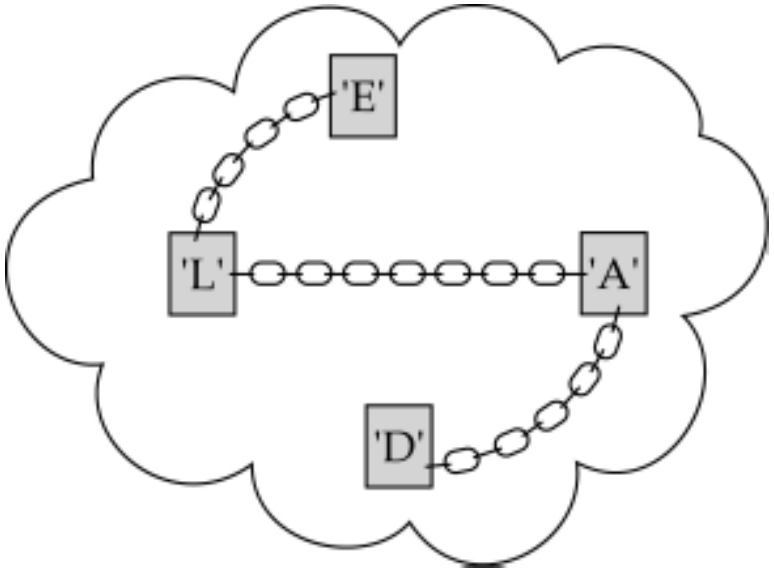
How should we preserve the order of the stack elements?

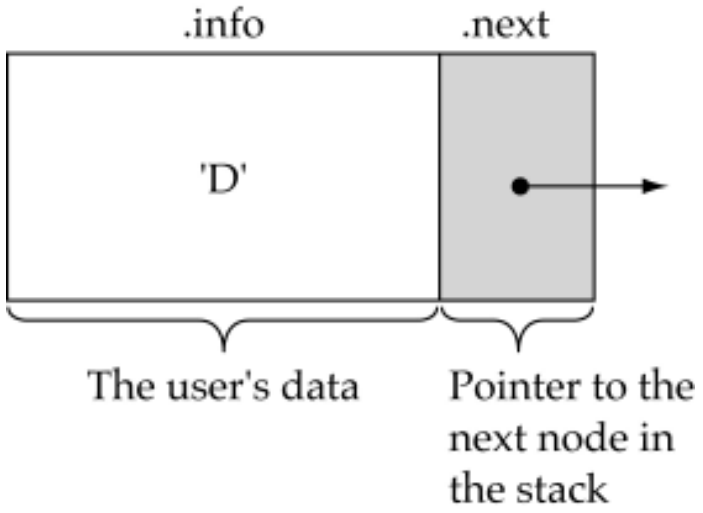


Chaining the stack elements together

Each node in the stack should contain two parts:

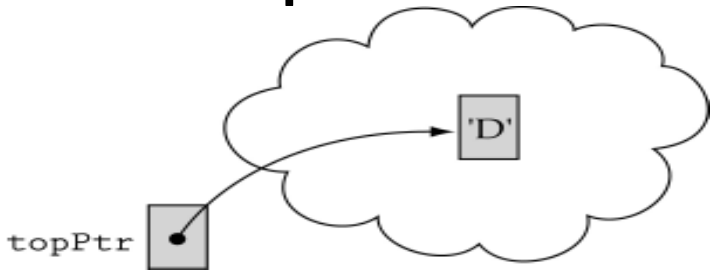
- info: the user's data
- next: the address of the next element in the stack



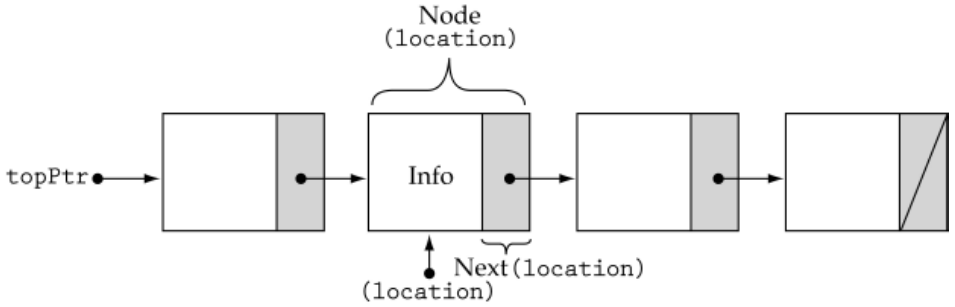


First and last stack elements

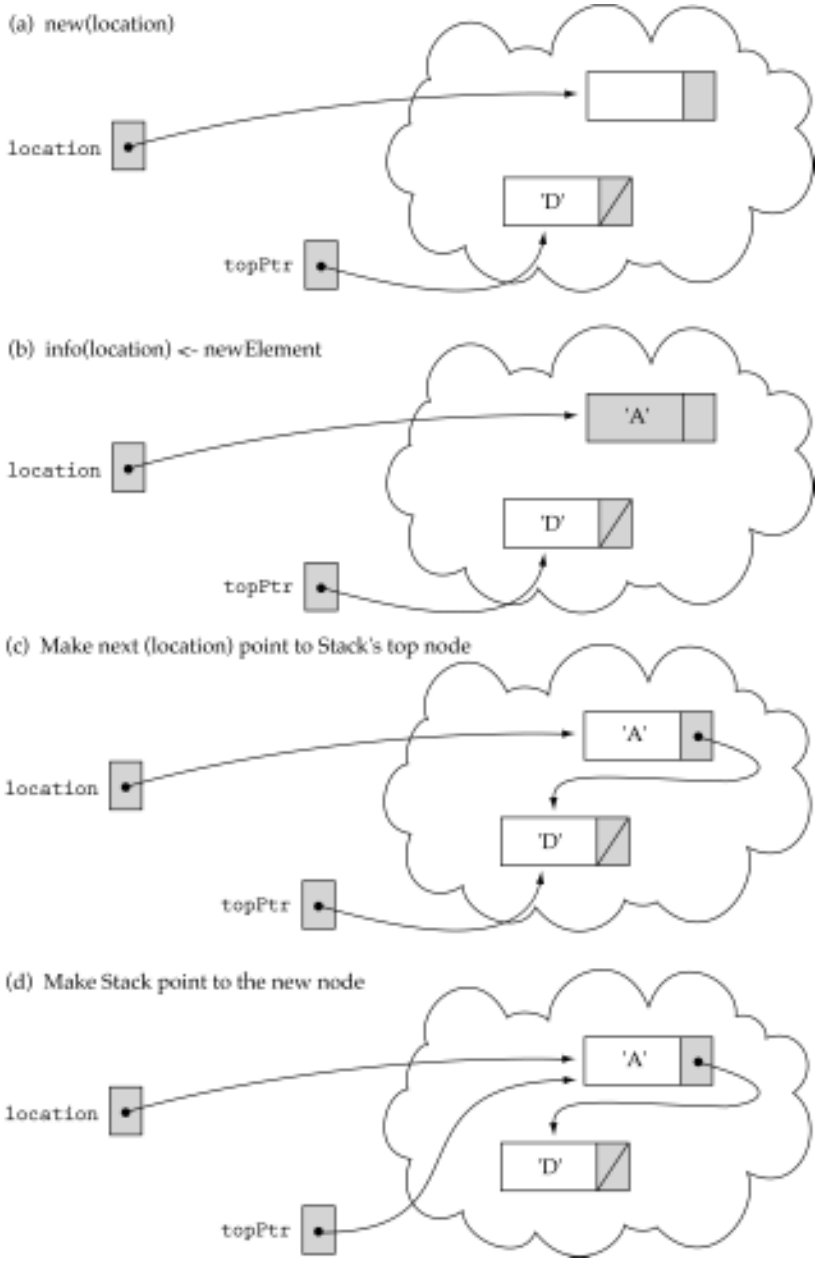
We need a data member to store the pointer to the top of the stack



