M-2 Linear Data Structures

* Abstract Data Types (ADT)

Data types are broadly classified into two : Built-in data types like int, float, double, long etc. and we can perform basic operations with them such as addition, subtraction, division etc. The other data type is user-defined data type, these can be defined along with their operations. such data structures that are not built-in are known as Abstract Data types.

Abstract Data Type (ADT) is a type of object whose behavior is defined by a set of values and set of operations. 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.

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

```
Interface Stack {
    public void push (string str);
        // add new element
    public string pop();
        // deletes element
}
```

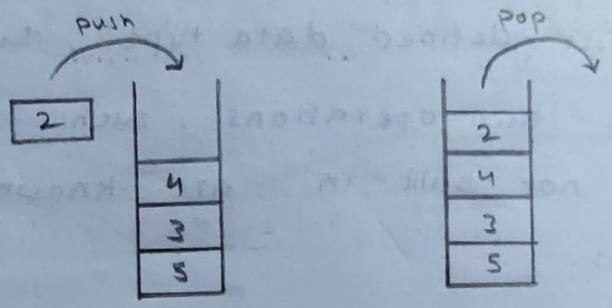
abstract methods

* Stack

A stack is an ordered list in which all insertions and deletions are made at one end called top.

principle : In stack always the last item to be put in to the stack is the first item to be removed. So stack is a **Last In First Out** or follows LIFO principle.

Representation of stacks

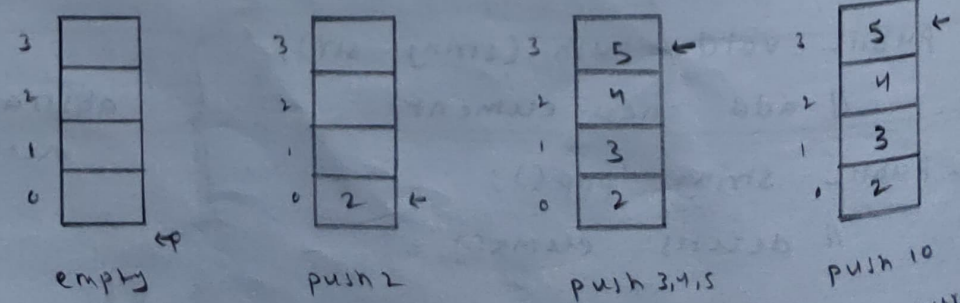


The pointer used in stack is called "top" with default value "-1" which is used to access the top element of the stack.

→ Stack Operations

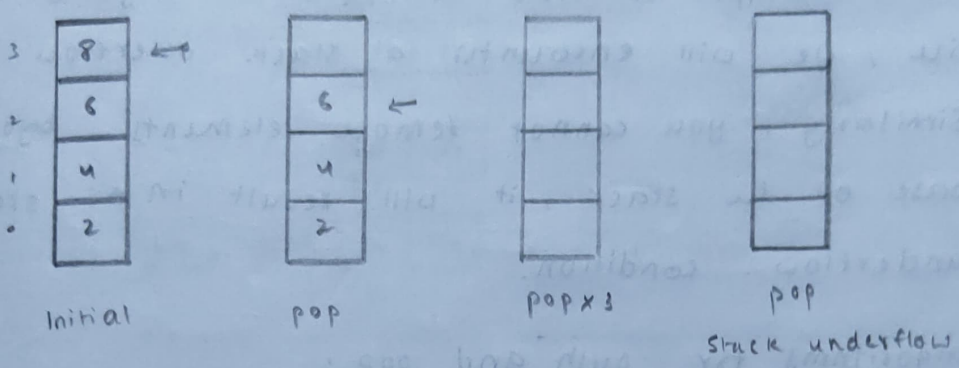
- **Push operation**

It is used to add new elements in to the top of the stack. At the time of adding first, its checked if the stack is full or not. Pushing a data element into an already full stack generally an error called "stack overflow". pointer is incremented.



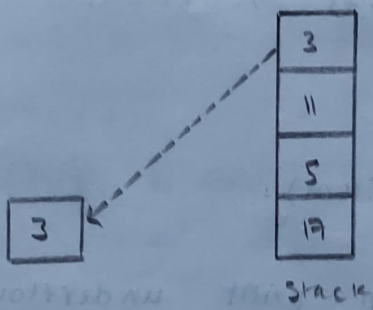
- Pop operation

It is used to delete elements from the top of the stack. At the time of deletion, first stack is checked if its empty. Deleting from an empty stack, it generates an error called "stack underflow". pointer top is decremented.



