Module -2

Linear Data Structures and their Sequential Storage Representation

Teaching Hours: 10

1. Stack

A stack is an abstract data type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example − deck of cards or pile of plates etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, We can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

1.1 Operations and Applications

Basic Operations Performed on Stack :

1. Create
2. Push
3. Pop
4. Empty

A. Creating Stack :

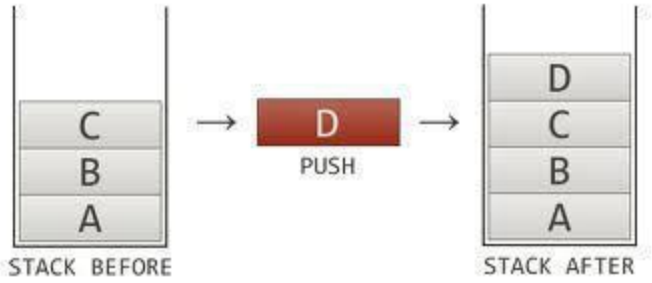
1. Stack can be created by declaring the structure with two members.
2. One Member can store the actual data in the form of array.
3. Another Member can store the position of the topmost element.

```
typedef struct stack {
  int data[MAX];
  int top;
```

```
}stack;
```

B. Push Operation on Stack :

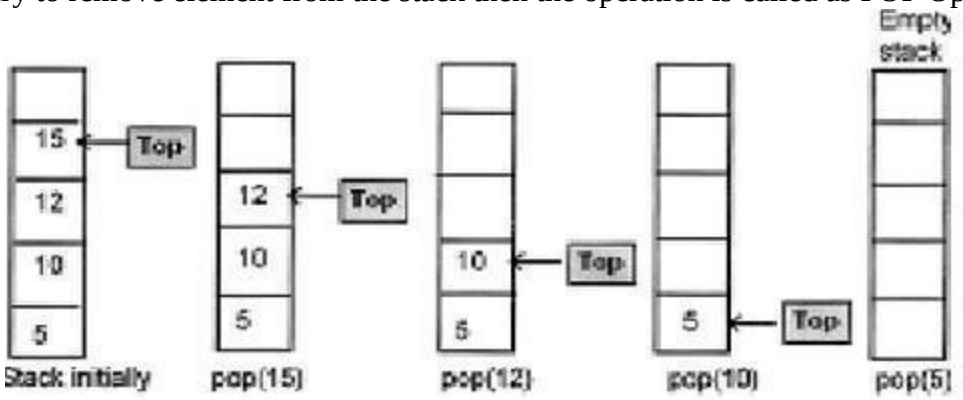
We have declared data array in the above declaration. Whenever we add any element in the _data‘ array then it



will be called as **“Pushing Data on the Stack”**. Suppose ―top‖ is a pointer to the top element in a stack. After every push operation, the value of ―top‖ is incremented by one.

C. Pop Operation on Stack :

Whenever we try to remove element from the stack then the operation is called as POP Operation on Stack.



Some basic Terms :

Concept	**Definition**
Stack Push	The procedure of inserting a new element to the top of the stack is known as **Push Operation**
Stack Overflow	Any attempt to insert a new element in already full stack is results into Stack Overflow.
Stack Pop	The procedure of removing element from the top of the stack is

Concept	Definition
	called **Pop Operation**.
Stack Underflow	Any attempt to delete an element from already empty stack results into Stack Underflow.

Application

1. Expression Evolution
2. Expression conversion
    1. Infix to Postfix
    2. Infix to Prefix
    3. Postfix to Infix
    4. Prefix to Infix
3. Parsing
4. Simulation of recursion
5. Fuction call

1.2 Polish and reverse polish expressions

Reverse Polish notation (RPN) is a mathematical notation in which every operator follows all of its operands, in contrast to Polish notation(PN), which puts the operator before its operands. It is also known as postfix notation and does not need any parentheses as long as each operator has a fixed number of operands.

1.3 Infix to postfix conversion

nfix to postfix conversion algorithm

There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

In this algorithm, all operands are printed (or sent to output) when they are read. There are more complicated rules to handle operators and parentheses.

