MODULE-3

Stacks and Queues

In definite instances in computers sciences, it is preferable to limit insertion and deletion to the beginning or end of the list, rather than in the center. Stacks and queues are two illustration of useful data structures.

Linear lists and arrays allow you to insert and delete elements at any point in the list, whether at the beginning, end, or center.

STACK:

A stack is a list of elements in which a member can only be added or removed at one end, known as the topmost of the stack. Stack is also referred to as LIFO (last in, first out) list. Because items can only be add or remove from the top, the final item added to a stack is the first item remove.

Stacks are associated with two basic operations:

The phrase "push" refers to inserting an element into a stack.

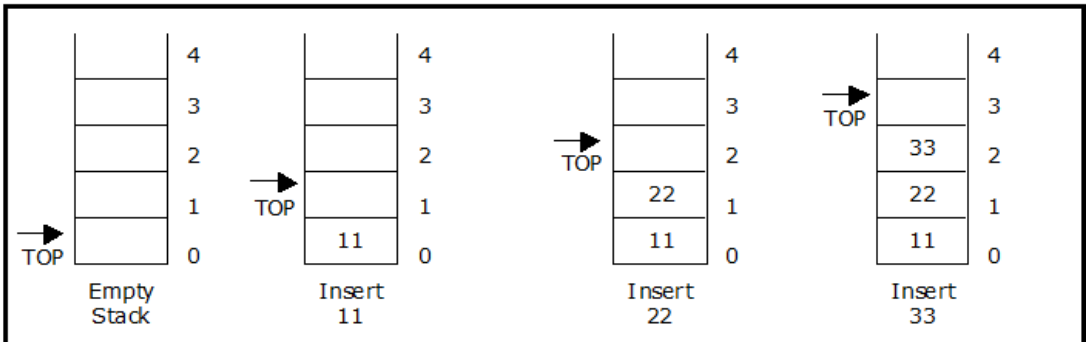
The term "pop" Mention to the act of removes an element from a stack.

Because all insertions and deletions occur at the same end of the stack, the final piece add to the stack is also the first element remove from the stacks. When you create a stack, the stacks base remains constant while the stack top changes as elements are add and remove. The top of the stack is the most accessible, and the bottom is the least accessible.

Stack Representation:

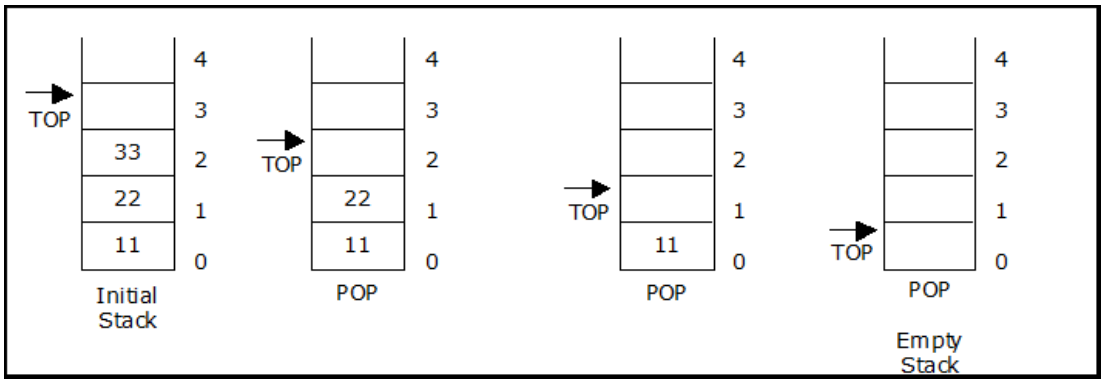
Consider a stacks with a content of six items. This is referred to as stack size. The number of components to be add shouldn't exceed the stack's maximal size. If we try to add a new element that is larger than the maximal sizing, we will hit a stack overrun problem. Similarly, you cannot delete pieces beyond the stack's base. If this occurs, we will encounter a stack underflow problem.

Push() performs the activity of adds an element to a stack.