- Peek operation

This operation is used to retrieve the top element from the stack without deleting it.



- Display operation

Displays all the contents of the stack ADT.

- isEmpty() [boolean return type]

Returns true if stack is empty else returns false.

- isFull() [boolean return type]

Returns true if stack is full else returns false.

→ Representation of a stack using arrays

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack.

If we attempt to add new element beyond max. size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack, it will result in a stack underflow condition.

algorithms for push and pop :

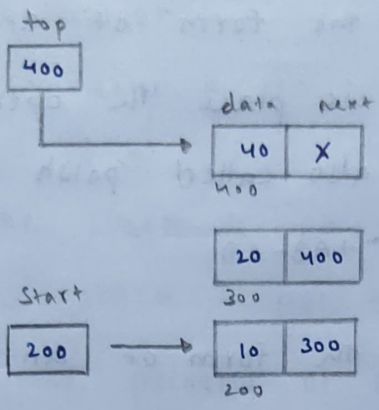
- Push (Stack, top, maxSize, item) :
  1. If top == maxSize, then print overflow & return.
  2. Step 2 set Stack[top] = item.
  3. Set top = top+1. (increment)
  4. Return

- Pop (Stack, top, item) :
  1. If Top = 0, then print underflow & return
  2. else set item = Stack [top].
  3. Set top = top-1. (increment)
  4. Return

- Peek (Stack, top, item) :
  1. return item = Stack [top].
  2. return

→ Implementation of stack using Linked List

We can represent stack as a linked list. In a stack, push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer.



→ Applications of stack

Some applications of stacks include the following:

- to check for balancing of paranthesis, brackets etc.
- to evaluate postfix expression.
- to convert infix to postfix / prefix form.
- to store intermediate args and return values in recursion.
- Depth First Search to find an element from a graph.

Algebraic expressions

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which mathematical operation is performed, it can be a variable (x, y, z ...) or a constant (5, 4, 6 ...). Operator is a symbol which signifies an operation. Algebraic expressions may be represented using 3 different notations: infix, prefix & postfix.

Infix notation: It is the form of an arithmetic expression in which we place the operator in between the two operands.

eg: (A+B) * (C - D) ; ((6 - (3+2) * 5) ^ 2 + 3)

Prefix notation: It is the form of an arithmetic notation in which we place the operator before its two operands. Its also called polish notation.

eg: +^-abc2 ; *+AB-CD

Postfix notation: It is the form of an arithmetic expression in which we place the operator after its two operands. Its also called reverse polish notation.

eg: 632+5*-2^3+ ; ab+c+2/

operator precedence: considering five binary operations: + (add), - (subtract), * (multiply), / (divide), ^ (exponent). We have the following order of precedence:

operator	precedence	value
^	highest	3
*, /	next highest	2
+, -	lowest	1

knowing all of these concepts, we can now use stacks to convert from one notation to another.

Conversion of Infix to Postfix

algorithm :

1. If the character is left paranthesis, push to stack.
2. If the character is an operand, add to the postfix expression.
3. If the character is an operator, check wether stack is empty.
   a) If the stack is empty, push opr into stack
   b) If the stack is not empty, check priority of opr.
      i) If the priority of scanned operator > operator present at the top of stack, then push operator into stack.
      ii) If the priority of scanned operator <= operator present at the top, then pop operator from stack and add to postfix expression and go to step ii)
4. If the character is right paranthesis, then pop all operator and operands from stack until it reaches left paranthesis and add to postfix exp.
5. After reading all characters, if stack is not empty then pop and add to postfix expression

following these steps on a given infix expression will give us an equivalent postfix notation of the algebraic expression.

Q. Convert ((A - (B + C)) * D) ↑ (E + F) to Postfix

Symbol	Postfix string	Stack
(		(
(		((
A	A	((
-	A	((-
(	A	((-(
B	A B	((-(
+	A B	((-(+
C	A B C	((-(+
)	A B C +	((-
)	A B C + -	(
*	A B C + -	(*
D	A B C + - D	(*
)	A B C + - D *	
↑	A B C + - D *	↑
(	A B C + - D *	↑(
E	A B C + - D * E	↑(
+	A B C + - D * E	↑(+
F	A B C + - D * E F	↑(+
)	A B C + - D * E F +	↑
	A B C + - D * E F + ↑	

∴ The equivalent postfix notation of the given infix algebraic expression ((A - (B+C)) * D) ↑ (E+F) is A B C + - D * E F + ↑

- Conversion of Infix to prefix

The precedence rules for converting an expression from infix to prefix are identical.

