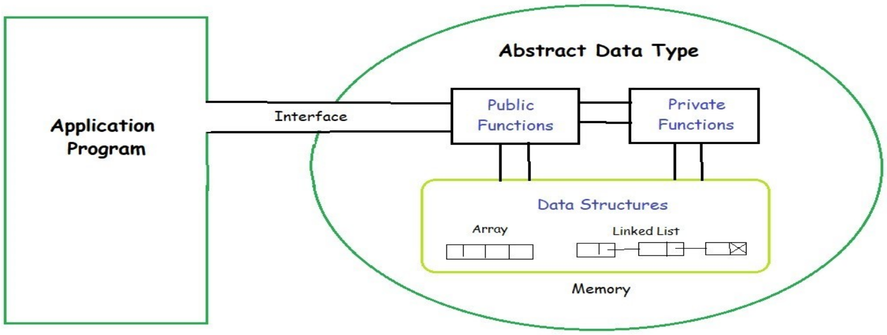
What is abstract data type?

An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.



The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.
So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type.

Implementing Stacks in Data Structures

Stacks in Data Structures is a linear type of data structure that follows the LIFO (Last-In-First-Out) principle and allows insertion and deletion operations from one end of the stack data structure, that is top. Implementation of the stack can be done by contiguous memory which is an array, and non-contiguous memory which is a linked list. Stack plays a vital role in many applications.

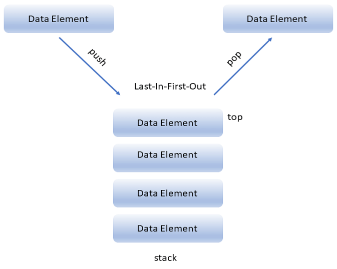
Introduction to Stack in Data Structures

The stack data structure is a linear data structure that accompanies a principle known as LIFO (Last In First Out) or FILO (First In Last Out). Real-life examples of a stack are a deck of cards, piles of books, piles of money, and many more.

This example allows you to perform operations from one end only, like when you insert and remove new books from the top of the stack. It means insertion and deletion in the stack data structure can be done only from the top of the stack. You can access only the top of the stack at any given point in time.

- Inserting a new element in the stack is termed a push operation.
- Removing or deleting elements from the stack is termed pop operation

Stack Representation in Data Structures



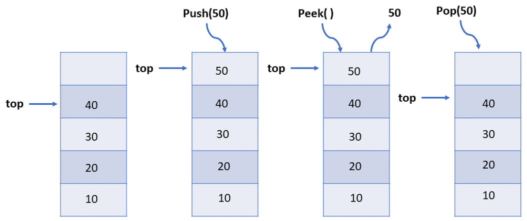
Working of Stack in Data Structures

Now, assume that you have a stack of books.

You can only see the top, i.e., the top-most book, namely 40, which is kept top of the stack.

If you want to insert a new book first, namely 50, you must update the top and then insert a new text.

And if you want to access any other book other than the topmost book that is 40, you first remove the topmost book from the stack, and then the top will point to the next topmost book.



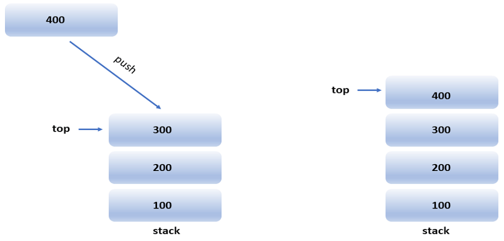
After working on the representation of stacks in data structures, you will see some basic operations performed on the stacks in data structures.

Basic Operations on Stack in Data Structures

There following are some operations that are implemented on the stack.

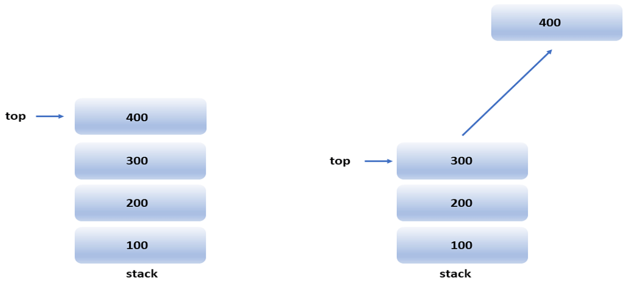
Push Operation

Push operation involves inserting new elements in the stack. Since you have only one end to insert a unique element on top of the stack, it inserts the new element at the top of the stack.



Pop Operation

Pop operation refers to removing the element from the stack again since you have only one end to do all top of the stack. So removing an element from the top of the stack is termed pop operation.



Peek Operation

Peek operation refers to retrieving the topmost element in the stack without removing it from the collections of data elements.

isFull()

isFull function is used to check whether or not a stack is empty.

isEmpty()

isEmpty function is used to check whether or not a stack is empty.

First, you will learn about the functions:

isFull()

The following is the algorithm of the isFull() function:

```
begin

 If
```

```
    top equals to maxsize

        return true

else

       return false

else if

end
```

The implementation of the isFull() function is as follows:

```
Bool isFull()

{

 if(top == maxsize)

 return true;

else

 return false;

}
```

isEmpty()

The following is the algorithm of the isEmpty() function:

```
Begin
 If
    topless than 1
        return true
else
       return false
else if
end
```

The implementation of the isEmpty() function is:

```
Bool isEmpty()
{
 if(top = = -1)
 return true;
else
```

```
 return false;

}
```

Push Operation

Push operation includes various steps, which are as follows :

Step 1: First, check whether or not the stack is full

Step 2: If the stack is complete, then exit

Step 3: If not, increment the top by one

Step 4: Insert a new element where the top is pointing

Step 5: Success

The algorithm of the push operation is:

```
Begin push: stack, item

If the stack is complete, return null

end if

top ->top+1;

stack[top] <- item

end
```

This is how you implement a push operation:

```
if(! isFull ())

{

top = top + 1;

stack[top] = item;

}

else {

 printf(“stack is full”);

}
```

Pop Operation

Step 1: First, check whether or not the stack is empty

Step 2: If the stack is empty, then exit

Step 3: If not, access the topmost data element

Step 4: Decrement the top by one

Step 5: Success

The following is the algorithm of the pop operation:

```
Begin pop: stack

If the stack is empty

 return null

end if

item -> stack[top] ;

Top -> top - 1;

Return item;

End
```

Implementing a pop operation is as follows:

```
int pop( int item){

If isEmpty()) {

item = stack[top];

top = top - 1;

return item;

}
```

```
else{
printf(“stack if empty”);
}
}
```

Peek Operation

The algorithm of a peek operation is:

```
begin to peek
return stack[top];
end
```

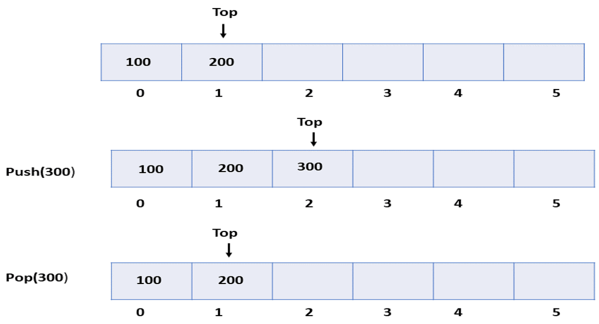
The implementation of the peek operation is:

```
int peek()
{
return stack[top];
}
```