Pushing operation on Stack

Pop() performs the activity of removes an element from the stack.



Poping operation on Stack

Stacks operation using Array implementation:

```
# include <stdio.h> # include <conio.h>
# include <stdlib.h># define MAX 6
int stack[MAX];
int top = 0;
int menu()
{
        int ch;
        clrscr();
        printf("\n … Stack operations using ARRAY... ");
        printf("\n ----------**********----------\n");
        printf("\n 1. Push ");   printf("\n 2. Pop ");    printf("\n 3. Display");
        printf("\n 4. Quit ");   printf("\n Enter your choice: ");
        scanf("%d", &ch);       return ch;
}
void display()
{
        int i;    if(top == 0)
        {
                printf("\n\nStack empty..");              return;
        }
        else
        {
                printf("\n\nElements in stack:");
                for(i = 0; i < top; i++)
                        printf("\t%d", stack[i]);
        } }
```

```
void pop()
{
        if(top == 0)
        {
                printf("\n\nStack Underflow..");                return;
        }
        else
                printf("\n\npopped element is: %d ", stack[--top]);
}
void push()
{
        int data;       if(top == MAX)
        {
                printf("\n\nStack Overflow..");                return;
        }
        else
        {
                printf("\n\nEnter data: ");                scanf("%d", &data);
                stack[top] = data;              top = top + 1;
                printf("\n\nData Pushed into the stack");
        }
}

void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                push();
                                break;
                        case 2:
                                pop();
                                break;
                        case 3:
                                display();
                                break;

                        case 4:
                                exit(0);
                }
                getch();
        } while(1);
}
```

This C program uses an array to implement basic stack operations. The stack is a data structures that adheres to the Last In First Out (LIFO) principle, which states that the last piece adding to the stack will be the first to be separate.

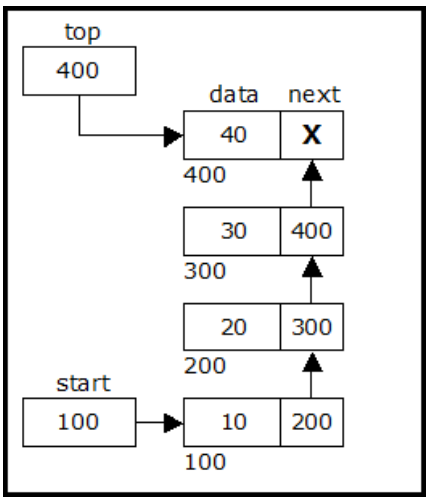
Here's a rundown of the operations:

1. **push():** This function inserts a new element onto the stack. It checks for stack overflow (when the stack is full) and allows the user to enter data, which is subsequently placed to the top of the stack if the stack is not full.
2. **pop():** This function is used to remove the stack's topmost member. It checks for stack underflow (when the stack is blank) and eliminates the top component if the stack is not empty.
3. **display():** This function shows all of the components in the stack, from bottom to top.
4. **menu():** Displays a basic menu to the user, allowing them to select from several stack operations (push, pop, display, and quit).

The main() function executes an indefinite loop (do-while loop) to continue displaying the menu to the user until the user chooses to stop (case 4). It takes the user's selection and performs the corresponding operation within the loop.

Linked List Execution of Stack:

A stack can be described as a linked list. Push and pop activities are done at the topmost of a stack. Using the top pointer, we may execute comparable operations at one end of the list.



linked list Implementation

Stacks operations using linked lists Implementation:

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>

struct stack
{
        int data;
        struct stack *next;
};

typedef struct stack node;
node *start=NULL;
node *top = NULL;

node* getnode()
{
        node *temp;
        temp=(node *) malloc( sizeof(node)) ;
        printf("\n Enter the data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void push(node *newnode)
{
        node *temp;
        if(start == NULL)
        {
                start = newnode;

                top = newnode;
        }
        else
        {
                temp = start;
                while( temp -> next != NULL)
                        temp = temp -> next;
                temp -> next = newnode;
                top = newnode;
        }
        printf("\n\n\t Data pushed into stack");
}
void pop()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t Stack underflow");
                return;
        }
        temp = start;
        if( start -> next == NULL)
        {
                printf("\n\n\t Popped element is %d ", top -> data);
                start = NULL;
                free(top);
                top = NULL;
        }
        else
        {
```