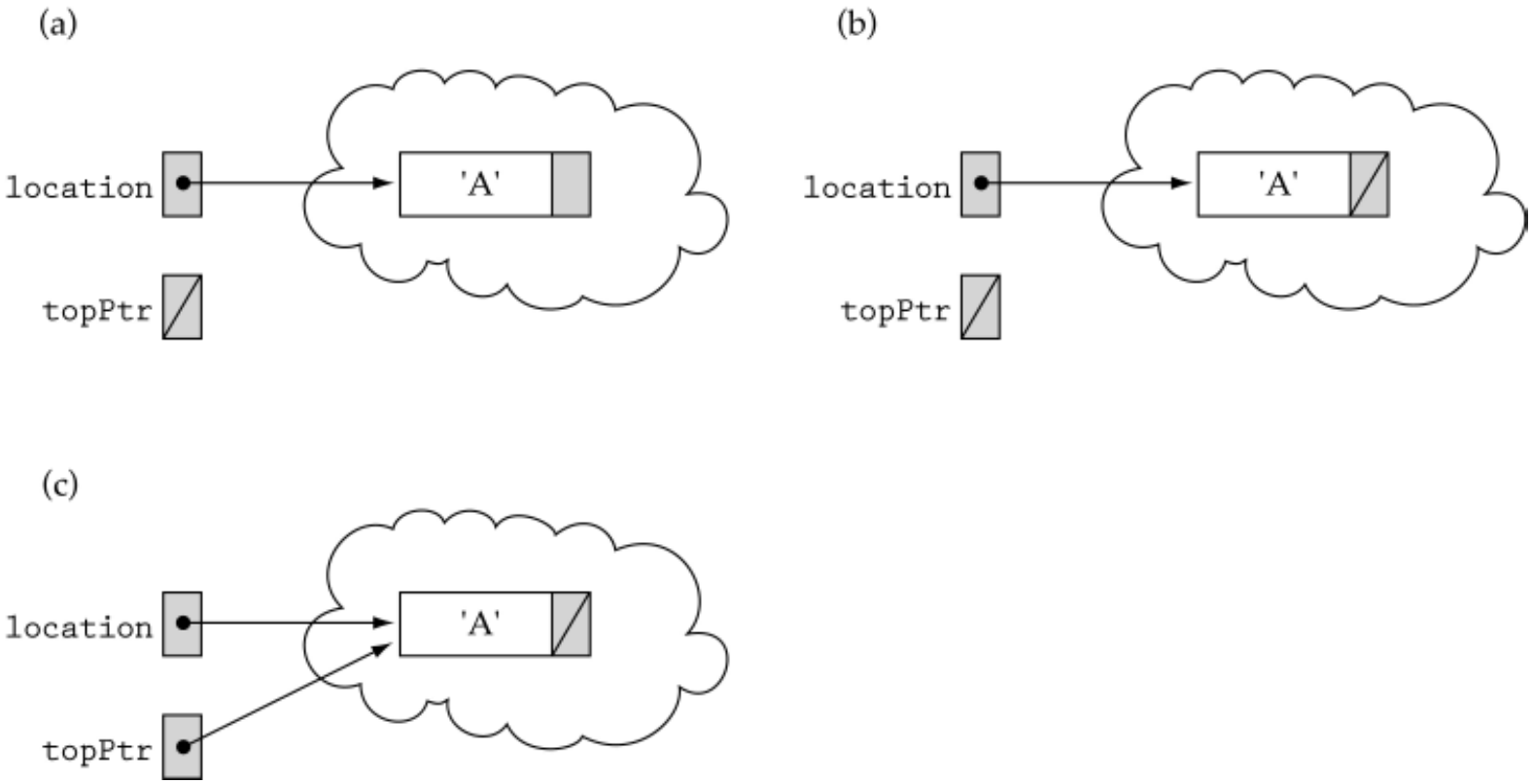
The *next* element of the last node should contain the value *NULL*