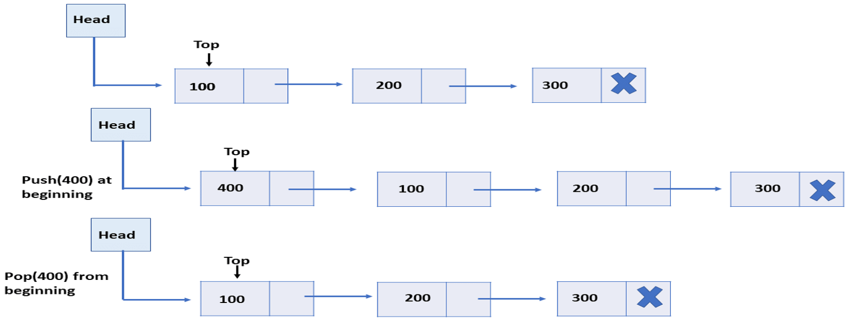
Implementation of Stack in Data Structures

You can perform the implementation of stacks in data structures using two data structures that are an array and a linked list.

- Array: In array implementation, the stack is formed using an array. All the operations are performed using arrays. You will see how all operations can be implemented on the stack in data structures using an array data structure.



- Linked-List: Every new element is inserted as a top element in the linked list implementation of stacks in data structures. That means every newly inserted element is pointed to the top. Whenever you want to remove an element from the stack, remove the node indicated by the top, by moving the top to its previous node in the list.



Application of Stack in Data Structures

Here are the top 7 applications of the stack in data structure:

- Expression Evaluation and Conversion

- Backtracking
- Function Call
- Parentheses Checking
- String Reversal
- Syntax Parsing
- Memory Management

1. Expression Evaluation and Conversion

There are three types of expression that you use in programming, they are:

Infix Expression: An infix expression is a single letter or an operator preceded by one single infix string followed by another single infix string.

- X
- X + Y
- (X + Y) + (A - B)

Prefix Expression: A prefix expression is a single letter or an operator followed by two prefix strings.

- X
- + X Y
- + + X Y - A B

Postfix Expression: A postfix expression (also called Reverse Polish Notation) is a single letter or an operator preceded by two postfix strings.

- X
- X Y +
- X Y + C D - +

Similarly, the stack is used to evaluate these expressions and convert these expressions like infix to prefix or infix to postfix.

2. Backtracking

Backtracking is a recursive algorithm mechanism that is used to solve optimization problems.

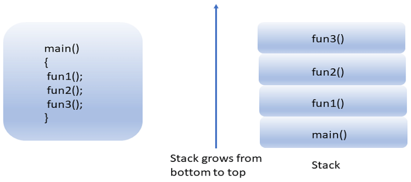
To solve the optimization problem with backtracking, you have multiple solutions; it does not matter if it is correct. While finding all the possible solutions in backtracking, you store the previously calculated problems in the stack and use that solution to resolve the following issues.

The N-queen problem is an example of backtracking, a recursive algorithm where the stack is used to solve this problem.

3. Function Call

Whenever you call one function from another function in programming, the reference of calling function stores in the stack. When the function call is terminated, the program control moves back to the function call with the help of references stored in the stack.

So stack plays an important role when you call a function from another function.



4. Parentheses Checking

Stack in data structures is used to check if the parentheses like ( ), { } are valid or not in programing while matching opening and closing brackets are balanced or not.

So it stores all these parentheses in the stack and controls the flow of the program.

For e.g ((a + b) * (c + d)) is valid but {{a+b})) *(b+d}] is not valid.

5. String Reversal

Another exciting application of stack is string reversal. Each character of a string gets stored in the stack.

The string's first character is held at the bottom of the stack, and the last character of the string is held at the top of the stack, resulting in a reversed string after performing the pop operation.

6. Syntax Parsing

Since many programming languages are context-free languages, the stack is used for syntax parsing by many compilers.

7. Memory Management

Memory management is an essential feature of the operating system, so the stack is heavily used to manage memory.

Advantages of Stack:

- **Easy implementation:** Stack data structure is easy to implement using arrays or linked lists, and its operations are simple to understand and implement.
- **Efficient memory utilization**: Stack uses a contiguous block of memory, making it more efficient in memory utilization as compared to other data structures.
- **Fast access time:** Stack data structure provides fast access time for adding and removing elements as the elements are added and removed from the top of the stack.
- **Helps in function calls:** Stack data structure is used to store function calls and their states, which helps in the efficient implementation of recursive function calls.
- **Supports backtracking:** Stack data structure supports backtracking algorithms, which are used in problem-solving to explore all possible solutions by storing the previous states.
- **Used in Compiler Design:** Stack data structure is used in compiler design for parsing and syntax analysis of programming languages.
- **Enables undo/redo operations**: Stack data structure is used to enable undo and redo operations in various applications like text editors, graphic design tools, and software development environments.

Disadvantages of Stack:

- **Limited capacity:** Stack data structure has a limited capacity as it can only hold a fixed number of elements. If the stack becomes full, adding new elements may result in stack overflow, leading to the loss of data.
- **No random access:** Stack data structure does not allow for random access to its elements, and it only allows for adding and removing elements from the top of the stack. To access an element in the middle of the stack, all the elements above it must be removed.
- **Memory management:** Stack data structure uses a contiguous block of memory, which can result in memory fragmentation if elements are added and removed frequently.
- **Not suitable for certain applications:** Stack data structure is not suitable for applications that require accessing elements in the middle of the stack, like searching or sorting algorithms.
- **Stack overflow and underflow**: Stack data structure can result in stack overflow if too many elements are pushed onto the stack, and it can result in stack underflow if too many elements are popped from the stack.
- **Recursive function calls limitations:** While stack data structure supports recursive function calls, too many recursive function calls can lead to stack overflow, resulting in the termination of the program.

Expression Evaluation Using Stack

Infix, Postfix and Prefix Expressions

Before we start writing the code for expression evaluation using stack, let us have a quick recap of the different kinds of expressions, namely, infix, postfix and prefix, we'll be solving.

Infix Expressions

The usual expressions which we encounter are infix expressions.
For example,
(A + B) * C / D - E

Postfix Expressions

In postfix expressions, the operators are written after the operands as shown below:
A B C + * D /

Prefix Expressions

Here, the operators are written before the operands. An example is,
/ * A + B C D
Now that we know what expressions we're dealing with, solving them will be even easier. Before moving on to evaluation, there is one more thing we need to learn - the precedence of operators.

Precedence of Operators

In our school days, we read about BODMAS or PEMDAS. What exactly were those? They were acronyms to remember the precedence of operators while solving an expression, but what is precedence?

The precedence of operators gives us an order in which operators are evaluated in any expression.

Computers have a similar idea of precedence, too, when it comes to operators. It is as follows:

1. Exponential (^)
2. Multiplication and division (* /)
3. Addition and subtraction (+ –)

Infix Expression Evaluation Using Stack

To begin with, let us see how infix expression evaluation using stack.