```
                while(temp -> next != top)
                {
                        temp = temp -> next;
                }
                temp -> next = NULL;
                printf("\n\n\t Popped element is %d ", top -> data);
                free(top);
                top = temp;
        }
}
void display()
{
        node *temp;
        if(top == NULL)
        {
                printf("\n\n\t\t Stack is empty ");
        }
        else
        {
                temp = start;

                 printf("\n\n\n\t\t Elements in the stack: \n");
                 printf("%5d  ", temp -> data);
                 while(temp != top)
                 {
                        temp = temp -> next;
                        printf("%5d   ", temp -> data);
                 }
         }
 }

 char menu()
 {
         char ch;
         clrscr();
         printf("\n \tStack operations using pointers.. ");
         printf("\n ----------*********----------\n");
         printf("\n 1. Push ");
         printf("\n 2. Pop ");
         printf("\n 3. Display");
         printf("\n 4. Quit ");
         printf("\n Enter your choice: ");
         ch = getche();
         return ch;
 }
```

```
void main()
{
        char ch;
        node *newnode;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                newnode = getnode();
                                push(newnode);
                                break;
                        case '2' :
                                pop();
                                break;
                        case '3' :
                                display();
                                break;
                        case '4':
                                return;
                }
                getch();
        } while( ch != '4' );
}
```

Algebraic Expressions:

A sanctioned collection of operators & operands is an algebraic expression. The measure on which a exact activity is performed is referred to as the operand. The operand can be a variable like a, b, or c or a constant like 2,3,4 and so on. The operator symbol represents a numerical or logical operation between the operands. +,
-, *, /, and similar operators are examples.

Three alternative notations can be used to represent an algebraic expression. There are three types of notations: infix, postfix, and prefix.

Infix: An arithmetical demonstration in which the arithmetical operator is fixed (placed) between the two operands.

For instance, (A + B) * (C - D)

Prefix: A kind of arithmetical notation in which the arithmetical operator is fixed (placed) before (pre) its two operands. Polish notation is named after the Polish mathematician Jan Lukasiewicz, who invented it in 1920.* + A B - C D as an example.

Postfix: The form of an arithmetic expression in which we fix (position) the arithmetic operator after (post) its two operands is known as a postfix. The postfix notation is also known as suffix notation and reverse polish notation.

For instance, A B + C D - *

The three most important characteristics of postfix reflection are:

1. The operands are ordered in the same way as in the corresponding infix expression.
2. Parenthesis are not required to unambiguously designate the expression.
3. The precedence of the operators is no longer significant when evaluating the postfix expression.

There are 5 binary transactions to consider: +, -, *, /, and $ or (exponentiation). The following multiple operations, in order of precedency (highest to lowest):

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Converting Expressions using Stack:

Let's change the expressions from one format to another. These can be accomplished as follows:

1. From In-fix to Post-fix
2. From Pre-fix to In-fix
3. From Post-fix to In-fix
4. From Post-fix to Pre-fix
5. From Pre-fix to In-fix
6. From Pre-fix to Post-fix

Converting from In-fix to Post-fix:

The following is the procedure for converting an In-fix expression to a Post-fix expression:

1. Begin From left to right, scan the infix phrase.
2. **a)** Place the scanned symbol on the stack if it is left parenthesis.

b)Place the scanned symbol immediately in the Post-fix expression (output) if it is an operand.