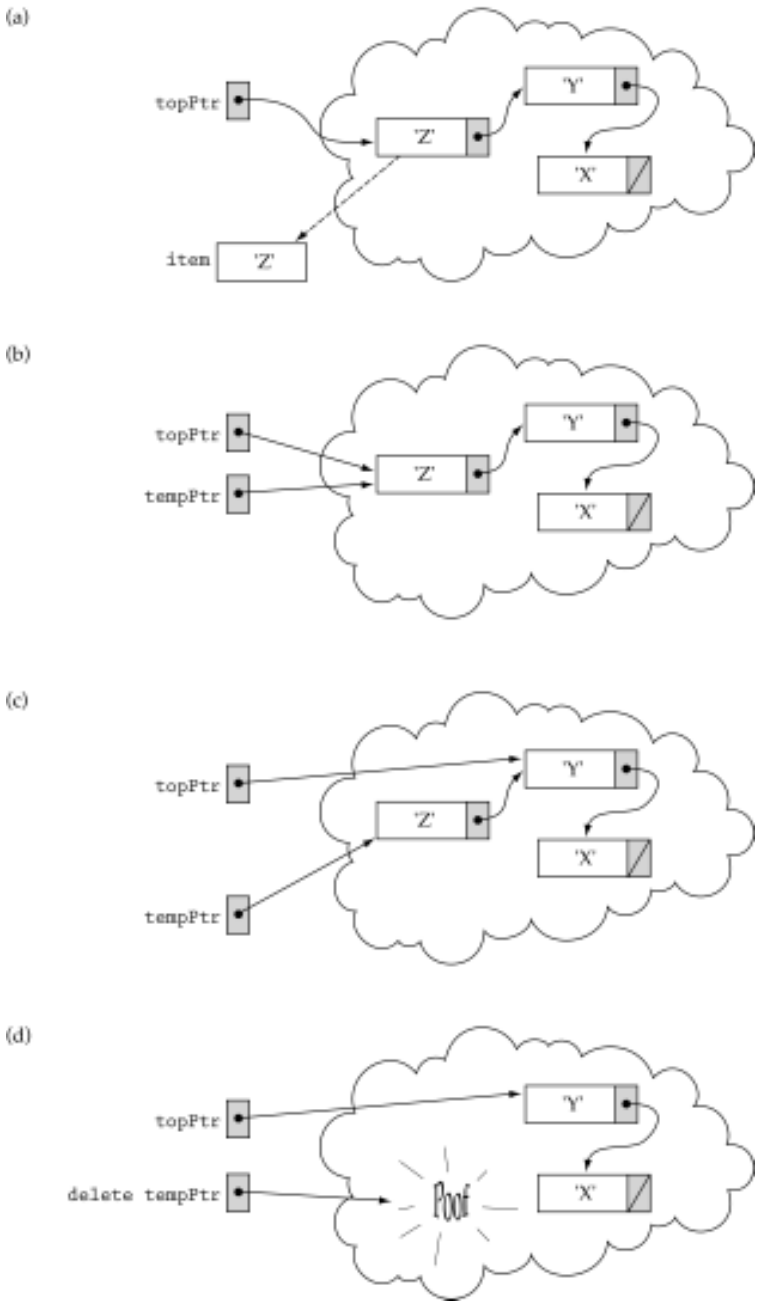
Pushing on a non-empty stack



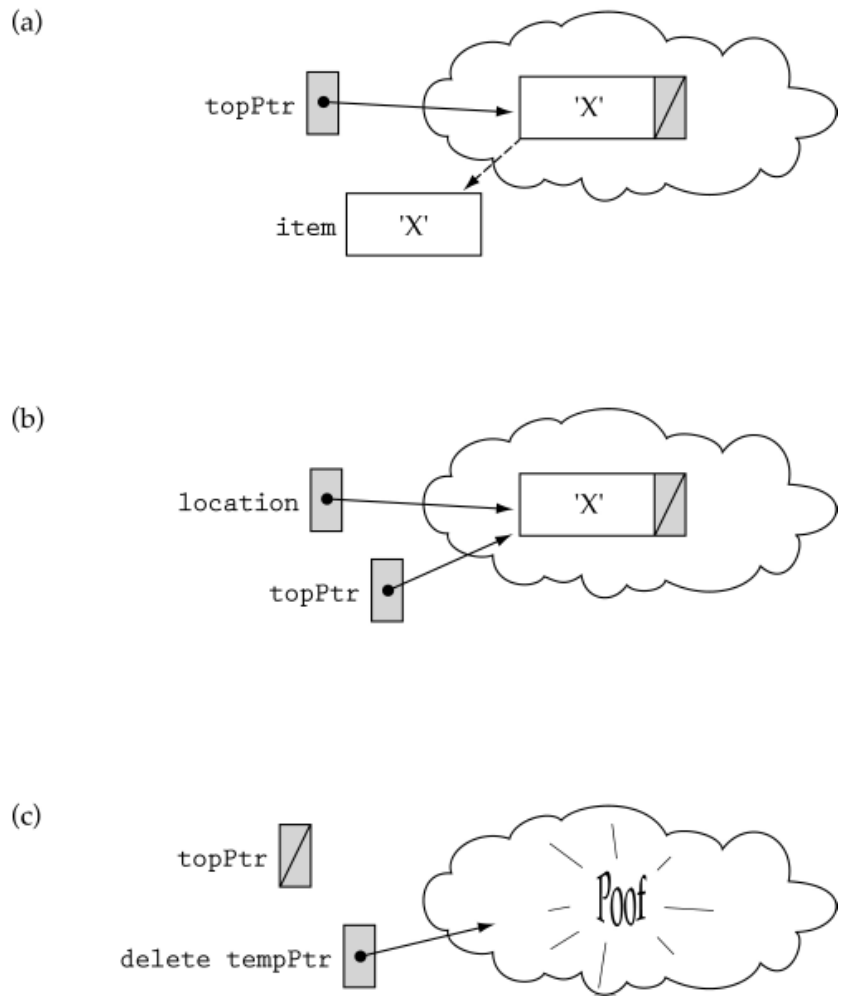
Pushing on an empty stack



Popping the top element



Popping the last element on the stack



QUEUE

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
2. Queue is referred to be as First In First Out list.
3. For example, people waiting in line for a rail ticket form a queue.

Applications of Queue

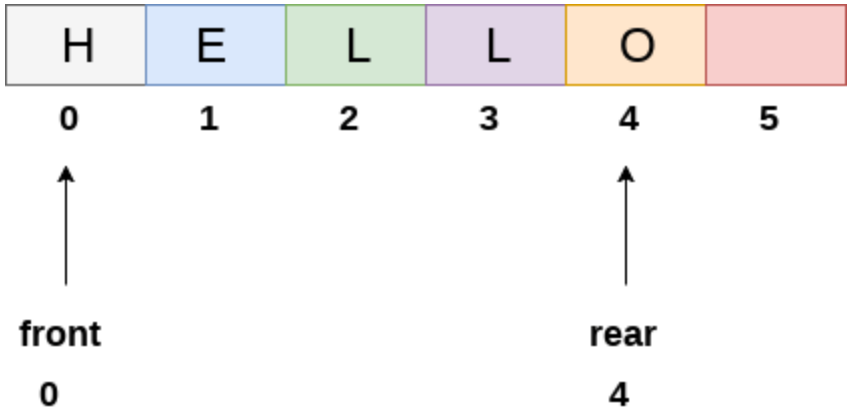
1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Operations on Queue

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** When the Queue is completely full, then it shows the overflow condition.
- **Queue underflow (isempty):** When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

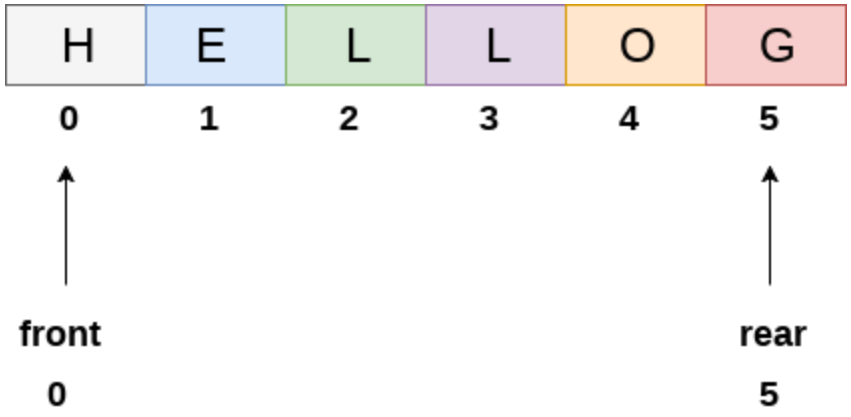
Applications of Circular Queue

- We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



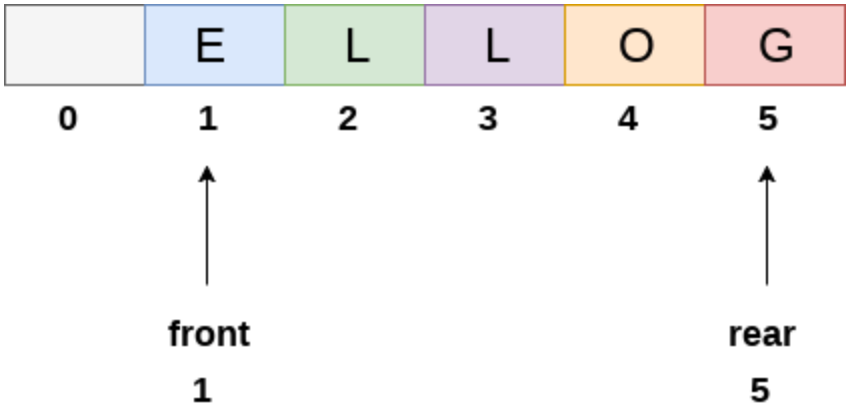
Queue

- The previous figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

- After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

- Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.
- If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.
- Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

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

C Function

```
void insert (int queue[], int max, int front, int rear, int item)
{
   if (rear + 1 == max)
   {
      printf("overflow");
   }
   else
   {
      if(front == -1 && rear == -1)
      {
         front = 0;
         rear = 0;
      }
      else
      {
         rear = rear + 1;
      }
      queue[rear]=item;
   }
}
```

Algorithm to delete an element from the queue

- If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.
- Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

Step 1: IF FRONT = -1 or FRONT > REAR

Write UNDERFLOW

ELSE

SET VAL = QUEUE[FRONT]

SET FRONT = FRONT + 1

[END OF IF]

Step 2: EXIT

C Function

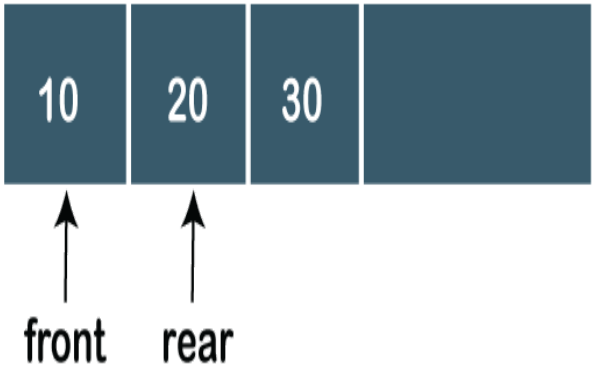
```
int delete (int queue[], int max, int front,
 int rear)
{
   int y;
   if (front == -1 || front > rear)

   {
      printf("underflow");
   }
   else
   {
      y = queue[front];
      if(front == rear)
      {
         front = rear = -1;
         else
         front = front + 1;

      }
      return y;
   }
}
```

