UNIT – III:

Stacks: Introduction to Stacks, Stack as an Abstract Data Type, Representation of Stacks through Arrays, Representation of Stacks through Linked Lists, Applications of Stacks, Stacks and Recursion

Queues: Introduction, Queue as an Abstract data Type, Representation of Queues, Circular Queues, Double Ended Queues- Deques, Priority Queues, Application of Queues

---

Stack:

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

- *Push*: is the term used to insert an element into a stack.
- *Pop*: is the term used to delete an element from a stack.

"Push" is the term used to insert an element into a stack. "Pop" is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

- Main update methods:
  - Push (e):Pushing (storing) an element on the stack.
  - Pop ( ):Removing (accessing) an element from the stack.
- Additional useful methods
  - Peek () Same as pop, but does not remove the element
  - Empty() Boolean, true when the stack is empty
  - Size () Returns the size of the stack

Stack ADT:

```
public class Stack
 {
                public Stack  { }
                public Boolean empty() { }
                public void push (String str) { }
                public String pop() { }
                public String peek ( ){ }
}
```

2.3 Stack Implementation:

peek(): prints the top element but does not remove.

Algorithm of peek() function –

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

isfull(): checks whether the stack is full or not**.**

- Algorithm of isfull() function –

```
begin procedure isfull
    if top equals to MAXSIZE
        return true
    else
        return false
    endif
end procedure
```

isempty(): Boolean, true when the stack is empty

- Algorithm of isempty() function –

```
begin procedure isempty
    if top less than 1
        return true
    else
        return false
    endif
end procedure
```

Push Operation

- Push operation involves a series of steps –
  - **Step 1** – Checks if the stack is full.
  - **Step 2** – If the stack is full, produces an error and exit.
  - **Step 3** – If the stack is not full, increments **top** to point next empty space.
  - **Step 4** – Adds data element to the stack location, where top is pointing.
  - **Step 5** – Returns success.
  - –

Algorithm for PUSH Operation

- begin procedure push: stack, data
  - if stack is full
    - return null
  - endif
    - top ← top + 1
    - stack[top] ← data
- end procedure

Pop Operation

- A Pop operation may involve the following steps –
  - Step 1 – Checks if the stack is empty.
  - Step 2 – If the stack is empty, produces an error and exit.

- Step 3 – If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 – Decreases the value of top by 1.
- Step 5 – Returns success.

- **Algorithm** for Pop Operation
- begin procedure pop: stack
  - if stack is empty
    - return null
  - endif
    - data ← stack[top]
    - top ← top - 1
  - return data
- end procedure

Representation of Stack:

A stack can be represented by means of Array, Structure, Pointer, and Linked List.

Array representation:

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 the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition.

When a element is added to a stack, the operation is performed by push(). Figure5.1 shows the creation of a stack and addition of elements using push().

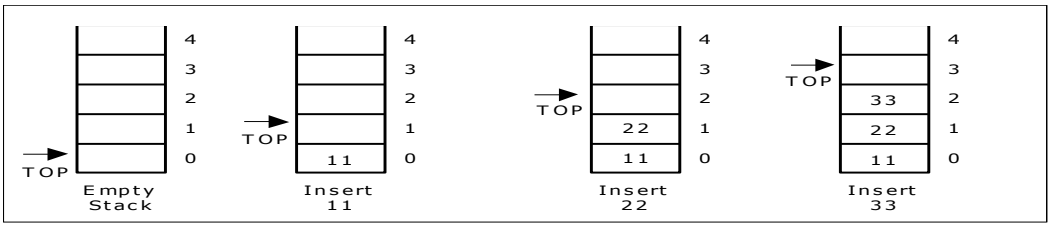


Figure 5.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure 5.2 shows a stack initially with three elements and shows the deletion of elements using pop().