The only change from postfix conversion is that we traverse the expression from right to left and the operator is placed before the operands, rather than after them.

Q. Convert (A+B) * (C-D) into prefix expression.

Symbol	Prefix String	Stack
)		)
D	D	)
-	D	)-
C	(D	)-
(	-(D	
*	*-(D	*
)	-(D	*)
B	B-(D	*)
+	B-(D	*)+
A	AB-(D	*)+
(	+AB-(D	*
	*+AB-(D	

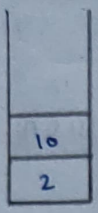
- Evaluation of postfix expression

algorithm:

1. When a number is seen, it is pushed into the stack.
2. When an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed into the stack.
3. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

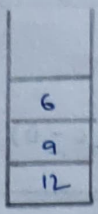
Q. Evaluate postfix expression : 2 10 + 9 6 - /

push 2
push 10

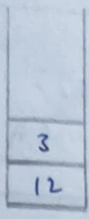


pop 2, 10
push 2+10=12

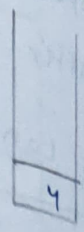
push 9
push 6



pop 9, 6
push 9-6=3



pop 3, 12
push 12/3=4



Evaluated answer of given expression is 4.

Q. Evaluate the postfix expression : 6 5 2 3 + 8 * + 3 + *

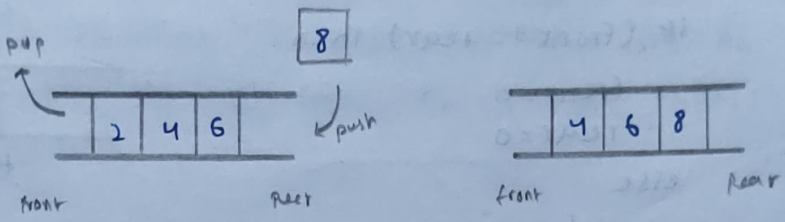
Symbol	operand 1	operand 2	value	stack
6				6
5				6, 5
2				6, 5, 2
3				6, 5, 2, 3
+	2	3	5	6, 5, 5
8				6, 5, 5, 8
*	5	8	40	6, 5, 40
+	5	40	45	6, 45
3				6, 45, 3
+	45	3	48	6, 48
*	6	48	288	288

* Queue

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.

Principle: Queue is referred to be as First In First Out List. It follows the FIFO principle.

Representation of Queues



The pointers used in Queues are front and Rear with initial value as -1. At enqueue, Rear is incremented and during dequeue, front is incremented.

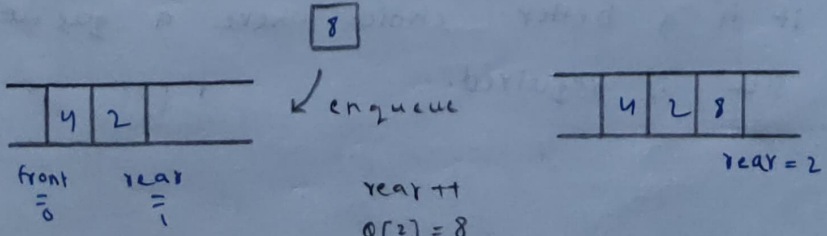
→ Queue operations

• Enqueue operation

It is used to add a new element to the rear end of the queue. Rear pointer is incremented.

algorithm:

```
If rear == size then,
    print "Queue is full"
    exit
else if (rear == 0) and (front == 0) then  // Queue is empty
    front = 1
rear = rear + 1
Q[rear] = item
```

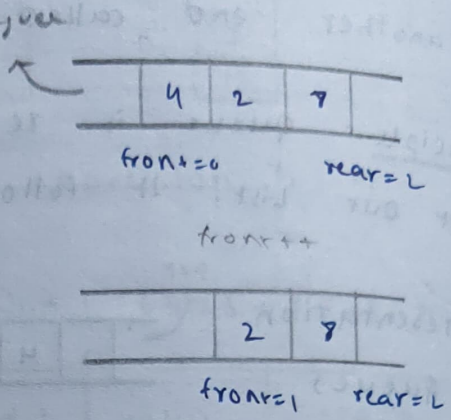


- Dequeue operation

It is used to remove an element from the front of the queue. Front pointer is incremented.

algorithm:

```
If (front == 0) then
    print "Queue is empty"
else
    item = queue[front]
    if (front == rear) then
        front = 0
        rear = 0
    else
        front = front + 1
```



- IsEmpty() : Checks if a queue is empty or not. Returns True if queue is empty and False if queue is not empty.
- peek() or front(): Acquires the data element available at the front node of the queue without deleting it.
- rear() : returns element at rear without deleting it.
- size() : This operation returns the size of the queue i.e. the total number of elements it contains.