Algorithm

Step 1: Create two stacks - the operand stack and the character stack.
Step 2: Push the character to the operand stack if it is an operand.
Step 3: If it is an operator, check if the operator stack is empty.
Step 4: If the operator stack is empty, push it to the operator stack.
Step 5: If the operator stack is not empty, compare the precedence of the operator and the top character in the stack.
If the character's precedence is greater than or equal to the precedence of the stack top of the operator stack, then push the character to the operator stack.
Otherwise, pop the elements from the stack until the character's precedence is less or the stack is empty.
Step 6: If the character is "(", push it into the operator stack.
Step 7: If the character is ")", then pop until "(" is encountered in the operator stack.

Example

Infix expression : 2 * (5 * (3 + 6)) / 5 - 2

Character	Action	Operand Stack	Operator Stack
2	Push to the operand stack	2	
*	Push to the operator stack	2	*
(	Push to the operator stack	2	(*
5	Push to the operand stack	5 2	(*
*	Push to the operator stack	5 2	* (*
(	Push to the operator stack	2 1	(* (*

3	Push to the operand stack	3 5 2	(* (*
+	Push to the operator stack	3 2 1	+ (* (*
6	Push to the operand stack	6 3 5 2	+ (* (*
)	Pop 6 and 3	5 2	+ (* (*
	Pop +	5 2	(* (*
	6 + 3 = 9, push to operand stack	9 5 2	(* (*
	Pop (	9 5 2	* (*
)	Pop 9 and 5	2	* (*
	Pop *	2	(*
	9 * 5 = 45, push to operand stack	45 2	(*
	Pop (	45 2	*
/	Push to the operator stack	45 2	/ *
5	Push to the operand stack	5 45 2	/ *
-	Pop 5 and 45	2	/ *
	Pop /	2	*
	45/5 = 9, push to the operand stack	9 2	*
	Pop 9 and 2		*
	Pop *		
	9 * 2 = 18, push to operand stack	18	

	Push - to the operator stack	18	-
2	Push to the operand stack	2 18	-
	Pop 2 and 18		-
	Pop -		
	18 - 2 = 16, push to operand stack	16	

Postfix Expression Evaluation Using Stack

Now that we know how to evaluate an infix expression let us move on to the next type - postfix evaluation.

Algorithm

Here we will use only one operand stack instead of two.

Step 1: Create an operand stack.
Step 2: If the character is an operand, push it to the operand stack.
Step 3: If the character is an operator, pop two operands from the stack, operate and push the result back to the stack.
Step 4:After the entire expression has been traversed, pop the final result from the stack.

Example

For example, let us convert the infix expression we used into postfix for expression evaluation using stack.

Postfix expression : 2 5 3 6 + * * 15 / 2 -

Character	Action	Operand Stack
2	Push to the operand stack	2
5	Push to the operand stack	5 2
3	Push to the operand stack	3 5 2
6	Push to the operand stack	6 3 5 2
+	Pop 6 and 3 from the stack	5 2
	6 + 3 = 9, push to operand stack	9 5 2

*	Pop 9 and 5 from the stack	2
	9 * 5 = 45, push to operand stack	45 2
*	Pop 45 and 2 from the stack	
	45 * 2 = 90, push to stack	90
5	Push to stack	5 90
/	Pop 15 and 90 from the stack	
	90 / 5 = 18, push to stack	18
2	Push to the stack	2 18
-	Pop 2 and 18 from the stack	
	18 - 2 = 16, push to stack	16

Prefix Expression Evaluation

The last kind of expression we are left to discuss is a prefix expression. Let us see how we'll evaluate it.

Algorithm

Understanding the algorithm to evaluate a prefix expression will be very easy since we already know how to evaluate a postfix expression.

Here, we will first reverse the prefix expression, and the rest of the algorithm is the same as that for a postfix expression.

Step 1: Reverse the postfix expression.
Step 2: Create an operand stack.
Step 3: If the character is an operand, push it to the operand stack.
Step 4: If the character is an operator, pop two operands from the stack, operate and push the result back to the stack.
Step 5:After the entire expression has been traversed, pop the final result from the stack.

Example

Let us again convert the infix expression from our first example to a prefix expression to evaluate it.
Prefix expression: - / * 2 * 5 + 3 6 5 2
Reversed prefix expression: 2 5 6 3 + 5 * 2 * / -

Character	Action	Operand Stack
2	Push to the operand stack	2
5	Push to the operand stack	5 2
6	Push to the operand stack	6 5 2
3	Push to the operand stack	3 6 5 2
+	Pop 3 and 6 from the stack	5 2
	3 + 6 = 9, push to operand stack	9 5 2
5	Push to the operand stack	5 9 5 2
*	Pop 5 and 9 from the stack	5 2
	5 * 9 = 45, push to operand stack	45 5 2
2	Push to operand stack	2 45 5 2
*	Pop 2 and 45 from the stack	5 2
	2 * 45 = 90, push to stack	90 5 2
/	Pop 90 and 5 from the stack	2
	90 / 5 = 18, push to stack	18 2
-	Pop 18 and 2 from the stack	
	18 - 2 = 16, push to stack	16

Convert Infix expression to Postfix expression

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left. Consider the expression: **a + b * c + d**

- The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it.
- The result is then added to d after another scan.

The repeated scanning makes it very inefficient. Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix(or prefix) form before evaluation.

The corresponding expression in postfix form is **abc*+d+**. The postfix expressions can be evaluated easily using a stack.

How to convert an Infix expression to a Postfix expression?

To convert infix expression to postfix expression, use the ***stack data structure****. Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.*

Below are the steps to implement the above idea:

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
   - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a ‘**(**‘], then push it in the stack. [‘**^**‘ operator is right associative and other operators like ‘**+**‘,’**–**‘,’*****‘ and ‘**/**‘ are left-associative].
     - Check especially for a condition when the operator at the top of the stack and the scanned operator both are ‘**^**‘. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
     - In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
   - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
     - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is a ‘**(**‘, push it to the stack.

5. If the scanned character is a ‘**)**’, pop the stack and output it until a ‘**(**‘ is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Complexity

The time and space complexity of Conversion of Infix expression to Postfix expression algorithm is :

- Worst case time complexity: **Θ(n^2)**
- Average case time complexity: **Θ(n^2)**
- Best case time complexity: **Θ(n^2)**
- Space complexity: **Θ(n)**
  where N is the number of literals in Infix Expression

- Complexity of algorithm in both worst and best case is O(n^2), as expression is iterated two times simultaneously, firstly for scanning the infix expression and secondly while poping out of stack.
  Eg : a + b - d
  * As in above Infix expression, O(n) will be the complexity for scanning each literal, while at the same time we pop the literals from stack, hence the complexity of algorithm is O(n*n) i.e : O(n^2).
- For storing Infix expression of n literals the space complexity is O(n) and for stack to hold atmost n literals the space complexity is O(n), hence
  * Total space complexity is O(n+n) = O(2n) i.e : O(n)