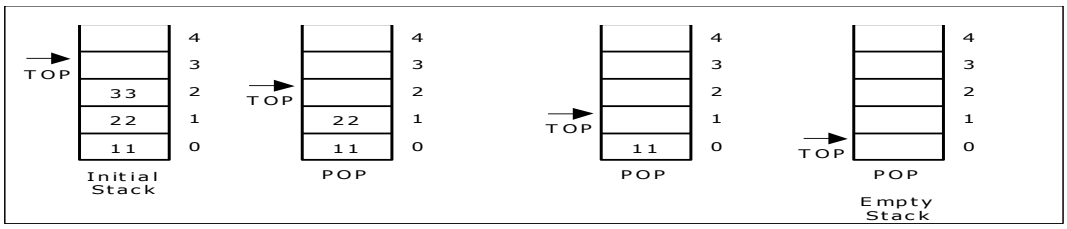


Figure 5.2. Pop operations on stack

Linked List Representation of Stack:

We can represent a 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. The linked stack looks as shown in figure 6.9.1:

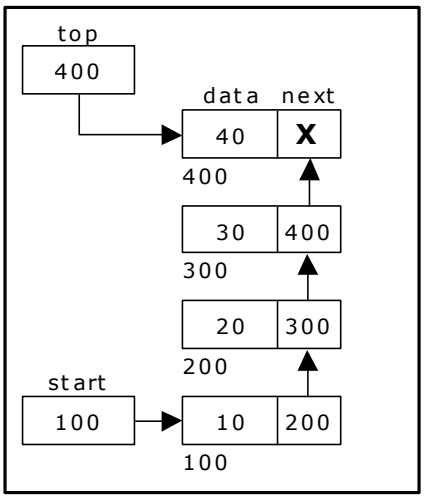


Figure 6.9.1. Linked stack representation

Source code for stack operations, using array:

```
# 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);
}
```

Source code for stack operations, using linked list:

```
	# include <stdio.h>
	# include <conio.h>
	# include <stdlib.h>
	struct stack
	{
		int data;
		struct stack *next;
	};
	void push();
	void pop();
	void display();
	typedef struct stack node;
	node *start=NULL;
	node *top = NULL;
	node* getnode()
	{
		struct stack *temp;
		temp=(node *) malloc( sizeof(node)) ;
		printf("\n Enter data ");
		scanf("%d", &temp -> data);
		temp -> next = NULL;
		return temp;
	}
	void push(node *newnode)
	{
		node *temp;
		if( newnode == NULL )
		{
			printf("\n Stack Overflow..");
			return;
```

```
        }
        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:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable

like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

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

Example: (A + B) * (C - D)

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: * + A B – C D

Postfix: It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as *suffix notation* and is also referred to *reverse polish notation*.

Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix expression.
2. The parentheses are not needed to designate the expression un-ambiguously.
3. While evaluating the postfix expression the priority of the operators is no longer relevant.

We consider five binary operations: +, -, *, / and $ or ↑ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

Operator	Precedence	Value
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

Applications of Stacks:

Reversing a data
Infix to postfix
Post expression evaluation
Infix to prefix
Prefix expression evaluation
Stacks and Recursion

Reversing a data

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

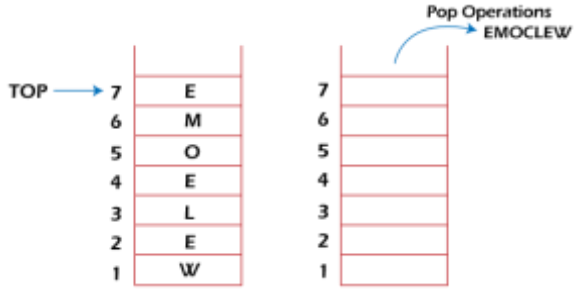
There are different reversing applications:

- Reversing a string
- Converting Decimal to Binary

Reverse a String

A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the last in first out property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is

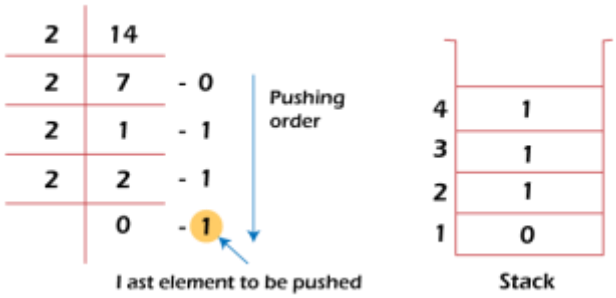
on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



Converting Decimal to Binary:

Although decimal numbers are used in most business applications, some scientific and technical applications require numbers in either binary, octal, or hexadecimal. A stack can be used to convert a number from decimal to binary/octal/hexadecimal form. For converting any decimal number to a binary number, we repeatedly divide the decimal number by two and push the remainder of each division onto the Stack until the number is reduced to 0. Then we pop the whole Stack and the result obtained is the binary equivalent of the given decimal number.

Example: Converting 14 number Decimal to Binary:



In the above example, on dividing 14 by 2, we get seven as a quotient and one as the reminder, which is pushed on the Stack. On again dividing seven by 2, we get three as quotient and 1 as the reminder, which is again pushed onto the Stack. This process continues until the given number is not reduced to 0. When we pop off the Stack completely, we get the equivalent binary number 1110.

Infix to postfix expression conversion:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.
   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).
   c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
   d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or equal)* to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example 1:
Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

Symbol	Postfix string	Stack	
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	+ *	
End of string	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Program to convert an infix to postfix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
char postfix[50];
char infix[50];
char opstack[50];                /*  operator stack  */
int i, j, top = 0;

int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                 /*  priority value of  op  */
        int pv2;                 /*  priority value of op_at_stack  */
        char operators[] = {'+', '-', '*', '/', '%', '^', '(' };
        int priority_value[] = {0,0,1,1,2,3,4};
        if( op_at_stack == '(' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if(op_at_stack == operators[k])
                        pv2 = priority_value[k];
```

```
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}

void push(char op)        /* op - operator  */
{
        if(top == 0)
        {
                opstack[top] = op;
                top++;
        }
        else
        {
                if(op != '(' )
                {
                        /* before pushing the operator 'op' into the stack check priority of op  with top
                of opstack if less then pop the operator from stack then push into postfix string else
                push op onto stack itself  */

                        while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                        {
                                        postfix[j] = opstack[--top];
                                        j++;
                        }
                }
                opstack[top] = op;      /* pushing onto stack  */
                top++;
        }
}

pop()
{
        while(opstack[--top] != '(' )               /*  pop until '(' comes   */
        {
                postfix[j] = opstack[top];
                j++;
        }
}

void main()
{
        char ch;
        clrscr();
        printf("\n Enter Infix Expression  : ");
        gets(infix);
        while( (ch=infix[i++]) != 0)
        {
                switch(ch)
                {
                        case ' ' : break;
                        case '(' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
```

```
                        case '%' :
                                push(ch);               /* check priority and push  */
                                break;
                        case ')' :
                                pop();
                                break;
                        default :
                                postfix[j] = ch;
                                j++;
                }
        }
        while(top >= 0)
        {
                postfix[j] =  opstack[--top];
                j++;
        }
        postfix[j] = '\0';
        printf("\n Infix  Expression    :  %s ", infix);
        printf("\n Postfix Expression  :  %s ", postfix);
        getch();
}
```