→ Representation of Queues

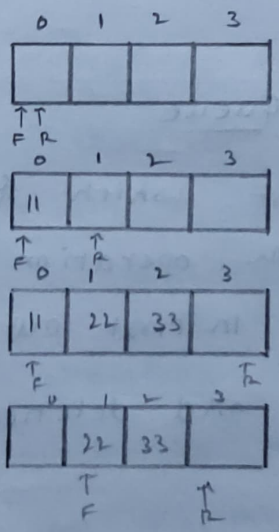
There are two ways to represent a queue in a memory using an array: It uses a one-dimensional array and it is a better choice where a queue of fixed size is required.

using a linked list : It uses a double linked list and provides a queue whose size can vary during processing.

Representation of Queues using array

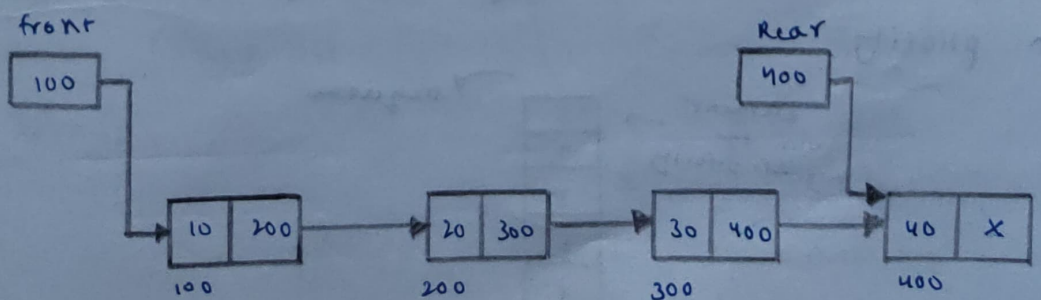
A one dimensional array, say Q[1...N], can be used to represent a queue. Two pointers, namely FRONT and REAR, are used to indicate the two ends of the queue.

Insertion of new element, the pointer REAR will be the consultant and for deletion the pointer FRONT will be the consultant.



Representation of Queues using Linked List

Queue can be represented using Linked List. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers front and rear for our linked queue implementation.



→ Applications of Queue

- It is used to schedule the jobs to be processed by the Central Processing Unit (CPU).
- When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out basis.
- Breadth first search uses a queue data structure to find an element from a graph.

---

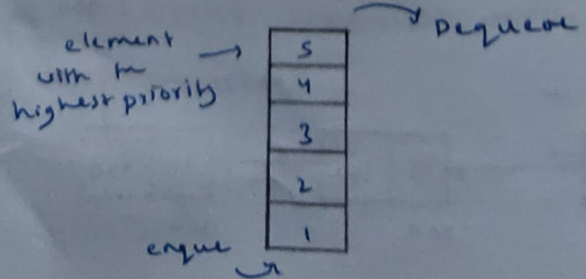
* Types of Queue

→ Simple Queue (or) Linear Queue

A queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out [FIFO]. Insertion occurs at rear end and deletion at front.

→ Priority Queue

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



→ Circular Queue

Circular queue is one in which the insertion of new element is done at the very first location of the queue if the last location of the queue is full.

Suppose if we have a queue of n elements then after adding th element at the last index i.e. (n-1)th, as queue is starting with 0 index, the next element will be inserted at the very first location of th queue which was not possible in simple linear queue.

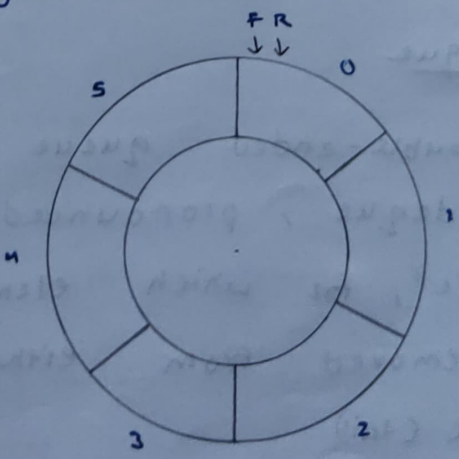
Th basic operations of a circular queue are:

- Insertion: Inserting an element into a circular queue results in Rear = (Rear +1) % max, where max is th maximum size of th array.
- Deletion: Deleting an element from a circular queue results in Front = (Front +1) % max, where max is the maximum size of th array.
- Traverse: Displaying the elements of a circular q.