Queue

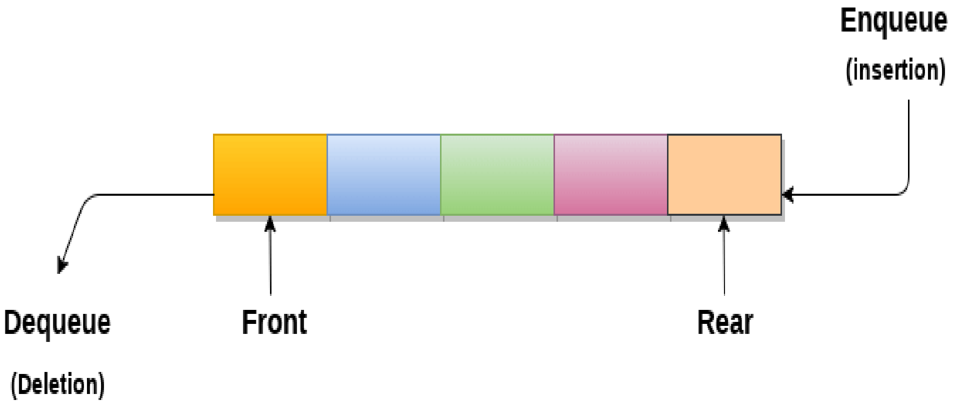
Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

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.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



Basic Operations on Queue:

- **enqueue():** Inserts an element at the end of the queue i.e. at the rear end.
- **dequeue():** This operation removes and returns an element that is at the front end of the queue.
- **front():** This operation returns the element at the front end without removing it.
- **rear():** This operation returns the element at the rear end without removing it.
- **isEmpty():** This operation indicates whether the queue is empty or not.
- **isFull():** This operation indicates whether the queue is full or not.
- **size():** This operation returns the size of the queue i.e. the total number of elements it contains.

Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

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.

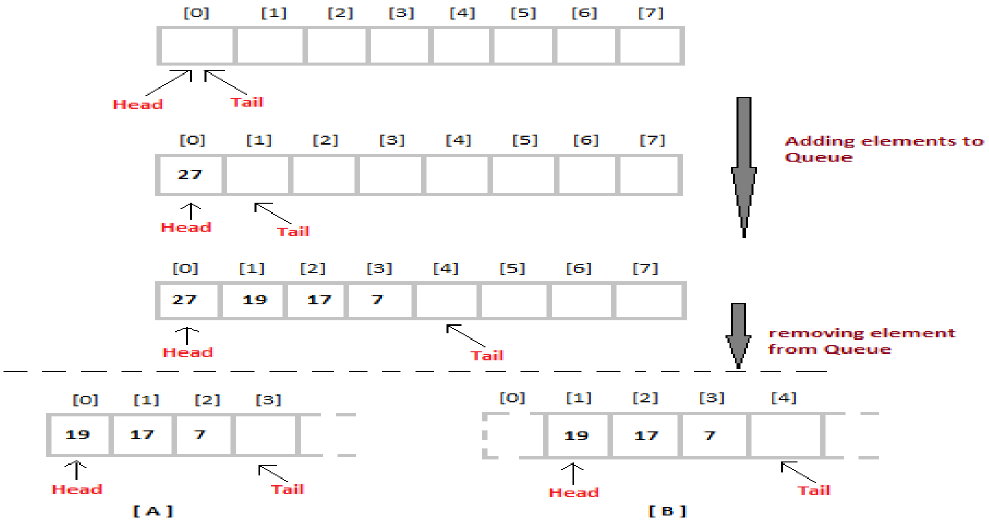
Complexity

Data Structure	**Time Complexity**								**Space Compleity**
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from `0`). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

Types of Queue

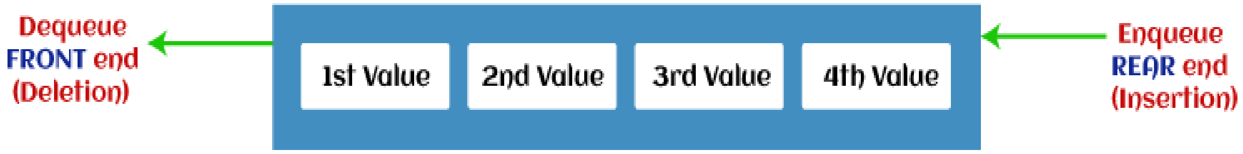
There are four different types of queue that are listed as follows -

- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Let's discuss each of the type of queue.

Simple Queue or Linear 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 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.

Array representation of 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 to insert any element in a queue

```
○ Step1: IF REAR = MAX - 1
    Write OVERFLOW
    Goto step3
    [END OF IF]
○ Step2: 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
```

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.

```
  ○  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
```

Advantages of Array Implementation:

- Easy to implement.
- A large amount of data can be managed efficiently with ease.
- Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.

Disadvantages of Array Implementation:

- Static Data Structure, fixed size.
- If the queue has a large number of enqueue and dequeue operations, at some point (in case of linear increment of front and rear indexes) we may not be able to insert elements in the queue even if the queue is empty (this problem is avoided by using circular queue).
- Maximum size of a queue must be defined prior.

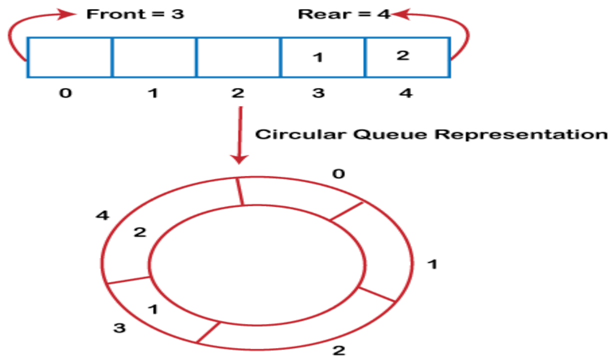
Circular Queue

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 representation of circular queue is shown in the below image -

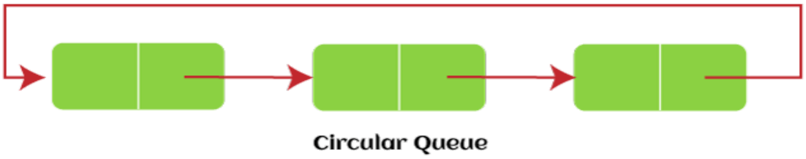
Circular Queue

Why was the concept of the circular queue introduced?

There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see



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, we can insert an element in the first position. This action is not possible in a simple queue.

Operations on Circular Queue

The following are the operations that can be performed on a 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.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **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.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **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.