Post expression evaluation

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious 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

Infix to prefix expression Conversion

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that traverse the expression from right to left and the operator is placed before the operands rather than after them. The prefix form of a complex expression is not the mirror image of the postfix form.

Example 1:

Convert the infix expression A + B - C into prefix expression.

Symbol	Prefix String	Stack	Remarks
C	C		
-	C	-	
B	B C	-	
+	B C	- +	
A	A B C	- +	
End of string	- + A B C	The input is now empty. Pop the output symbols from the stack until it is empty.	

Program to convert an infix to prefix expression:

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

char prefix[50];
char infix[50];
char opstack[50];                /* operator stack  */
int j, top = 0;

void insert_beg(char ch)
{
        int k;
        if(j == 0)
                prefix[0] =  ch;
        else
        {
                for(k = j + 1; k > 0; k--)
                        prefix[k] = prefix[k - 1];
                prefix[0] = ch;
        }
        j++;
}
```

```
int lesspriority(char op, char op_at_stack)
{
        int k;
        int pv1;                    /* priority value of  op  */
        int pv2;                    /* priority value of op_at_stack  */
        char operators[] = {'+', '-', '*', '/', '%', '^', ')'};
        int priority_value[] = {0, 0, 1, 1, 2, 3, 4};
        if(op_at_stack == ')' )
                return 0;
        for(k = 0; k < 6; k ++)
        {
                if(op == operators[k])
                        pv1 = priority_value[k];
        }
        for(k = 0; k < 6; k ++)
        {
                if( op_at_stack == operators[k] )
                        pv2 = priority_value[k];
        }
        if(pv1 < pv2)
                return 1;
        else
                return 0;
}

void push(char op)              /* op – operator  */
{
        if(top == 0)
        {
                opstack[top] = op;
                top++;
        }
        else
        {
                if(op != ')')
                {
                        /* before pushing the operator 'op' into the stack check priority of op with top
                of operator stack if less pop the operator from stack then push into postfix string else
                push op onto stack itself  */