Example:

1. A * B + C becomes A B * C +

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

The rule used in lines 1, 3 and 5 is to print an operand when it is read. The rule for line 2 is to push an operator onto the stack if it is empty. The rule for line 4 is if the operator on the top of the stack has higher precedence than the one being read, pop and print the one on top and then push the new operator on. The rule for line 6 is that when the end of the expression has been reached, pop the operators on the stack one at a time and print them.

2. A + B * C becomes A B C * +

Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.

	current symbol	operator stack	postfix string
1	A		A
2	+	+	A

3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

In line 4, the '*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when the are both popped off in lines 6 and 7, their order will be reversed.

3. A * (B + C) becomes A B C + *

A subexpression in parentheses must be done before the rest of the expression.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A B
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	)	*	A B C +
8			A B C + *

Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker. When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

4. A - B + C becomes A B - C +

When operators have the same precedence, we must consider association. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

	current symbol	operator stack	postfix string
1	A		A
2	-	-	A
3	B	-	A B
4	+	+	A B -
5	C	+	A B - C
6			A B - C +

In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5. A * B ^ C + D becomes A B C ^ * D +

Here both the exponentiation and the multiplication must be done before the addition.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack. Since it has lower precedence, the '^' is popped and printed. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '*'. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6. A * (B + C * D) + E becomes A B C D * + * E +

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

1.4 Evaluation of postfix expression

In normal algebra we use the infix notation like a+b*c. The corresponding postfix notation is abc*+. The algorithm for the conversion is as follows :

- Scan the Postfix string from left to right.
- Initialise an empty stack.
- If the scannned character is an operand, add it to the stack. If the scanned character is an operator, there will be atleast two operands in the stack.
    - If the scanned character is an Operator, then we store the top most element of the stack(topStack) in a variable temp. Pop the stack. Now evaluate topStack(Operator)temp. Let the result of this operation be retVal. Pop the stack and Push retVal into the stack.

  Repeat this step till all the characters are scanned.

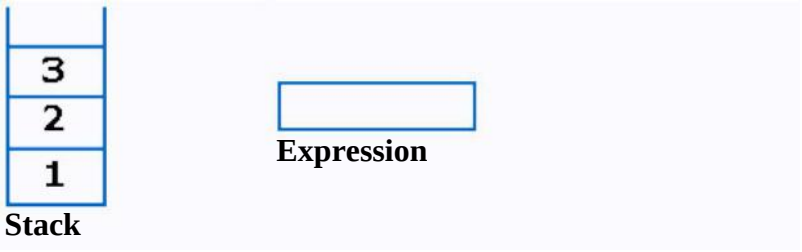
- After all characters are scanned, we will have only one element in the stack. Return topStack.

Example :

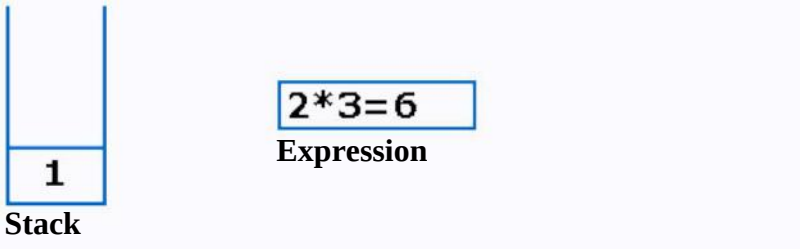
Let us see how the above algorithm will be imlemented using an example.

Postfix String : 123*+4-

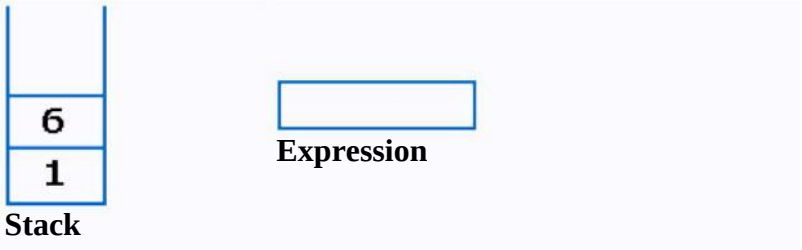
Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are operands. Thus they will be pushed into the stack in that order.