Enqueue operation

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

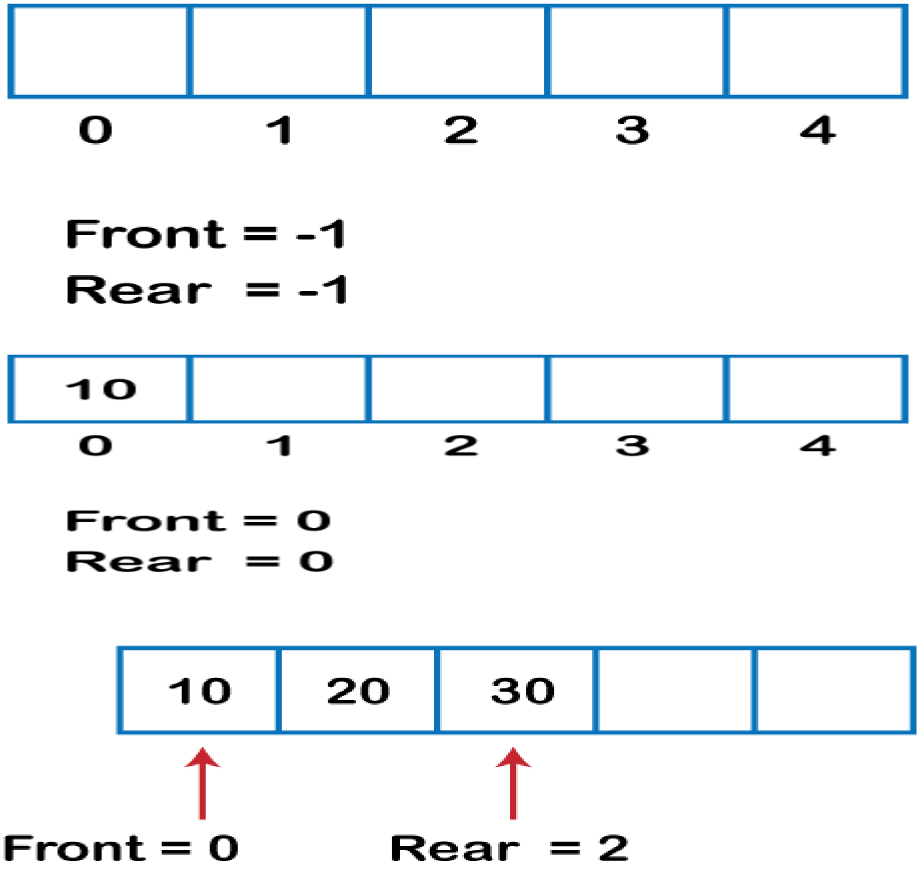
The steps of dequeue operation are given below:

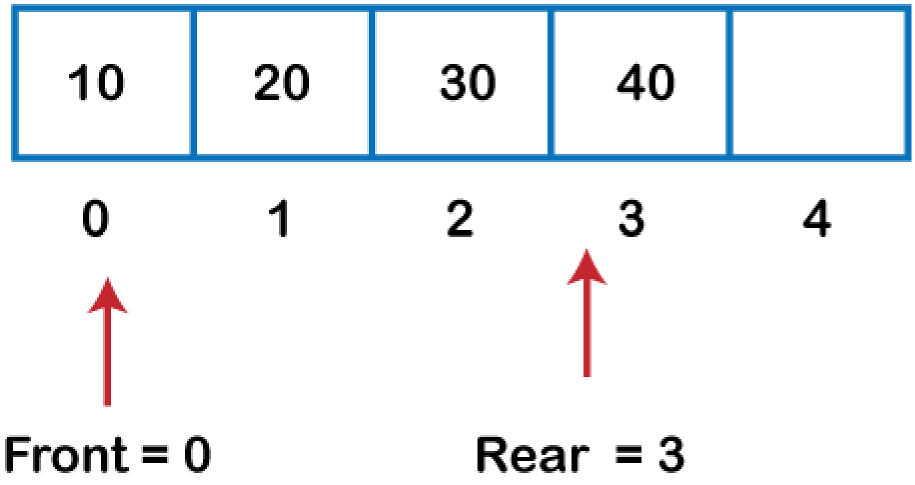
- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

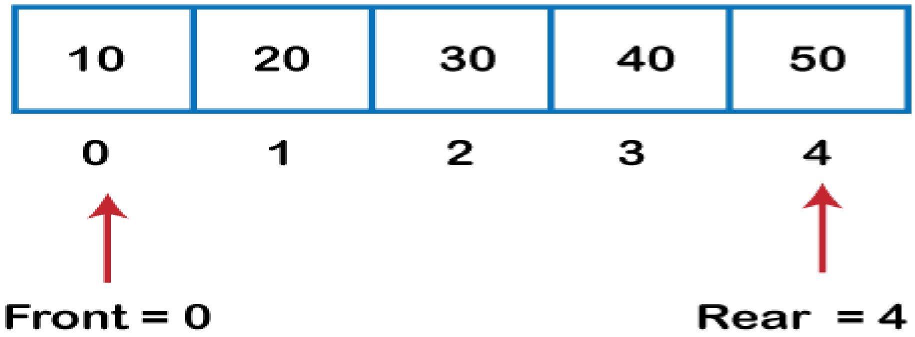
```
Step 1: IF FRONT = -1
    Write " UNDERFLOW "
    Goto Step 4
    [END of IF]

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
```

Let's understand the enqueue and dequeue operation through the diagrammatic representation.

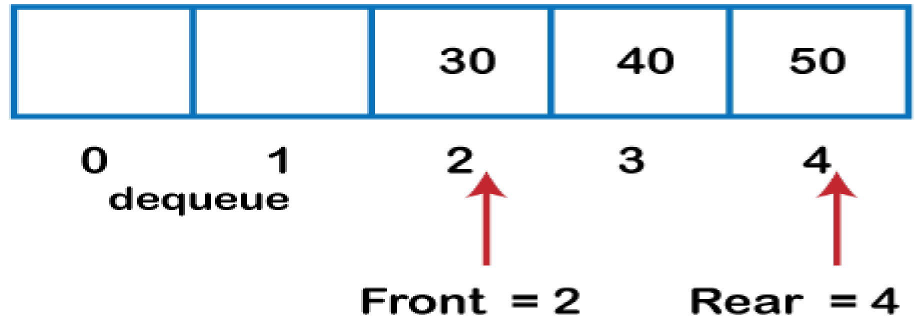




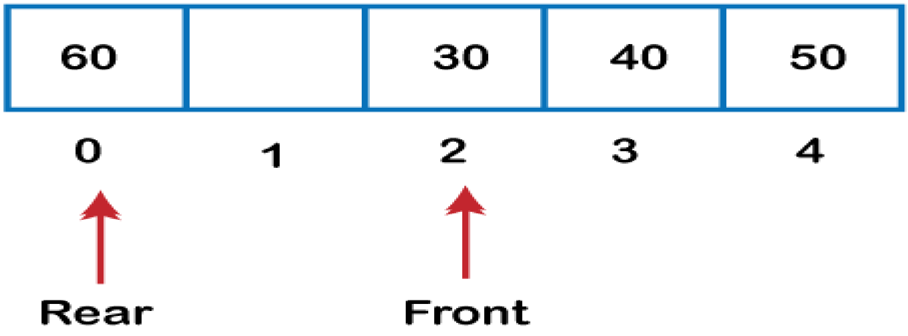
10
20
30
40
0
1
2
3
4
Front = 0
Rear = 3