                        while(lesspriority(op, opstack[top-1]) == 1 && top > 0)
                        {
                                        insert_beg(opstack[--top]);
                        }
                }
                opstack[top] = op;                      /* pushing onto stack   */
                top++;
        }
}

void pop()
{
        while(opstack[--top] != ')')                    /*  pop until ')' comes;  */
                insert_beg(opstack[top]);
}

void main()
{
```

```
        char ch;
        int l, i = 0;
        clrscr();
        printf("\n Enter Infix Expression  : ");
        gets(infix);
        l = strlen(infix);
        while(l > 0)
        {
                ch = infix[--l];
                switch(ch)
                {
                        case ' ' : break;
                        case ')' :
                        case '+' :
                        case '-' :
                        case '*' :
                        case '/' :
                        case '^' :
                        case '%' :
                                push(ch);               /* check priority and push  */
                                break;
                        case '(' :
                                pop();
                                break;
                        default :
                                insert_beg(ch);
                }
        }
        while( top > 0 )
        {
                insert_beg( opstack[--top] );
                j++;
        }
        prefix[j] = '\0';
        printf("\n Infix Expression    :  %s ", infix);
        printf("\n Prefix Expression  :  %s ", prefix);
        getch();
}
```

Prefix expression evaluation

The order in which operations are performed is defined by their order in the expression. This makes use of parenthesis and precedence rule redundant. Thus, the evaluation of expressions becomes simpler.

Infix Expressions are first converted into Prefix/Postfix Expressions. Then they are evaluated. Evaluation of Prefix/Postfix Expressions can be performed using stack in one pass.

Algorithm

The Prefix Expression is scanned from Right-To-Left.

1. Initialize a stack.
2. Scan Prefix Expression from Right-To-Left.
   - If the scanned character is an operand, Push it to Stack.
   - If the scanned character is an operator, Pop operands from the stack depending upon the type of operator. Perform the operation and Push the result back to Stack.
3. Repeat step 2 until all characters of prefix expression are scanned.

//Program to Evaluate Prefix Expression

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

```
#include<conio.h>
#include<string.h>
int s[50];
int top=0;
void push(int ch);
int pop();
int main()
{
                        int a,b,c,i;
                        char prefix[50];
                        clrscr();

                        printf("\nEnter the prefix string in figures(1 digit nos);");
                        gets(prefix);
                        //for(i=0;i<strlen(prefix);i++)
                        for(i=strlen(prefix)-1;i>=0;i--)
                        {
                         if(prefix[i]=='+')
                         {
                                 c=pop()+pop();
                                 push(c);
                         }
                         else if(prefix[i]=='-')
                         {
                                 a=pop();
                                 b=pop();
                                 c=b-a;
                                 push(c);
                         }
                         else if(prefix[i]=='*')
                         {       a=pop();
                         b=pop();
                         c=b*a;
                         push(c);
                         }
                         else if(prefix[i]=='/')
                         {
                                 a=pop();
                                 b=pop();
                                 c=b/a;
                                 push(c);
                         }
                         else
                         {
                                 push(prefix[i]-48);
                                 //printf("\n INT=%d - CHAR=%d",prefix[i]-48,c);
                         }
                        }
                        printf("\nFinal ans = %d",pop());

                        getch();
                        return 0;
}
```

```
void push(int ch)
{
                        top++;
                        s[top]=ch;
}
int pop()
{
                        int ch;
                        ch=s[top];
                        top=top-1;
                        return(ch);
}
```

Stacks and recursion:

Recursion is a programming pattern that is useful in situations when a task can be naturally split into several tasks of the same kind, but simpler. A recursive function is one that calls itself.

For example, to find out pow(x,n) there are two ways to implement.

There are two ways to implement it.

1. **Iterative thinking: the for loop:**

```
function pow(x, n) {
  let result = 1;
    for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
alert( pow(2, 3) ); // 8
```

2. **Recursive thinking: simplify the task and call self:**

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
alert( pow(2, 3) ); // 8
```

Please note how the recursive variant is fundamentally different.

When pow(x, n) is called, the execution splits into two branches:

1. If n == 1, then everything is trivial. It is called the base of recursion, because it immediately produces the obvious result: pow(x, 1) equals x.
2. Otherwise, we can represent pow(x, n) as x * pow(x, n - 1). In maths, one would write $x^n = x * x^{n-1}$. This is called a recursive step: we transform the task into a simpler action (multiplication by x) and a simpler call of the same task (pow with lower n). Next steps simplify it further and further until n reaches 1.