Types of Queue

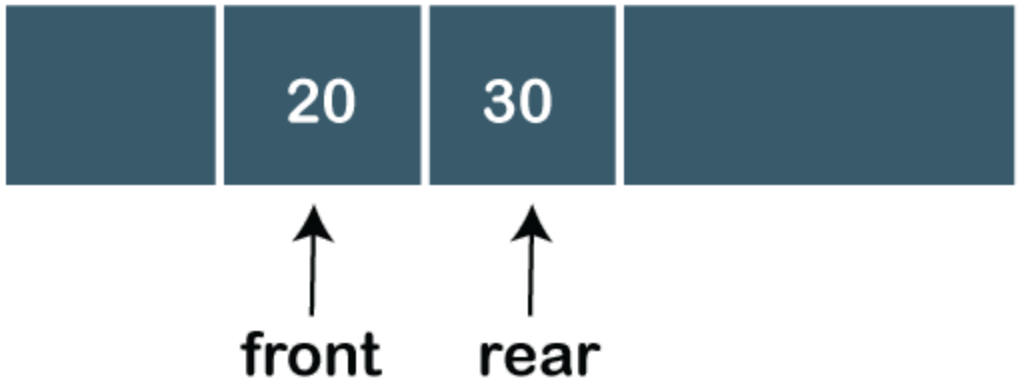
1. Linear/Simple Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended Queue)

1. Linear/Simple Queue

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule. The linear Queue can be represented, as shown in the below figure:



The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion. If we want to show the deletion, then it can be represented as:

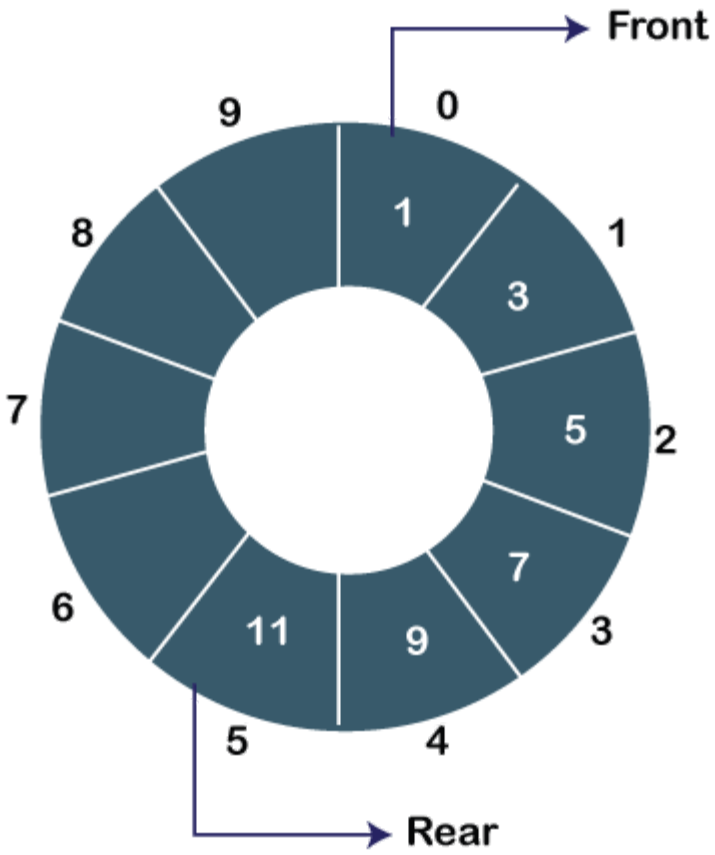


we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.

The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the **overflow** condition as the rear is pointing to the last element of the Queue.

2. Circular Queue

- In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as **Ring Buffer** as all the ends are connected to another end. The circular queue can be represented as:



- The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

Applications of Circular Queue

- **The circular Queue can be used in the following scenarios:**

1. **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
2. **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
3. **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Operations on Circular Queue

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Enqueue operation

- **The steps of enqueue operation are given below:**
- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

-There are two scenarios in which queue is not full:

If rear != max - 1, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.

If front != 0 and rear = max - 1, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

-There are two cases in which the element cannot be inserted:

When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.

front== rear + 1;

Algorithm to insert an element in a circular queue

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

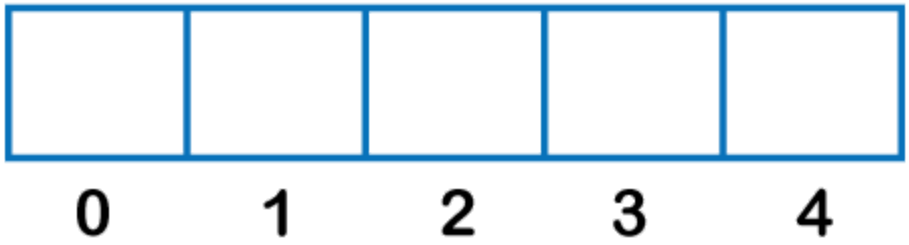
Dequeue Operation

- The steps of dequeue operation are given below:
- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

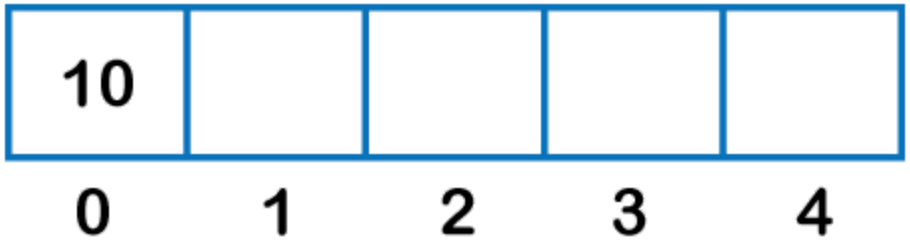
- **Step 1:** IF FRONT = -1
  Write " UNDERFLOW "
  Goto Step 4
  [END of IF]
- **Step 2:** SET VAL = QUEUE[FRONT]
- **Step 3:** IF FRONT = REAR
  SET FRONT = REAR = -1
  ELSE
  IF FRONT = MAX -1
  SET FRONT = 0
  ELSE
  SET FRONT = FRONT + 1
  [END of IF]
  [END OF IF]
- **Step 4:** EXIT

enqueue and dequeue operation through the diagrammatic representation.