c)If the scanned symbol is a right parenthesis, continue popping items from the stack and inserting them into the postfix expression until we find the matching left parenthesis.

d)If the scanned symbol is an operator, continue removing all the operators from the stack and inserting them into the postfix expression if and only if the precedence of the operator at the top of the stack is greater than (or greater than or equal to) the precedence of the scanned operator and push the scanned operator onto the stack; otherwise, pop the scanned operator onto the stack.

Convert the following infix expression A + (B * C – (D / E ↑ F) * G) * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
(	A	+ (	
B	A B	+ (	
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(	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 ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Converting from In-fix to Pre-fix:

For transforming an expression from infix to prefix, the precedence rules are the same. The sole difference between postfix and prefix conversion is that the expression is traversed from right to left and the operator is placed before the operands rather than after them. A complex expression's prefix form is not the inverse of its postfix form.

Convert the infix expression A ↑ B * C – D + E / F / (G + H) into prefix expression.

SYMBOL	PREFIX STRING	STACK	REMARKS
)		)	
H	H	)	
+	H	) +	
G	G H	) +	
(	+ G H		
/	+ G H	/	
F	F + G H	/	
/	F + G H	/ /	
E	E F + G H	/ /	
+	/ / E F + G H	+	
D	D / / E F + G H	+	
-	D / / E F + G H	+ -	
C	C D / / E F + G H	+ -	
*	C D / / E F + G H	+ - *	
B	B C D / / E F + G H	+ - *	
↑	B C D / / E F + G H	+ - * ↑	
A	A B C D / / E F + G H	+ - * ↑	
End of string	+ - * ↑ A B C D / / E F + G H	The input is now empty. Pop the output symbols from the stack until it is empty.	

Converting from postfix to infix:

The following is the procedure for converting a postfix expression to an infix expression:

1. From left to right, scan the postfix phrase.
2. Push the scanned symbol into the stack if it is an operand.
3. If the scanned symbol is an operator, remove two symbols from the stack and stringify it by inserting the operator between the operands and pushing it back onto the stack.
4. Repeat steps 2 and 3 until the phrase is completed.

Conversion from postfix to prefix:

The following is the procedure for converting a postfix expression to a prefix expression:

1. From left to right, scan the postfix phrase.
2. Push the scanned symbol into the stack if it is an operand.
3. If the scanned symbol is an operator, remove two symbols from the stack and stringify it by putting the operator in front of the operands and pushing it back onto the stack.
4. Repeat steps 2 and 3 until the phrase is completed.

Convert the following postfix expression A B C * D E F ^ / G * - H * + into its equivalent prefix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A *BC	Pop two operands and place the operator in front the operands and push the string.
D	A *BC D	Push D
E	A *BC D E	Push E
F	A *BC D E F	Push F
^	A *BC D ^EF	Pop two operands and place the operator in front the operands and push the string.
/	A *BC /D^EF	Pop two operands and place the operator in front the operands and push the string.
G	A *BC /D^EF G	Push G
*	A *BC */D^EFG	Pop two operands and place the operator in front the operands and push the string.
-	A - *BC*/D^EFG	Pop two operands and place the operator in front the operands and push the string.
H	A - *BC*/D^EFG H	Push H
*	A *- *BC*/D^EFGH	Pop two operands and place the operator in front the operands and push the string.
+	+A*- *BC*/D^EFGH	
End of string	The input is now empty. The string formed is prefix.	

Conversion from prefix to infix:

The following is the procedure for converting a prefix expression to an infix expression:

1. Scan the prefix expression from left to right (in reverse order).
2. Push the scanned symbol into the stack if it is an operand.
3. If the scanned symbol is an operator, remove two symbols from the stack and stringify it by inserting the operator between the operands and pushing it back onto the stack.
4. Repeat steps 2 and 3 until the phrase is completed.