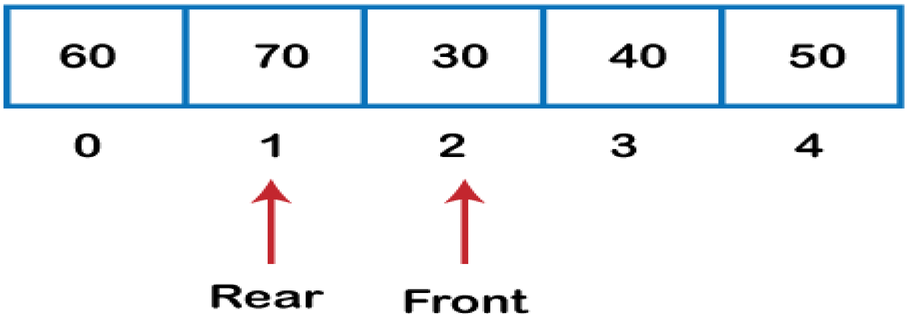


10
20
30
40
50
0
1
2
3
4
Front = 0
Rear = 4



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





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

A priority queue is an extension of a queue that contains the following characteristics:

- 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

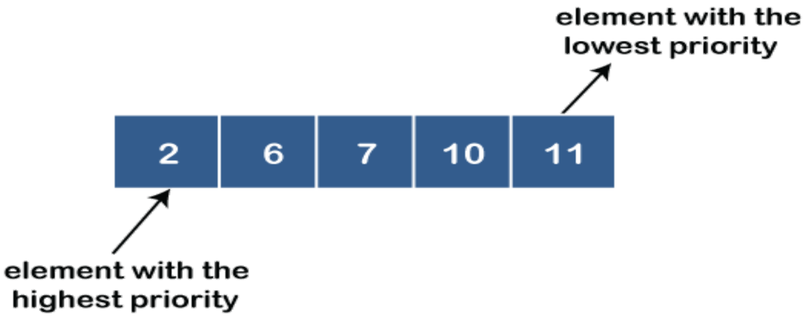
All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **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.

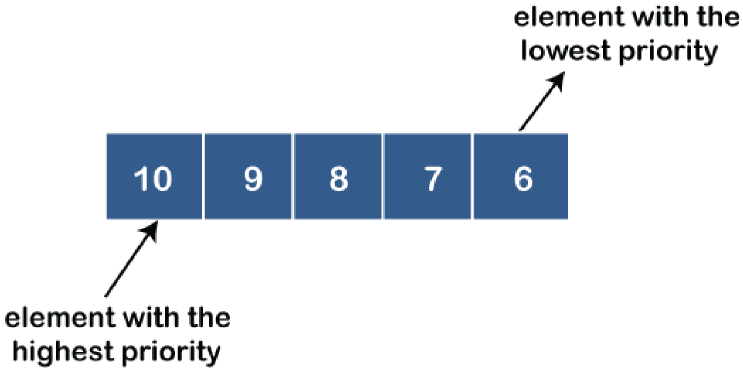
Types of Priority Queue

There are two types of priority queue:

- **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.



- **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.

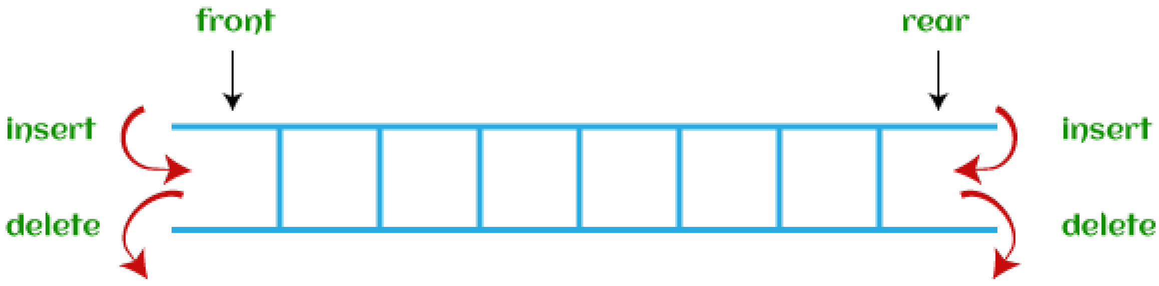


Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

The representation of the deque is shown in the below image -

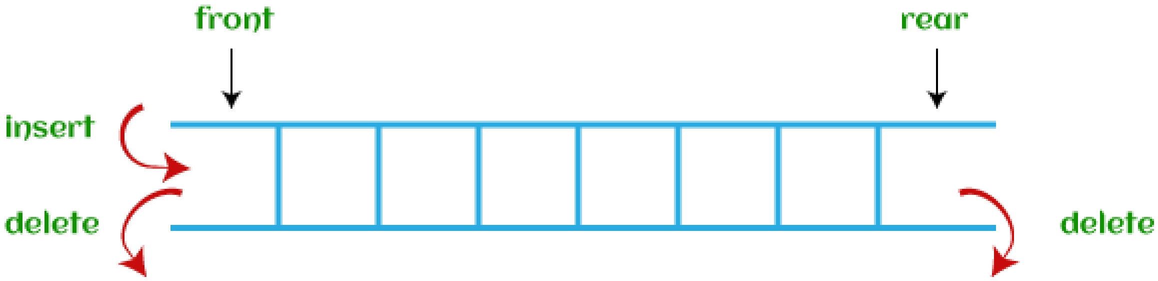


double ended queue

To know more about the deque, you can click the link - https://www.javatpoint.com/ds-deque

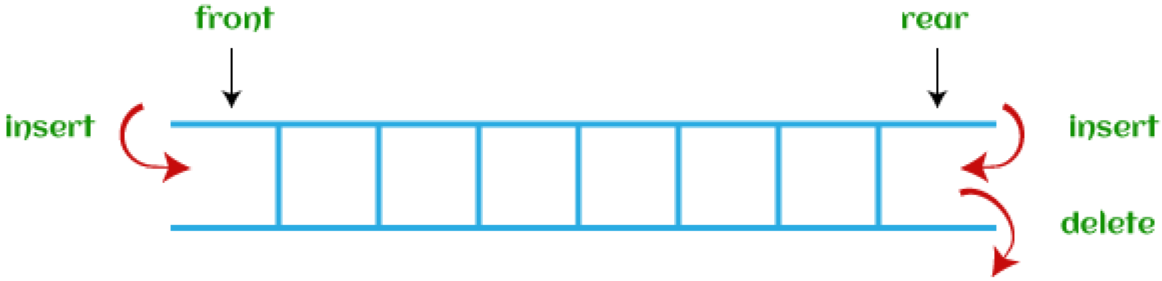
There are two types of deque that are discussed as follows -

- **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

- **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Now, let's see the operations performed on the queue.

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** It 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.
- **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):** It shows the overflow condition when the queue is completely full.
- **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

Now, let's see the ways to implement the queue.