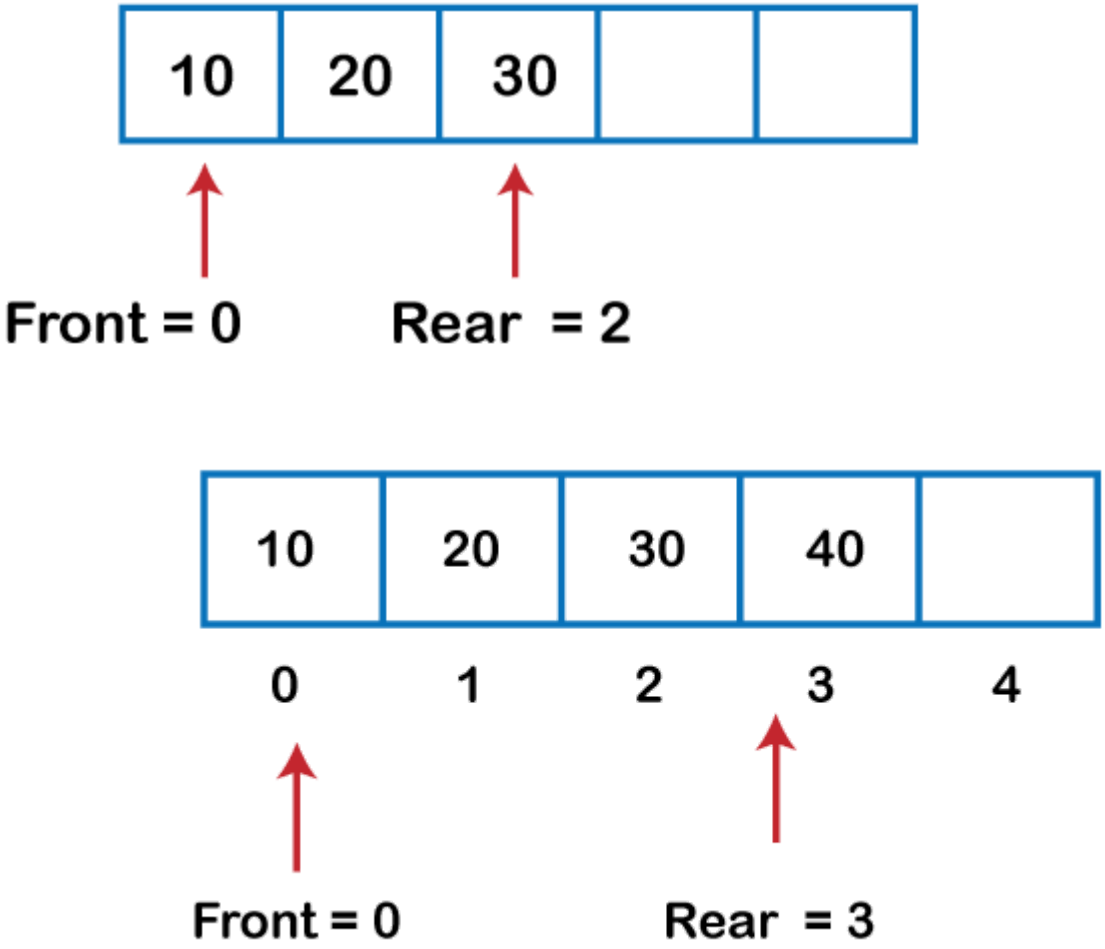
Front = -1

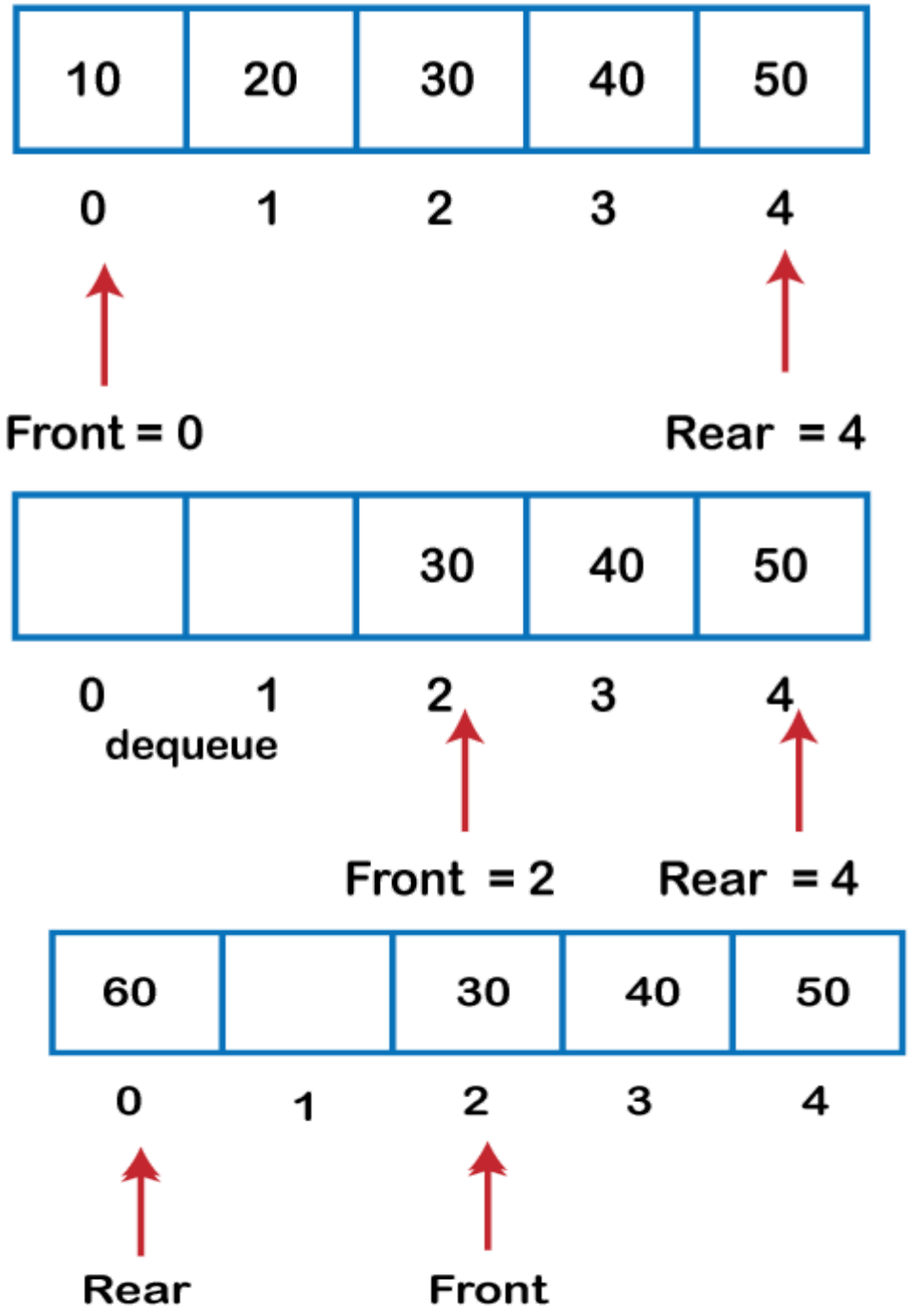
Rear = -1

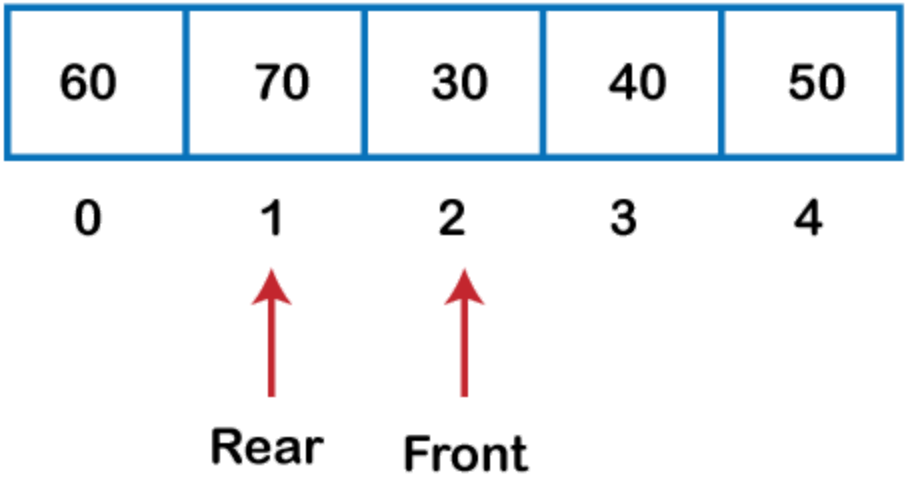


Front = 0

Rear = 0


10
20
30
Front = 0
Rear = 2
10
20
30
40
0
1
2
3
4
Front = 0
Rear = 3


10
20
30
40
50
0
1
2
3
4
Front = 0
Rear = 4
30
40
50
0
1
2
3
4
dequeue
Front = 2
Rear = 4
60
30
40
50
0
1
2
3
4
Rear
Front


60
70
30
40
50
0
1
2
3
4
Rear
Front

3. Priority Queue

- A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.
- In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority.
- the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

What is a priority queue?