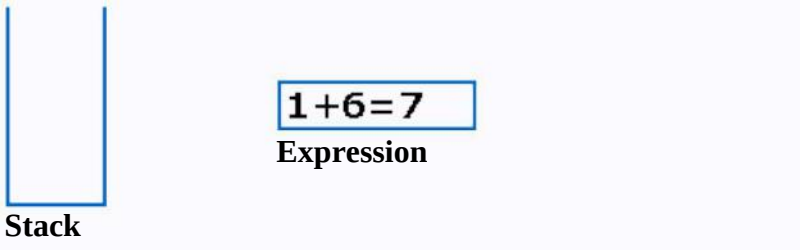
Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(2*3) that has been evaluated(6) is pushed into the stack.

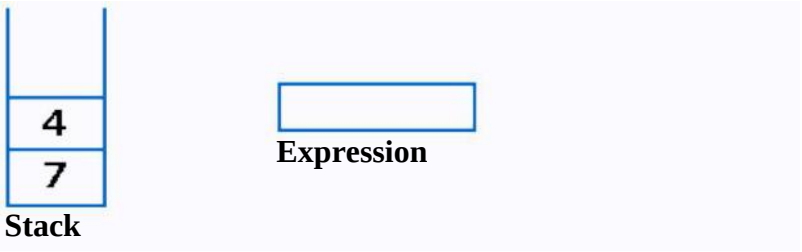


Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.

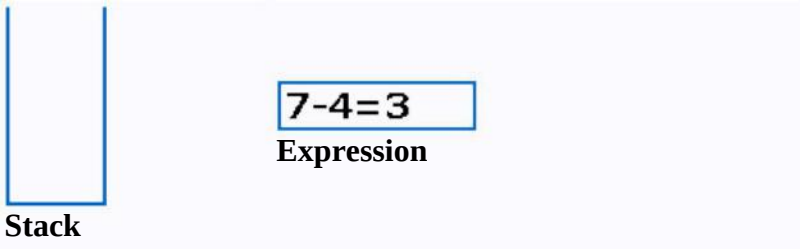


The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.

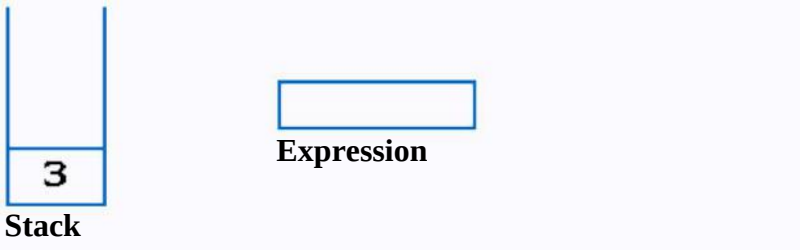
Next character scanned is "4", which is added to the stack.



Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(7-4) that has been evaluated(3) is pushed into the stack.



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.

End result :

- Postfix String : 123*+4-
- Result : 3

1.5 Infix to prefix

1. Step 1. Push —)‖ onto STACK, and add —(— to end of the A
2. Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
3. Step 3. If an operand is encountered add it to B
4. Step 4. If a right parenthesis is encountered push it onto STACK
5. Step 5. If an operator is encountered then:
6. a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same
7. or higher precedence than the operator.
8. b. Add operator to STACK
9. Step 6. If left parenthesis is encontered then
10. a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd)
11. b. Remove the left parenthesis
12. Step 7. Exit

1.6 Postfix to infix conversion

1. While there are input symbol left
2. Read the next symbol from input.
3. If the symbol is an operand

4. Otherwise,
   the symbol is an operator.
5. If there are fewer than 2 values on the stack
   Show Error /* input not sufficient values in the expression */
6. Else
   Pop the top 2 values from the stack.
   Put the operator, with the values as arguments and form a string.
   Encapsulate the resulted string with parenthesis.
   Push the resulted string back to stack.
7. If there is only one value in the stack

8. If there are more values in the stack
   Show Error /* The user input has too many values */

2. Recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration). The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.

2.1 Factorial

The factorial of a positive number n is given by:

factorial of n (n!) = 1*2*3*4....n

The factorial of a negative number doesn't exist. And, the factorial of 0 is 1.