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent infix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G (E^F)	Pop two operands and place the operator in between the operands and push the string.
D	H G (E^F) D	Push D
/	H G (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
*	H ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
C	H ((D/(E^F))*G) C	Push C
B	H ((D/(E^F))*G) C B	Push B
*	H ((D/(E^F))*G) (B*C)	Pop two operands and place the operator in front the operands and push the string.
-	H ((B*C)-((D/(E^F))*G))	Pop two operands and place the operator in front the operands and push the string.
*	(((B*C)-((D/(E^F))*G))*H)	Pop two operands and place the operator in front the operands and push the string.
A	(((B*C)-((D/(E^F))*G))*H) A	Push A
+	(A+(((B*C)-((D/(E^F))*G))*H))	Pop two operands and place the operator in front the operands and push the string.
End of string	The input is now empty. The string formed is infix.	

Converting from Pre-fix to Post-fix:

The following is the procedure for converting a prefix expression to a postfix expression:

1. Scan the prefix expression from left to right (in reverse order).
2. Push the scanned symbol into the stack if it is an operand.
3. If the scanned symbol is an operator, remove two symbols from the stack and stringify it by inserting the operator after the operands and pushing it back onto the stack.
4. Repeat steps 2 and 3 until the phrase is completed.

Convert the following prefix expression + A * - * B C * / D ^ E F G H into its equivalent postfix expression.

Symbol	Stack	Remarks
H	H	Push H
G	H G	Push G
F	H G F	Push F
E	H G F E	Push E
^	H G EF^	Pop two operands and place the operator after the operands and push the string.
D	H G EF^ D	Push D
/	H G DEF^/	Pop two operands and place the operator after the operands and push the string.
*	H DEF^/G*	Pop two operands and place the operator after the operands and push the string.
C	H DEF^/G* C	Push C
B	H DEF^/G* C B	Push B
*	H DEF^/G* BC*	Pop two operands and place the operator after the operands and push the string.
-	H BC*DEF^/G*-	Pop two operands and place the operator after the operands and push the string.
*	BC*DEF^/G*-H*	Pop two operands and place the operator after the operands and push the string.
A	BC*DEF^/G*-H* A	Push A
+	ABC*DEF^/G*-H*+	Pop two operands and place the operator after the operands and push the string.
End of string	The input is now empty. The string formed is postfix.	

Evaluation of Post-fix expression:

A stack is used to easily evaluate the postfix expression. When a number is seen, it is added to the stack; when an operator is seen, it is applied to the two numbers that are popped from the stack, and the result is added to the stack. There is no need to know any precedence rules when an expression is given in postfix notation; this is our clear advantage.

Example-1: Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example -2: Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Applications of stacks:

1.Compilers employ stack to ensure that parentheses, brackets, and braces are balanced.

2.A postfix expression is evaluated using stack.

3.Stack is used to transform an infix expression into a postfix/prefix expression.

4.All intermediate arguments and return values are saved on the processor's stack during recursion.

5.During a function call, the return address and arguments are pushed into a stack and popped off upon return.

Queue:

A queue is a type of list in which things are added at one end called the back and deleted at the other end called the front. A queue is also known as a "FIFO" or "First-in-First-Out" list.

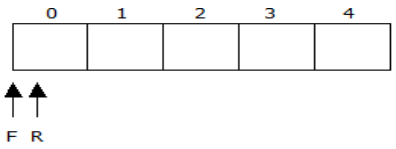
The operations of a queue are analogous to those of a stack, with the exception that insertions are made at the end of the list rather than the beginning. We will employ the following queue operations:

enqueue: inserts an element at the bottom of the queue.

dequeue: removes an element from the queue's beginning.

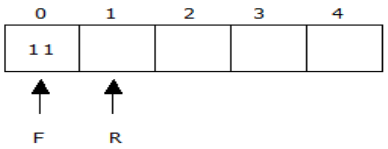
Representation of Queue:

Consider a queue with a maximal capacity of 5 elements. The queue is initially blank.



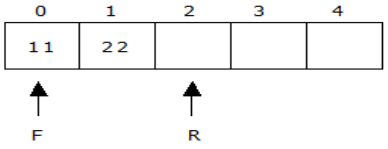
Queue Empty
FRONT = REAR = 0

Now, insert 11 to the queue. Then queue status will be:



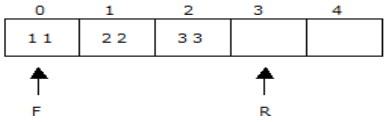