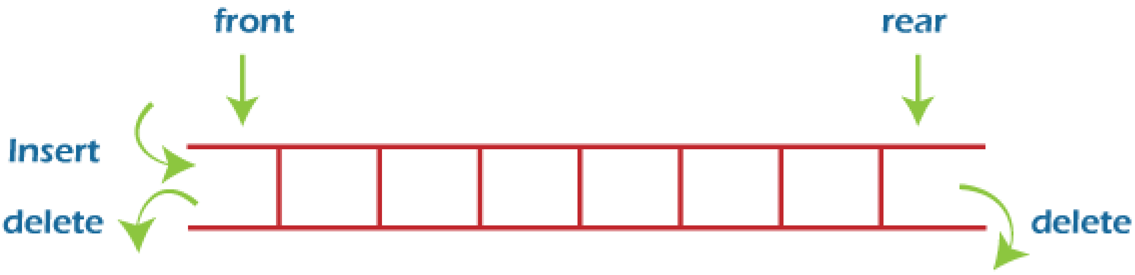
Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

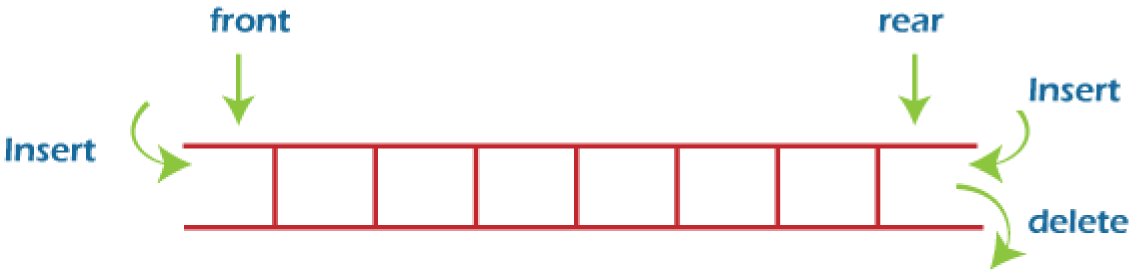
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Operations performed on deque

There are the following operations that can be applied on a deque -

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque -

- Get the front item from the deque

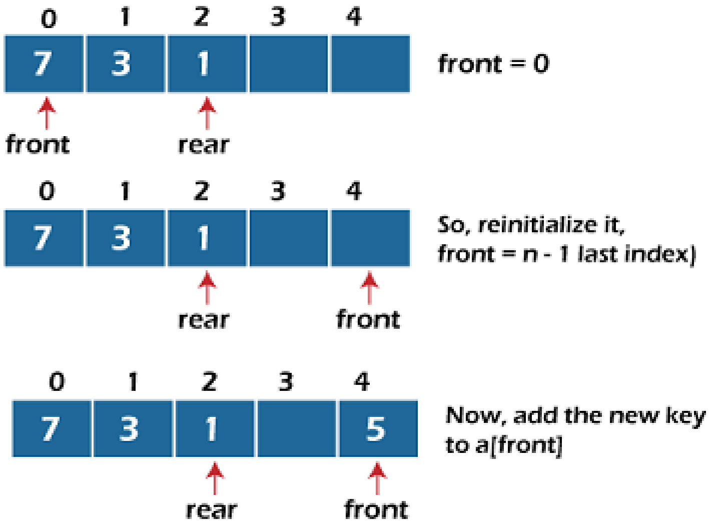
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

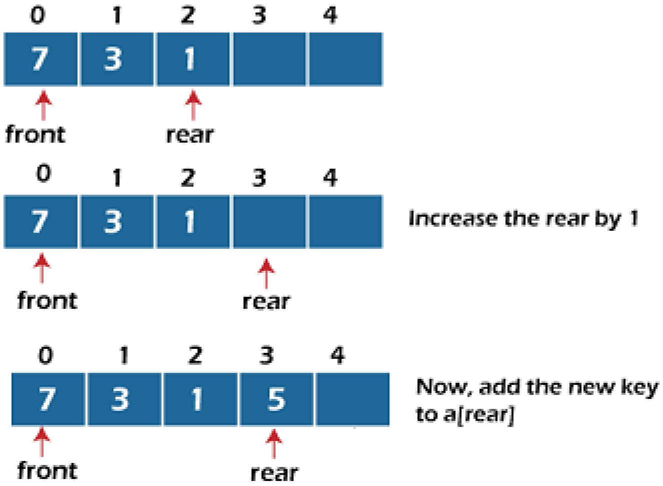
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

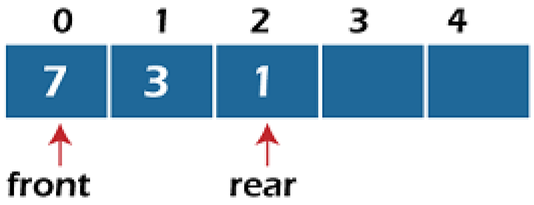
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

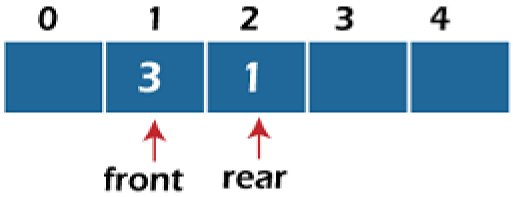
If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).





After deleting the element
7 front end

Deletion at the rear end

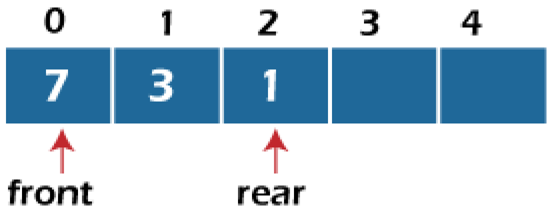
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

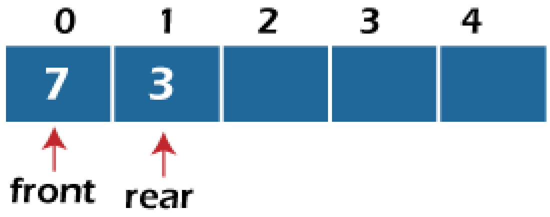
If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).





After deleting element 1
from rear end

Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

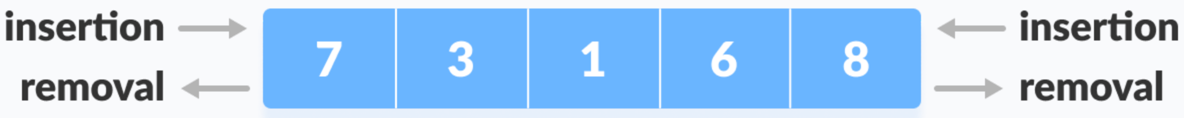
The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Applications of deque

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Deque Data Structure

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



Representation of Deque

Types of Deque

- **Input Restricted Deque**

  In this deque, input is restricted at a single end but allows deletion at both the ends.

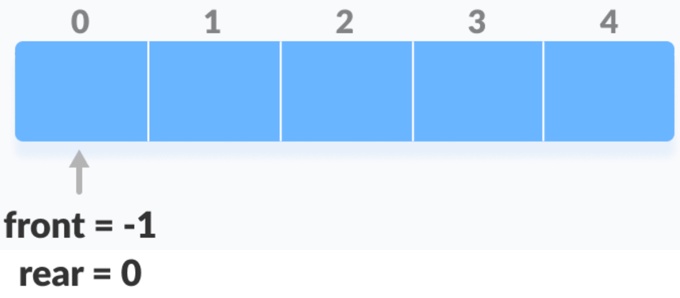
- **Output Restricted Deque**
  In this deque, output is restricted at a single end but allows insertion at both the ends.

Operations on a Deque

Below is the circular array implementation of deque. In a circular array, if the array is full, we start from the beginning.
But in a linear array implementation, if the array is full, no more elements can be inserted. In each of the operations below, if the array is full, "overflow message" is thrown.