You will learn to find the factorial of a number using recursion in this example. Visit this page to learn, how you can find the factorial of a number using loop .

Example: Factorial of a Number Using Recursion

```
#include <stdio.h>
long int multiplyNumbers(int n);


int main()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    printf("Factorial of %d = %ld", n, multiplyNumbers(n)); return
    0;
}
long int multiplyNumbers(int n)
{
    if (n >= 1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}
```

2.2 GCD

```
#include <stdio.h>

int hcf(int n1, int n2);

int main()

{

   int n1, n2;

   printf("Enter two positive integers: ");

   scanf("%d %d", &n1, &n2);

   printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2)); return

   0;

}

int hcf(int n1, int n2)

{

    if (n2!=0)

       return hcf(n2, n1%n2);

    else

       return n1;

}
```

2.3 Fibonacci Sequence

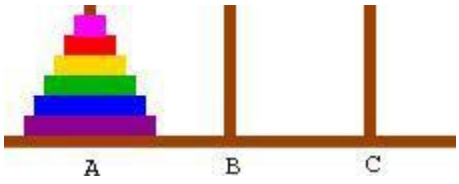
```
#include<stdio.h>

void printFibonacci(int);

int main(){

   int k,n;
   long int i=0,j=1,f;

   printf("Enter the range of the Fibonacci series: ");
   scanf("%d",&n);

   printf("Fibonacci Series: ");
   printf("%d %d ",0,1);
   printFibonacci(n);

   return 0;
}

void printFibonacci(int n){

   static long int first=0,second=1,sum;

   if(n>0){
       sum = first + second;
       first = second;
       second = sum;
       printf("%ld ",sum);
       printFibonacci(n-1);
   }

}
```

2.4 Tower of Hanoi

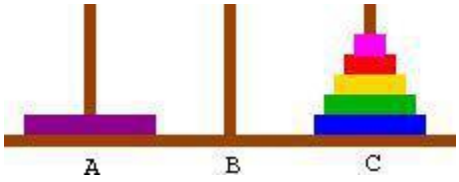
Using recursion often involves a key insight that makes everything simpler. Often the insight is determining what data exactly we are recursing on - we ask, what is the essential feature of the problem that should change as we call ourselves? In the case of isAJew, the feature is the person in question: At the top level, we are asking about a person; a level deeper, we ask about the person's mother; in the next level, the grandmother; and so on.

In our Towers of Hanoi solution, we recurse on the largest disk to be moved. That is, we will write a recursive function that takes as a parameter the disk that is the largest disk in the tower we want to move. Our function will also take three parameters indicating from which peg the tower should be moved (*source*), to which peg it should go (*dest*), and the other peg, which we can use temporarily to make this happen (*spare*).

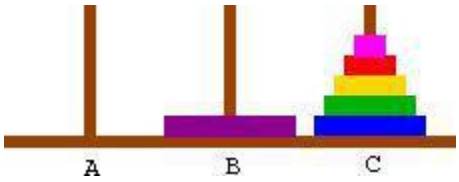
At the top level, we will want to move the entire tower, so we want to move disks 5 and smaller from peg A to peg B. We can break this into three basic steps.



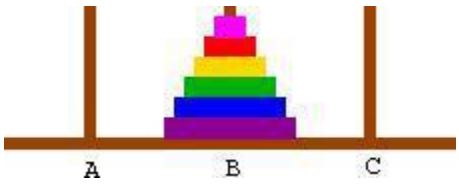
1. Move disks 4 and smaller from peg A (*source*) to peg C (*spare*), using peg B (*dest*) as a spare. How do we do this? By recursively using the same procedure. After finishing this, we'll have all the disks smaller than disk 4 on peg C. (Bear with me if this doesn't make sense for the moment - we'll do an example soon.)



2. Now, with all the smaller disks on the spare peg, we can move disk 5 from peg A (*source*) to peg B (*dest*).



3. Finally, we want disks 4 and smaller moved from peg C (*spare*) to peg B (*dest*). We do this recursively using the same procedure again. After we finish, we'll have disks 5 and smaller all on*dest*.