REAR = REAR + 1 = 1
FRONT = 0

Next, insert 22 to the queue. Then the queue status is:



REAR = REAR + 1 = 2
FRONT = 0

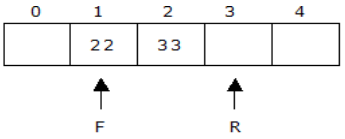
Again insert another element 33 to the queue. The status of the queue is:



REAR = REAR + 1 = 3
FRONT = 0

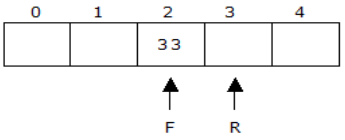
Now remove a piece. The element that was remove is the one at the top of the queue.

So the queues current state is:



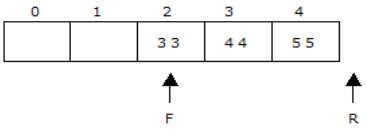
REAR = 3
FRONT = FRONT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



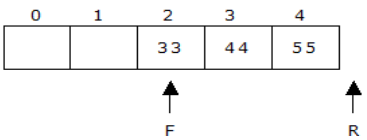
REAR = 3
FRONT = FRONT + 1 = 2

Now, insert new elements 44 and 55 into the queue. The queue status is:



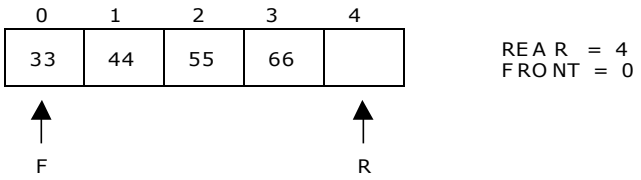
REAR = 5
FRONT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



REAR = 5
FRONT = 2

Even if there are 2 open emplacement in the linear queue, inserting element 66 is no longer possible. To solve this problem, the queue items should be pushed to the front of the line, leaving a empty position at the back. The FRONT and REAR must then be appropriately well-adjusted. The component 66 might be placed at the back end. The queue status after this activity is as follows:



This challenge can be avoided by treating queue position 0 as a position that comes after position 4, i.e., by treating the queue as a circular queue.

Queue operations using array:

A one-dimensional array Q(1:n) and two variables front and back are required to form a queue. We will use the following conventions for these two variables: front is always one less than the real front of the queue, and rear always points to the last piece in the queue. Thus, front = rear if and only if the queue has no elements. Thus, the initial condition is front = rear = 0.

The following are the numerous queue operations for creating, deleting, and displaying queue elements:

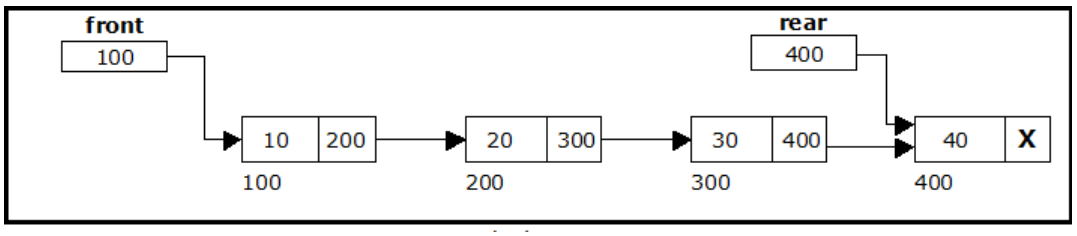
1.insertQ(): adds an element to the end of the queue. Q.

2.deleteQ(): This function deletes the first element of Q.

3.displayQ(): displays the queue's elements.

Linked List Execution of Queue:

A queue can be represented as a linked list. Data is erased from the front end and inserted at the back end of a queue. Similar procedures can be performed on the two ends of a list. For our linked queue implementation, we employ two pointers, one in front and one in back.



Queue Applications:

1.It is used to schedule jobs for the CPU to process.

2.When many users deliver print jobs to the same printer, each print job is stored in the printing queue. The printer then prints those jobs in the order of first in, first out (FIFO).

3.Breadth first search finds an element in a graph using a queue data structure.

Circular Queue:

By considering the array Q[MAX] as circular, a more efficient queue representation is created. The queue might hold an unlimited number of things.

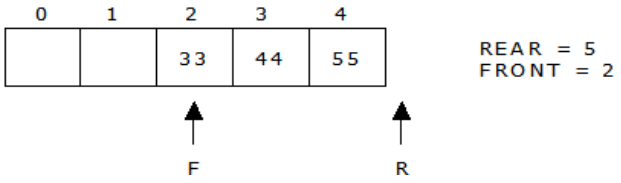
This queue implementation is known as a circular queue because it uses its storage array as if it were a circle rather than a linear list.

There are two issues with linear queueing. They are as follows:

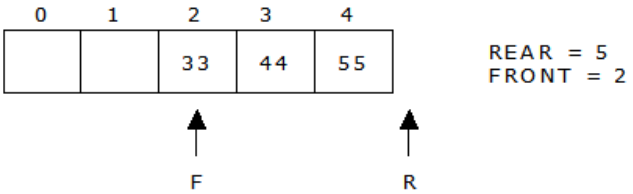
Time consuming: the linear time required to get the elements to the front of the queue.

Signaling queue full: even if the queue has an empty position.

For example, let us consider a linear queue status as follows:



Next, add another element to the queue, say 66. We are unable to add 66 to the queue because the back has exceeded the queue's maximum capacity (i.e., 5). There will be a signal indicating that the queue is filled. The current queue status is as follows:

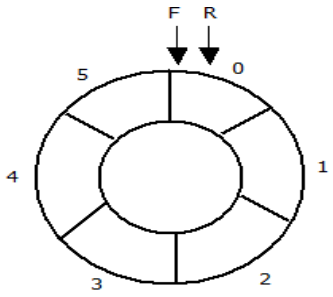


This problem can be solved by treating queue position zero as a position that comes after position four, and then treating the queue as a circular queue.

If we reach the end of the circular queue for inserting elements, we can insert additional components if the slots at the beginning of the circular queue are empty.

Circular Queue Representation:

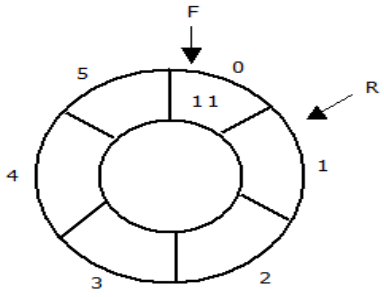
Consider a circular-queue with a maximal (MAX) capacity of 6 elements. The queue is initially blank.



Circular Queue