- A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.
- The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.
- For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

- We have a priority queue that contains the following values:
- **1, 3, 4, 8, 14, 22**
- All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:
- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

. **There are two types of priority queue:**

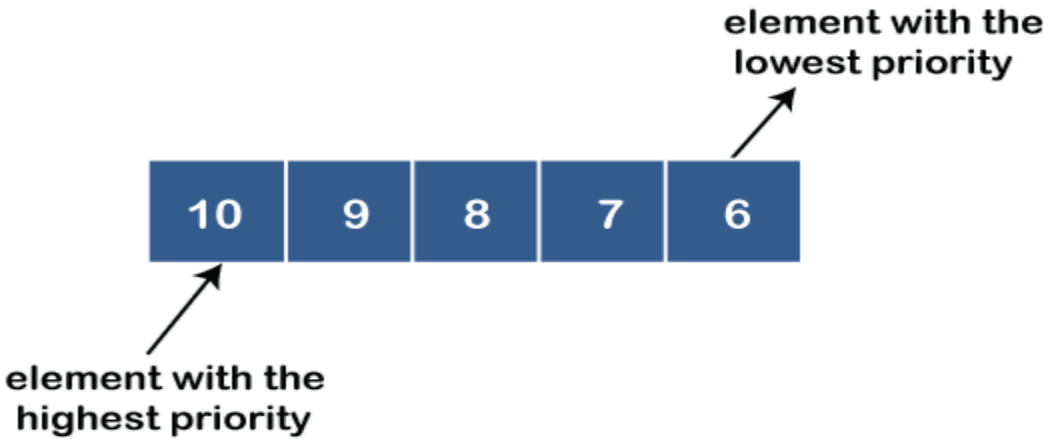
1. **Ascending order priority queue**
2. **Descending order priority queue**

1. Ascending order priority queue

- In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

2. **Descending order priority queue**

- In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Applications of Priority queue

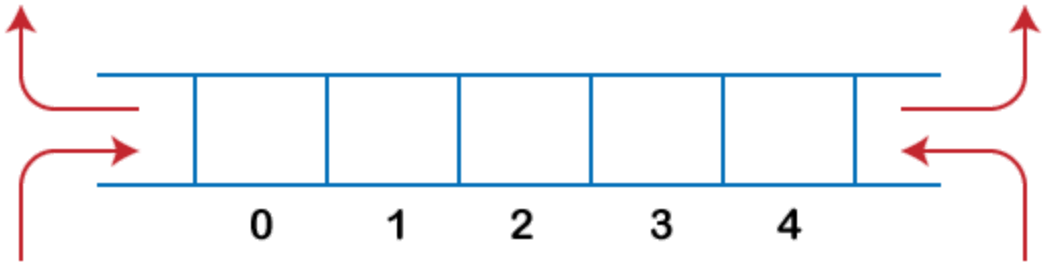
- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

Implementation of Priority Queue

- The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue

4. Dequeue

- The dequeue stands for **Double Ended Queue**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.

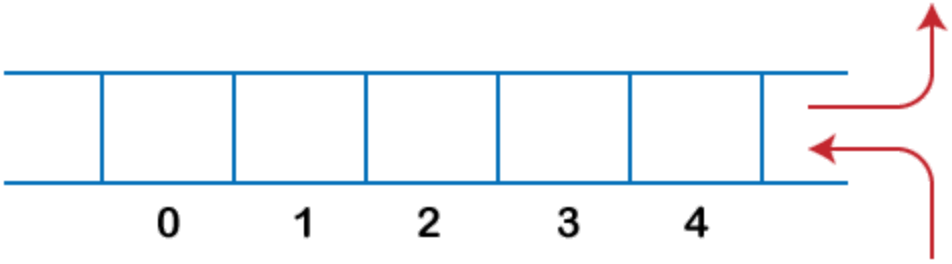


Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

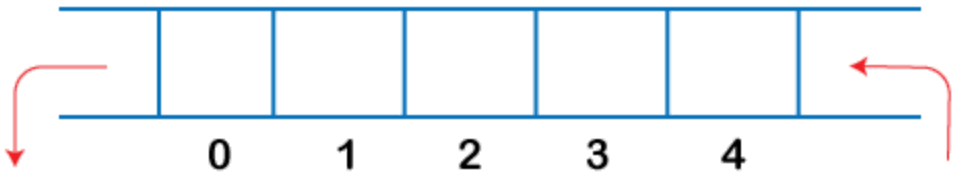
properties of deque.

- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

- In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.

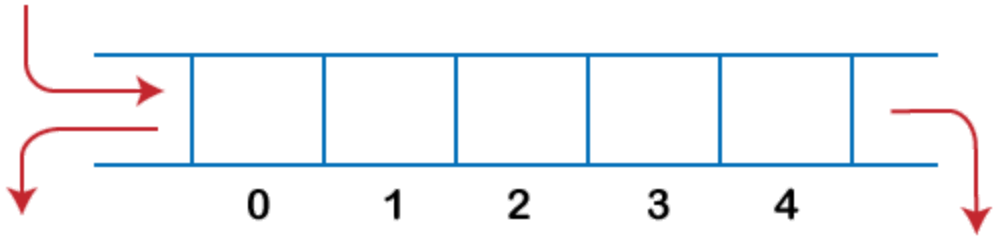


- In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.

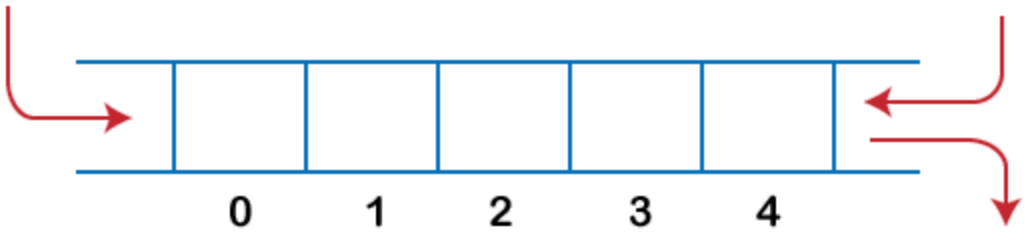


There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. Input-restricted queue: The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Operations on Deque

1. Insert at front

2. Delete from end

3. insert at rear

4. delete from rear

- Other than insertion and deletion, we can also perform **peek** operation in deque.
Through **peek**operation, we can get the **front** and the **rear** element of the dequeue.

- **We can perform two more operations on dequeue:**
- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

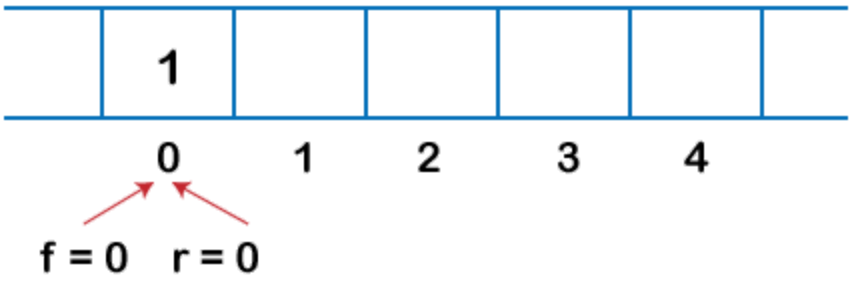
Applications of Deque

- The deque can be used as a **stack** and **queue**; therefore, it can perform both redo and undo operations.
- It can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
- It can be used for multiprocessor scheduling..