Before performing the following operations, these steps are followed.

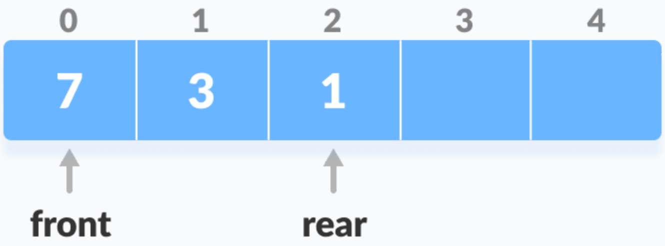
1. Take an array (deque) of size `n`.
2. Set two pointers at the first position and set `front = -1` and `rear = 0`.



Initialize an array and pointers for deque

1. Insert at the Front

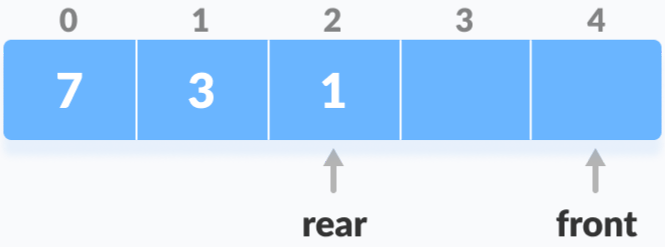
This operation adds an element at the front.



1. Check the position of front.

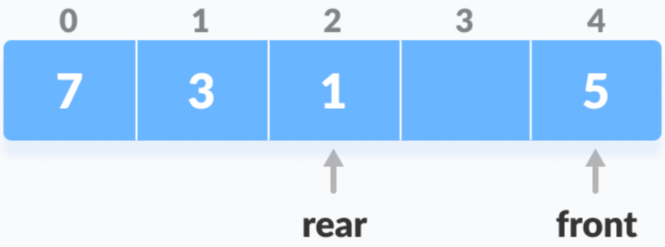
   Check the position of front

2. If `front < 1`, reinitialize `front = n-1` (last index).



Shift front to the end

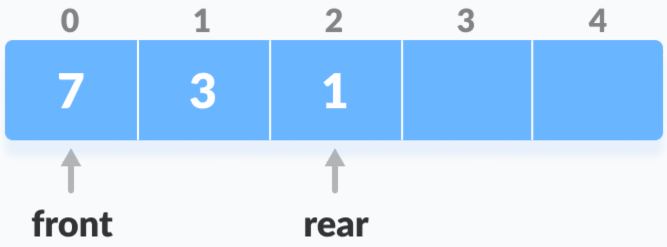
3. Else, decrease `front` by 1.
4. Add the new key `5` into `array[front]`.



Insert the element at Front

2. Insert at the Rear

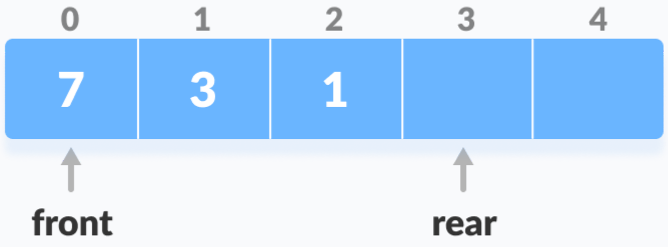
This operation adds an element to the rear.



1. Check if the array is full.

Check if deque is full

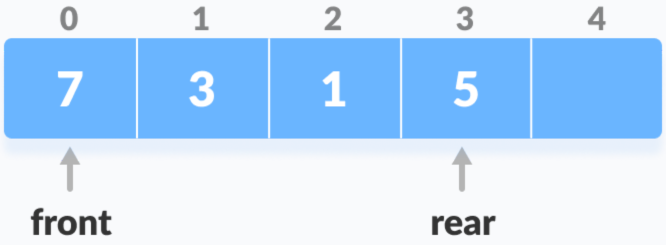
2. If the deque is full, reinitialize `rear = 0`.



3. Else, increase `rear` by 1.

Increase the rear

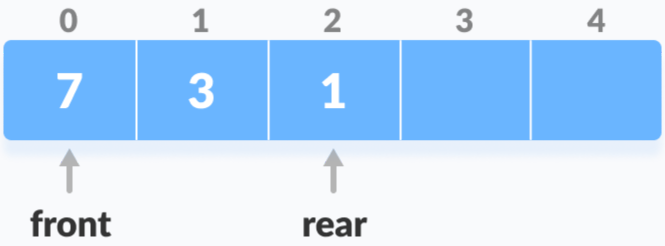
4. Add the new key `5` into `array[rear]`.



Insert the element at rear

3. Delete from the Front

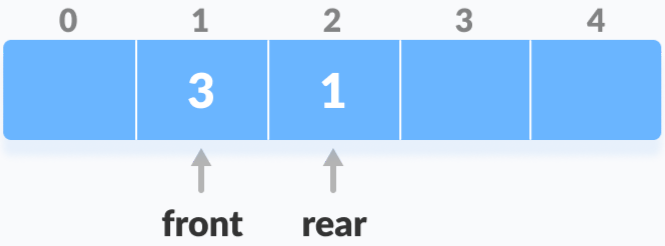
The operation deletes an element from the front.



1. Check if the deque is empty.

   Check if deque is empty

2. If the deque is empty (i.e. `front = -1`), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. `front = rear`), set `front = -1` and `rear = -1`.
4. Else if `front` is at the end (i.e. `front = n - 1`), set go to the front `front = 0`.

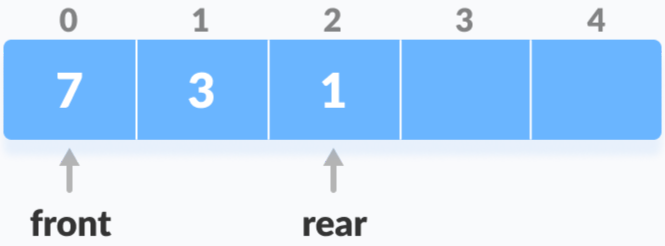


5. Else, `front = front + 1`.

   Increase the front

4. Delete from the Rear

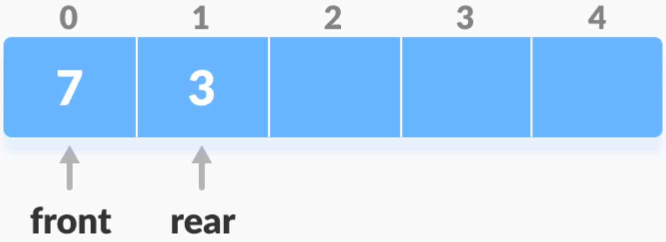
This operation deletes an element from the rear.



1. Check if the deque is empty.

   Check if deque is empty

2. If the deque is empty (i.e. `front = -1`), deletion cannot be performed (**underflow condition**).
3. If the deque has only one element (i.e. `front = rear`), set `front = -1` and `rear = -1`, else follow the steps below.
4. If `rear` is at the front (i.e. `rear = 0`), set go to the front `rear = n - 1`.



5. Else, `rear = rear - 1`. Decrease the rear