```
Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0
```

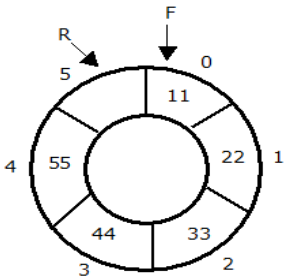
Now, insert 11 to the circular queue. Then circular queue status will be:



Circular Queue

```
FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1
```

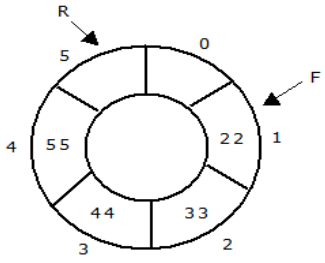
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



Circular Queue

```
FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5
```

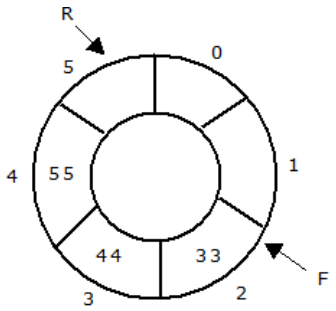
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4
```

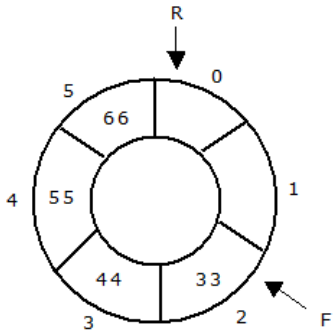
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3
```

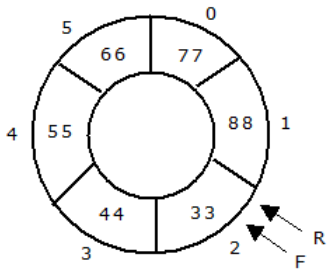
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