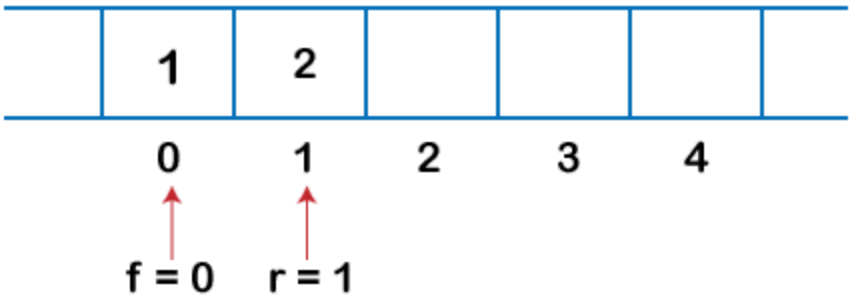
Implementation of Deque using a circular array

- Enqueue operation

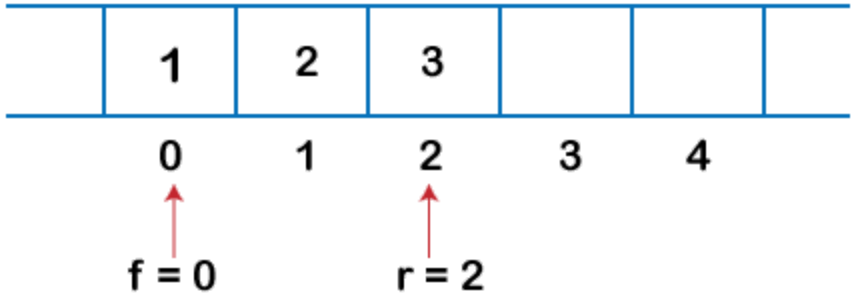
1. Initially, we are considering that the deque is empty, so both front and rear are set to -1, i.e., **f = -1** and **r = -1**.
2. As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then **front is equal to 0,** and the **rear is also equal to 0.**



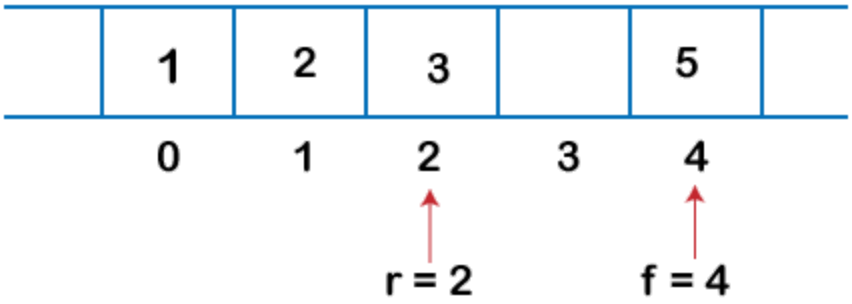
3. Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the second element, and the front is pointing to the first element.



4. Suppose we are again inserting the element from the rear end. To insert the element, we will first increment the rear, and now rear points to the third element.

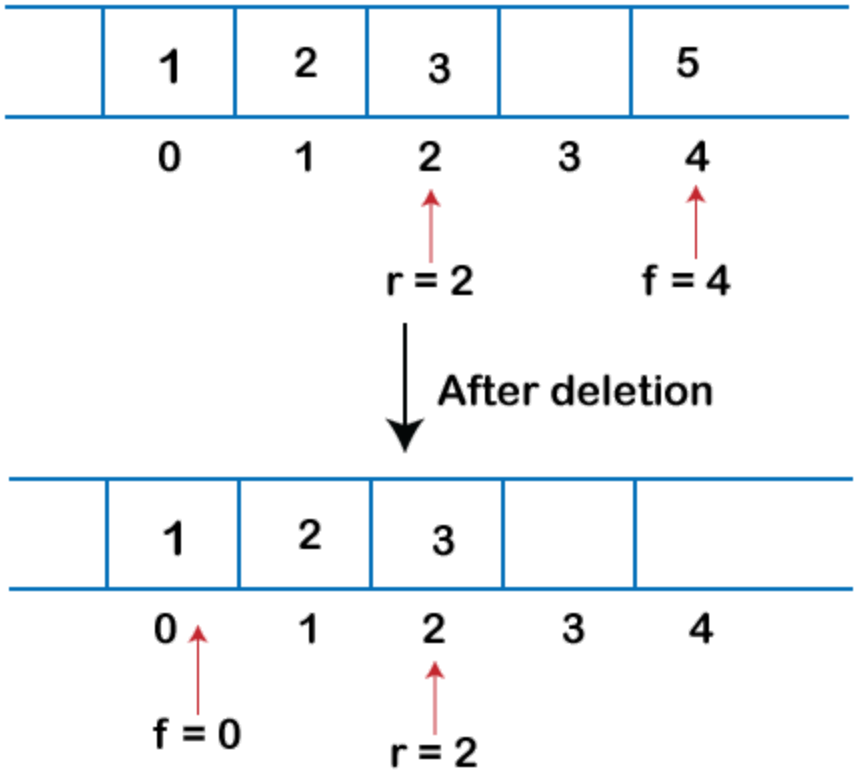


5. If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as **(n -1),** which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:

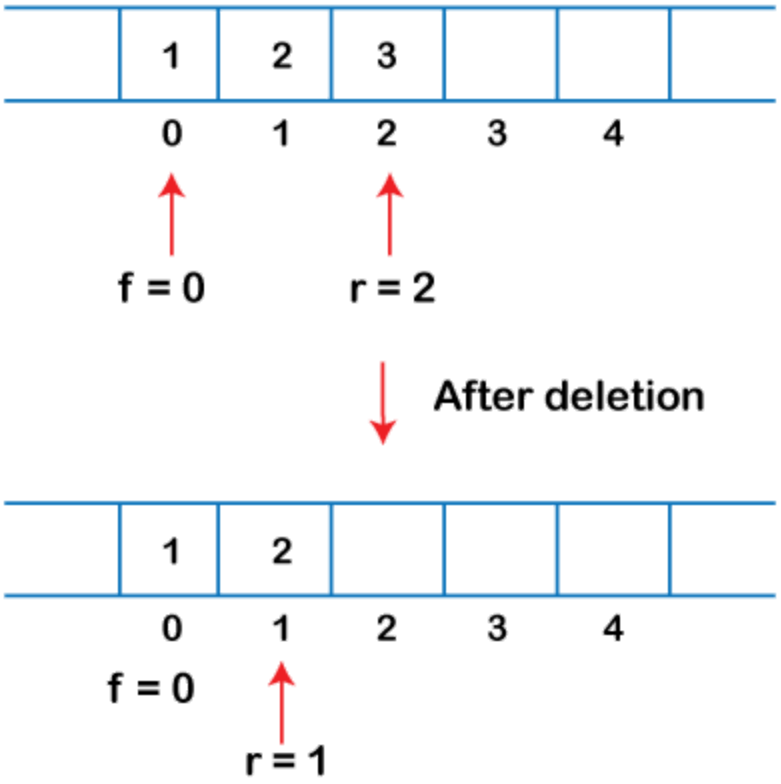


Dequeue Operation

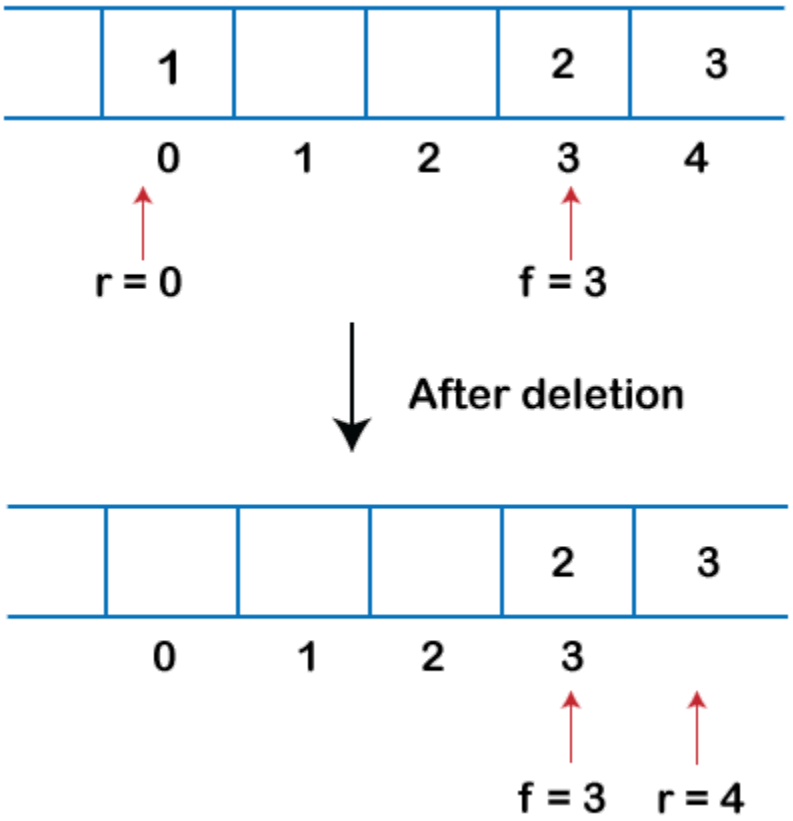
1. If the front is pointing to the last element of the array, and we want to perform the delete operation from the front. To delete any element from the front, we need to set **front=front+1**. Currently, the value of the front is equal to 4, and if we increment the value of front, it becomes 5 which is not a valid index. Therefore, we conclude that if front points to the last element, then front is set to 0 in case of delete operation.