2.5 Binomial Co-efficient(*nCr*)

The binomial coefficient $\binom{n}{k}$ is the number of ways of picking k *unordered* outcomes from n possibilities, also known as a combination or combinatorial number. The symbols ${}_nC_k$ and $\binom{n}{k}$ are used to denote a binomial coefficient, and are sometimes read as "n choose k."

$\binom{n}{k}$ therefore gives the number of k-subsets possible out of a set of n distinct items. For example, The 2-subsets of $\{1, 2, 3, 4\}$ are the six pairs $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, and $\{3, 4\}$, so $\binom{4}{2} = 6$. The number of lattice paths from the origin $(0, 0)$ to a point (a, b) is the binomial coefficient $\binom{a+b}{a}$ (Hilton and Pedersen 1991).

The value of the binomial coefficient for nonnegative n and k is given explicitly by

$$_nC_k \equiv \binom{n}{k} \equiv \frac{n!}{(n-k)!\,k!},$$

2.6 Ackerman's Recursive function

The Ackermann function is the simplest example of a well-defined total function which is computable but not primitive recursive, providing a counterexample to the belief in the early 1900s that every computable function was also primitive recursive (Dötzel 1991). It grows faster than an exponential function, or even a multiple exponential function.

The Ackermann function $A(x, y)$ is defined for integer x and y by

$$A(x, y) \equiv \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1, 1) & \text{if } y = 0 \\ A(x-1, A(x, y-1)) & \text{otherwise.} \end{cases} \quad (1)$$

Special values for integer x include

$$A(0, y) = y + 1 \quad (2)$$

$$A(1, y) = y + 2 \quad (3)$$

$$A(2, y) = 2y + 3 \quad (4)$$

$$A(3, y) = 2^{y+3} - 3 \quad (5)$$

$$A(4, y) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{y+3} - 3. \quad (6)$$

Expressions of the latter form are sometimes called power towers. $A(0, y)$ follows trivially from the definition. $A(1, y)$ can be derived as follows:

$$A(1, y) = A(0, A(1, y-1)) \quad (7)$$

$$= A(1, y-1) + 1 \quad (8)$$

$$= A(0, A(1, y-2)) + 1 \quad (9)$$

$$= A(1, y-2) + 2 \quad (10)$$

$$= \ldots \quad (11)$$

$$= A(1, 0) + y \quad (12)$$

$$= A(0, 1) + y = y + 2.$$

3. Queue

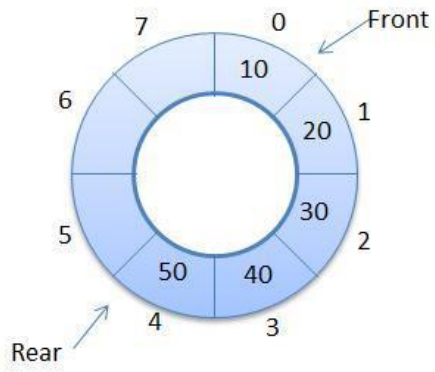
3.1 Operations

a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position, known as *enqueue*, and removal of entities from the front terminal position, known as *dequeue*. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed. Often a *peek* or *front* operation is also entered, returning the value of the front element without dequeuing it. A queue is an example of a linear data structure, or more abstractly a sequential collection.

3.2 Queue Variants

Circular Queue

A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue. Circular queues have a fixed size. Circular queue follows FIFO principle. Queue items are added at the rear end and the items are deleted at front

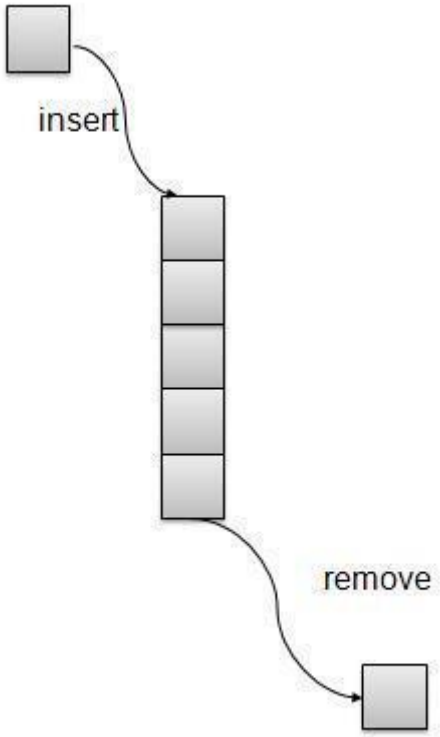


end of the circular queue.