```
FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4
```

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



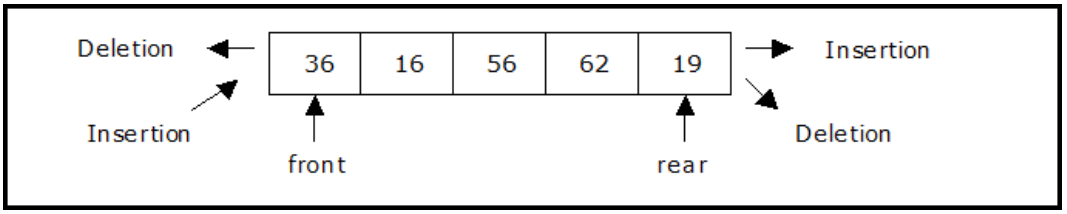
Circular Queue

```
FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6
```

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is full.

Deque:

In the preceding section, we saw a queue into which we enter items at one end and withdraw them at the other. In this section, we look at a queue extender that allows us to insert and remove items from both ends of the queue. This data structure is known as a deque. The term deque is an abbreviation for double-ended queue.



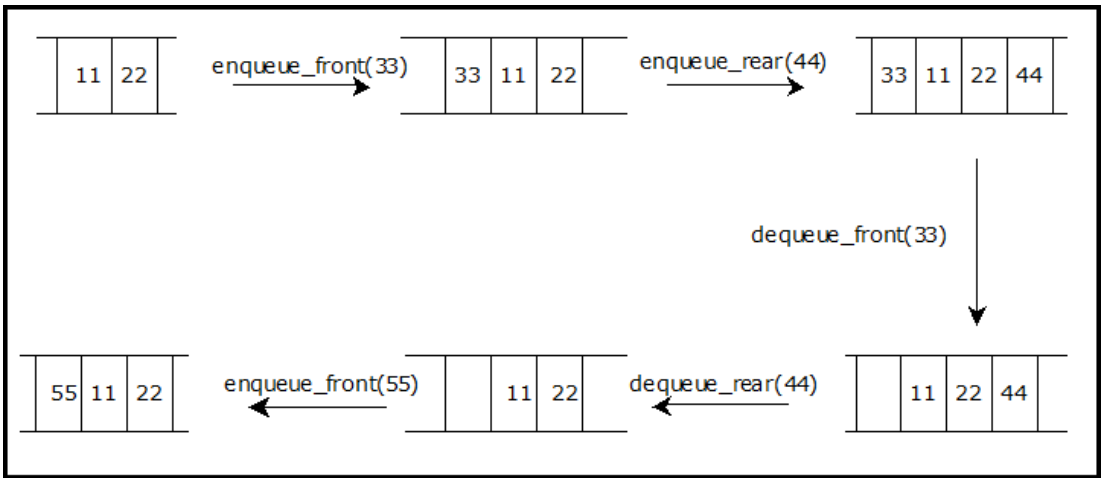
A deque has four operations.

enqueue_front: add an element to the front.

dequeue_front: remove an element from the front of the queue.

enqueue_rear: insert element in the back.

dequeue_rear: remove the element at the back of the queue.



Deque comes in two varieties. They are as follows: input restricted deque (IRD) and output restricted deque (ORD).

An Input restricted deque is a deque that enables insertions at one end but deletions at both ends.

A deque that allows deletions at one end but allows insertions at both ends is known as an output restricted deque.

Priority Queue:

A priority queue is a grouping of components in which each component has a priority and the ordering in which components are discarded and refined is determined by the following rules:

1.A higher priority component is handled before any lower priority element.

2.Duel components with the comparable priority are processed in the order they were introduced to the queue.

A priority queue example is a time sharing system, in which programs with higher priority are executed initial and programs with the comparable priority form a normal queue.

Heap is an high-octane version of the Priority Queue that may also be used for sorting purposes, which is known as heap sort.