Circular Queue is empty when Front = Rear = 0.

circular queues can be represented using:

1. Singly linked list
2. Doubly linked list
3. Arrays.



max = 6 Front = Rear = 0
Queue is empty

algorithm for enqueue in a circular queue:

```
if (front == (rear+1) / max_size)      // queue is full
    write queue over flow
else :
    take the value
    if (front == -1)                   // queue is empty
        set front = rear = 0
        queue [rear] = item
    else                               // queue is not empty
        rear = (rear+1) / max_size
        queue [rear] = item
```

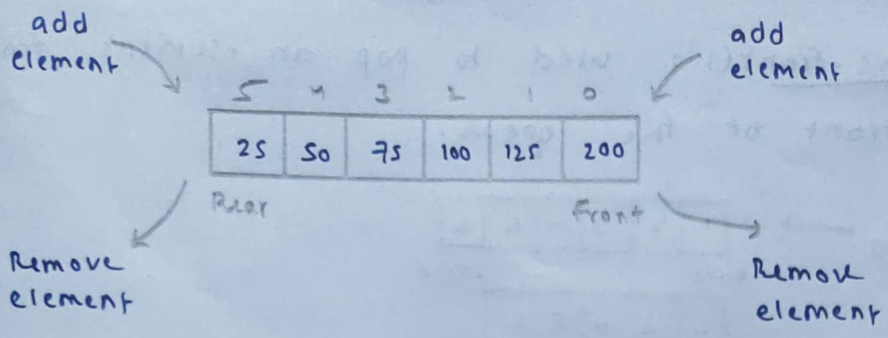
algorithm for dequeue in a circular queue:

```
if (front == -1)                       // queue is empty
    write queue is under flow
else
    item = queue [front]
    if (front == rear)                 // queue contains a single element
        set front = rear = -1
    else
        front = (front + 1) / max_size
```

→ Deque

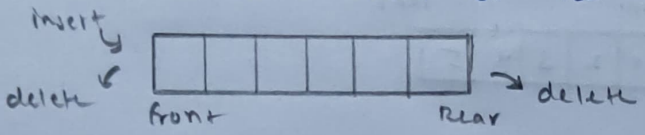
A double-ended queue (dequeue, often abbreviated to deque, pronounced deck) generalizing a queue, for which elements can be added to or removed from either the front (head) or back (tail)

It is also often called as head-tail linked list.

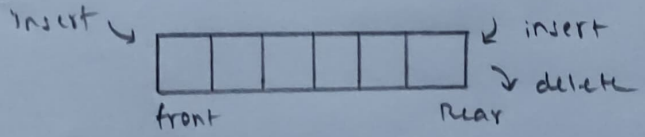


There are two types or variations of deque. They are:

- Input Restricted Deque (IRD): It is a deque, which allows insertion at one end but allows deletions at both ends of the queue.

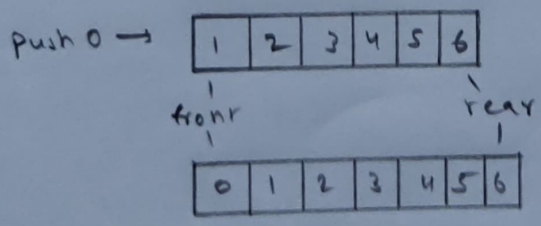


- Output Restricted Deque (ORD): It is a deque, which allows deletion at one end but allows insertions at both ends of the queue.

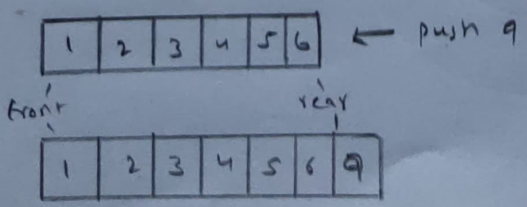


Deque Operations:

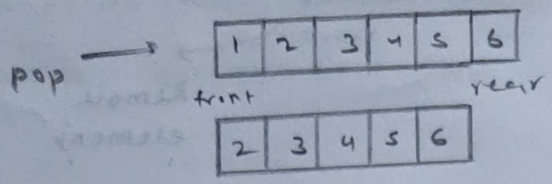
- Insert_front() - used to push an element to the front of the deque.



- insert_back(): used to push an element to the back.



- Remove-front(): used to pop an element from the front of the queue.



- Remove-back(): used to pop an element from the back of the queue.