Algorithm for Insertion in a circular queue

```
1.
2.  Insert CircularQueue ( )
3.
4.  1. If (FRONT == 1 and REAR == N) or (FRONT == REAR + 1) Then
5.
6.
7.
8.  3. Else
9.
10. 4.If (REAR == 0) Then [Check if QUEUE is empty]
11.
12.                 (a) Set FRONT = 1
13.
14.                 (b) Set REAR = 1
15.
16. 5. Else If (REAR == N) Then [If REAR reaches end if QUEUE]
17.
18. 6.              Set REAR = 1
19.
20. 7. Else
21.
22. 8.              Set REAR = REAR + 1 [Increment REAR by 1]
23.
24.   [End of Step 4 If]
25.
26. 9. Set QUEUE[REAR] = ITEM
27.
28. 10. Print: ITEM inserted
29.
30. [End of Step 1 If]
31.
32. 11. Exit
```

Priority Queue

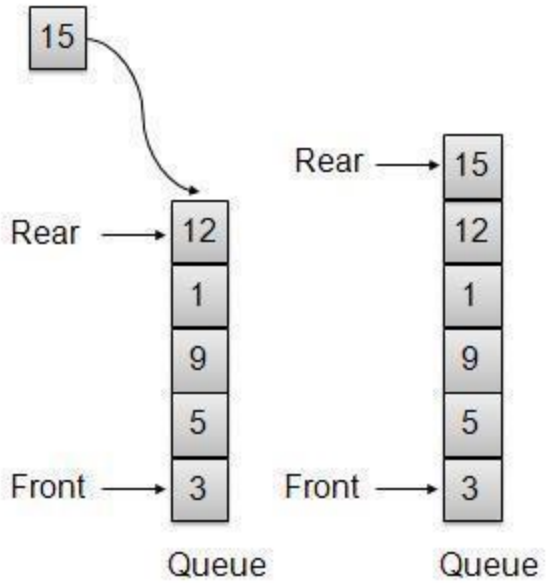


We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

Insert / Enqueue Operation

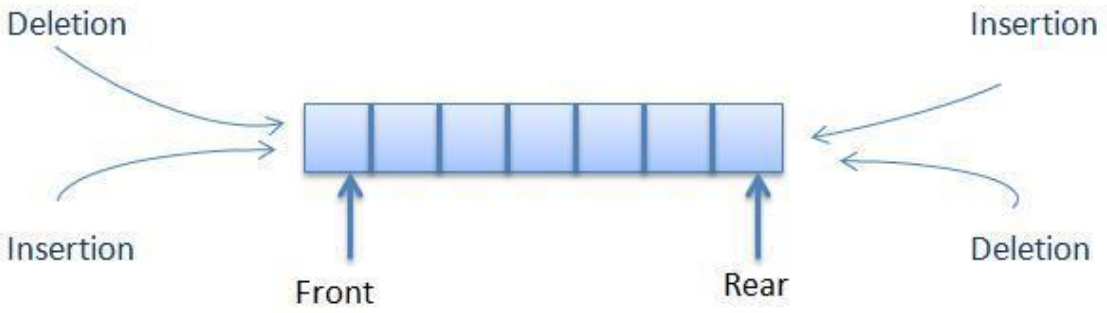
Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



One item inserted at rear end

Double Ended Queue

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.



Algorithm for Insertion at rear end

1.
2. Step -1: [Check for overflow]

```
3.
4.          if(rear==MAX)
5.
6.                  Print("Queue is Overflow‖);
7.
8.      return;
9.
10. Step-2: [Insert element]
11.
12.         else
13.
14.                 rear=rear+1;
15.
16.                 q[rear]=no;
17.
18.                 [Set rear and front pointer]
19.
20.                 if rear=0
21.
22.                         rear=1;
23.
24.                 if front=0
25.
26.                         front=1;
27.
28. Step-3: return
29.
```

3.3 Applications of Queues

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.