1
2
3
5
0
1
2
3
4
r = 2
f = 4
After deletion
1
2
3
0
1
2
3
4
f = 0
r = 2

2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., **rear=rear-1** as shown in the below figure:



3. If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set **rear=n-1** where **n** is the size of the array as shown in the below figure:



Let's create a program of deque.

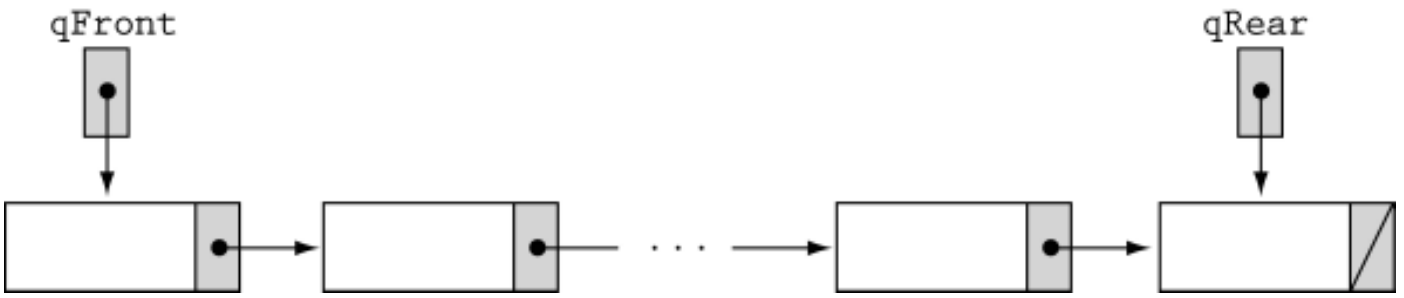
- **enqueue_front():** It is used to insert the element from the front end.
- **enqueue_rear():** It is used to insert the element from the rear end.
- **dequeue_front():** It is used to delete the element from the front end.
- **dequeue_rear():** It is used to delete the element from the rear end.
- **getfront():** It is used to return the front element of the deque.
- **getrear():** It is used to return the rear element of the deque.

Implementing queues using arrays

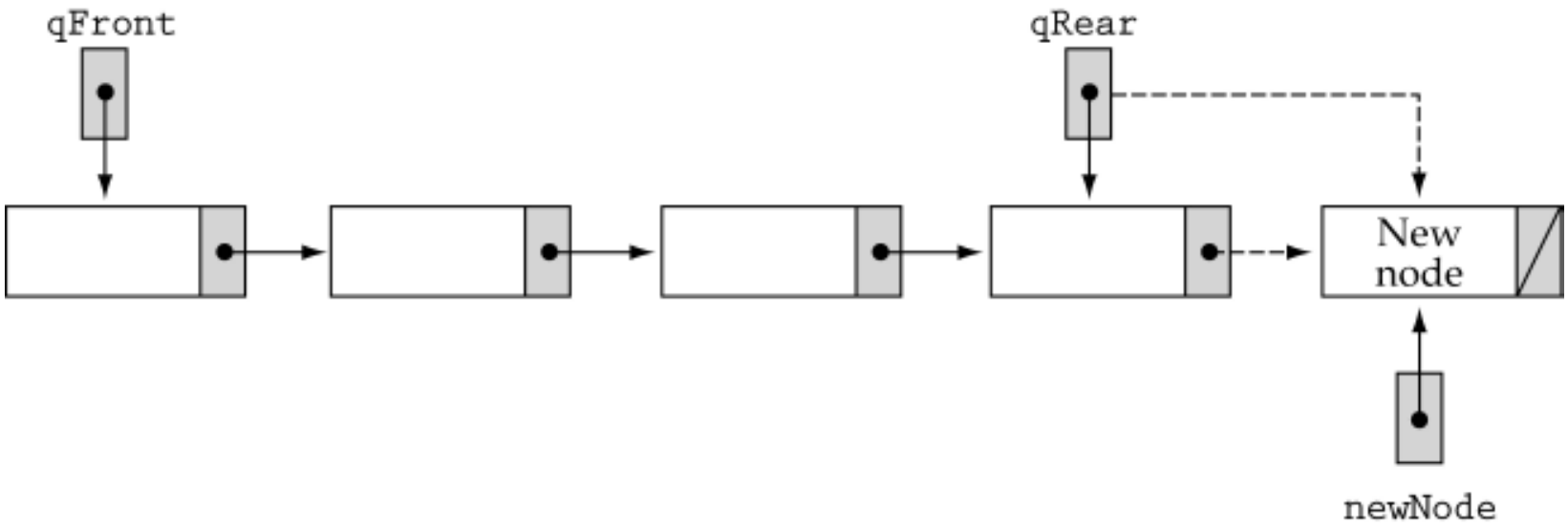
- Simple implementation
- The size of the queue must be determined when a stack object is declared
- Space is wasted if we use less elements
- We cannot "enqueue" more elements than the array can hold

Implementing queues using linked lists

- Allocate memory for each new element dynamically
- Link the queue elements together
- Use two pointers, *qFront* and *qRear*, to mark the front and rear of the queue

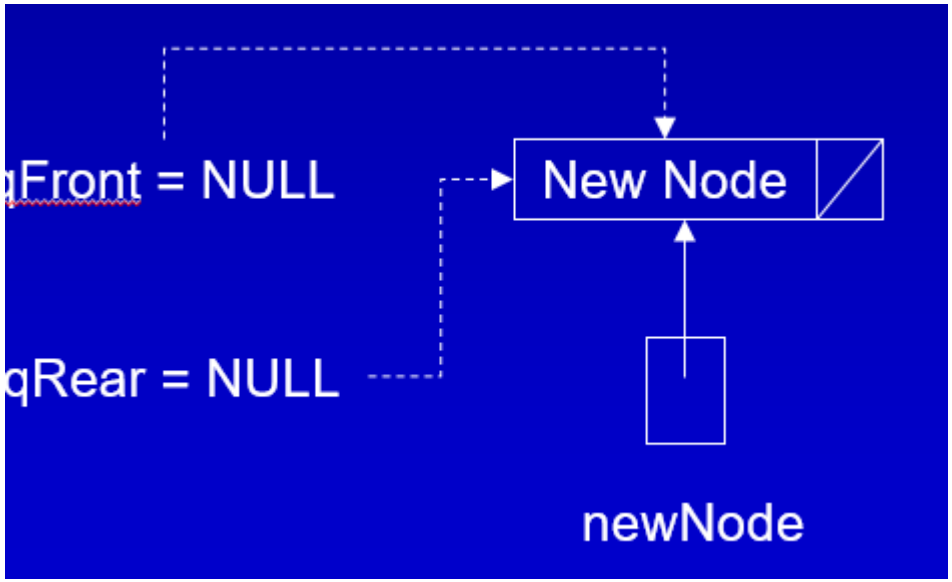


Enqueuing (non-empty queue)



Enqueuing (empty queue)

We need to make *qFront* point to the new node also



Dequeuing (the queue contains more than one element)