We can also say that pow recursively calls itself till n == 1.

Now let's examine how recursive calls work. For that we'll look under the hood of functions.

The information about the process of execution of a running function is stored in its *execution context*.

The execution context is an internal data structure that contains details about the execution of a function: where the control flow is now, the current variables, the value of this (we don't use it here) and few other internal details.

One function call has exactly one execution context associated with it.
When a function makes a nested call, the following happens:

- The current function is paused.
- The execution context associated with it is remembered in a special data structure called *execution context stack*.
- The nested call executes.
- After it ends, the old execution context is retrieved from the stack, and the outer function is resumed from where it stopped.

Queues: Introduction, Queue as an Abstract data Type, Representation of Queues, Circular Queues, Double Ended Queues- Deques, Priority Queues, Application of Queues.

A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a "FIFO" or "First-in-first-out" list.

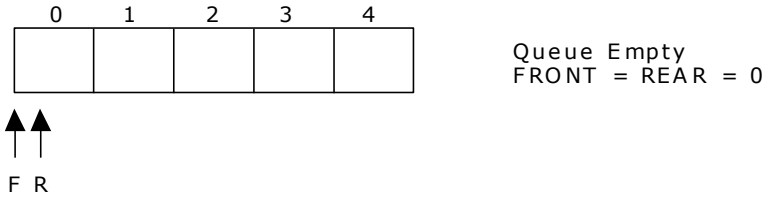
The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

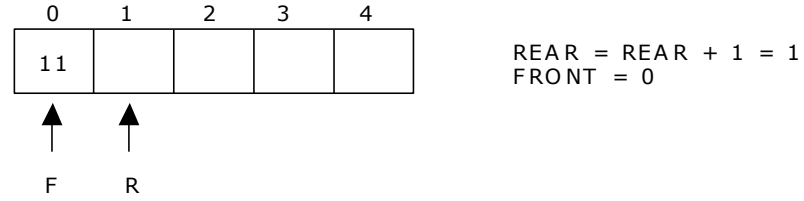
Representation of Queue:

Array representation:

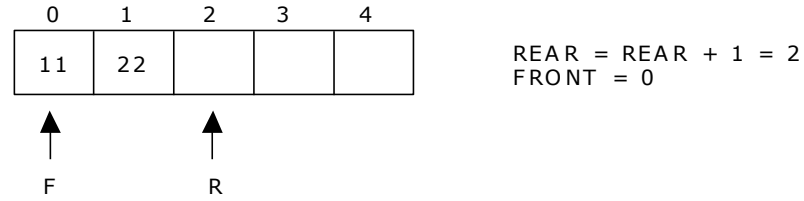
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



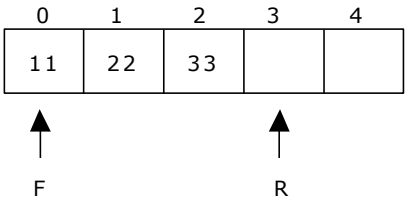
Now, insert 11 to the queue. Then queue status will be:



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

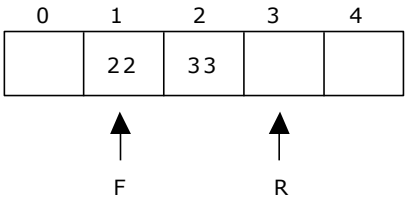


Again insert another element 33 to the queue. The status of the queue is:



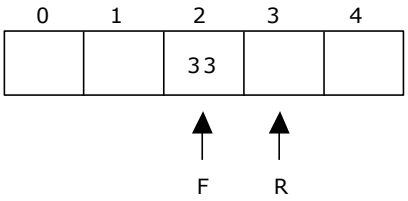
REAR = REAR + 1 = 3
FRONT = 0

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



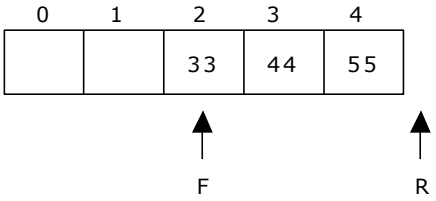
REAR = 3
FRONT = FRONT + 1 = 1

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:



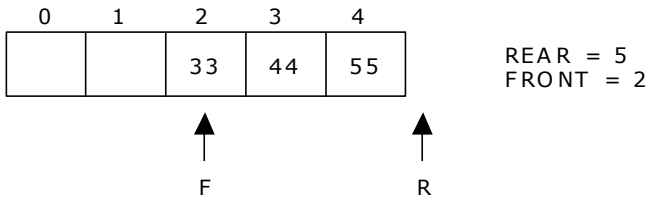
REAR = 3
FRONT = FRONT + 1 = 2

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

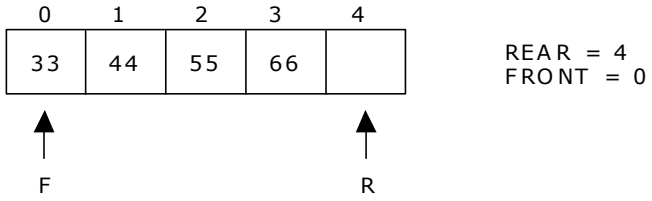


REAR = 5
FRONT = 2

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:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

QUEUE Abstract Data Type

```
public interface QueueInterface
{
        public Object dequeue()
        public boolean isEmpty();
        public void enqueue(Object element)
        public boolean isFull();
}
```

Implementation of Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.
    - Few more functions are required to make the above-mentioned queue operation efficient. These are −
- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.
    - In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

peek(): This function helps to see the data at the **front** of the queue.

- The algorithm of peek() function is as follows
- **Algorithm**

```
begin procedure peek
    return queue[front]
end procedure
```

isfull()

- As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full.Algorithm of isfull() function −
- **Algorithm**

begin procedure isfull
 if rear equals to MAXSIZE
 return true
else return false
endif
end procedureImplementation of isfull()

isempty()

- **Algorithm**

begin procedure isempty
 if front is less than MIN OR front is greater than rear
 return true
 else return false
 endif
end procedure

- If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.
- Here's the C programming code −

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

Step 1 − Check if the queue is full.

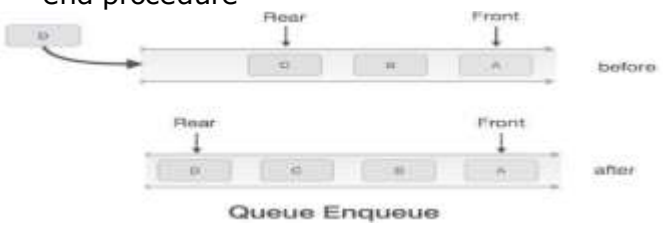
Step 2 − If the queue is full, produce overflow error and exit.

Step 3 − If the queue is not full, increment **rear** pointer to point the next empty space.

Step 4 − Add data element to the queue location, where the rear is pointing.

Step 5 − return success.

- procedure enqueue(data)
  - if queue is full
    - return overflow
  - endif
  - rear ← rear + 1
  - queue[rear] ← data
  - return true
- end procedure



Queue Enqueue

Source code Queue operations using array:

```
# include <stdio.h>
# include <conio.h>
# define MAX 6
int Q[MAX];
int front, rear;
```

```
void insertQ()
{
	int data;
	if(rear == MAX)
	{
		printf("\n Linear Queue is full");
		return;
	}
	else
	{
		printf("\n Enter data: ");
		scanf("%d", &data);
		Q[rear] = data;
		rear++;
		printf("\n Data Inserted in the Queue ");
	}
}
void deleteQ()
{
	if(rear == front)
	{
		printf("\n\n Queue is Empty..");
		return;
	}
	else
	{
		printf("\n Deleted element from Queue is %d", Q[front]);
		front++;
	}
}
void displayQ()
{
	int i;
	if(front == rear)
	{
		printf("\n\n\t Queue is Empty");
		return;
	}
	else
	{
		printf("\n Elements in Queue are: ");
		for(i = front; i < rear; i++)
		{
			printf("%d\t", Q[i]);
		}
	}
}
int menu()
{
	int ch;
	clrscr();
	printf("\n \tQueue operations using ARRAY..");
	printf("\n ----------**********----------\n");
```

```
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        scanf("%d", &ch);
        return ch;
}
void main()
{
        int ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case 1:
                                insertQ();
                                break;
                        case 2:
                                deleteQ();
                                break;
                        case 3:
                                displayQ();
                                break;
                        case 4:
                                return;
                }
                getch();
        } while(1);
}
```

Linked List Represenation of Queue:

We can represent a queue as a 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.

The linked queue looks as shown in figure 6.10.1:

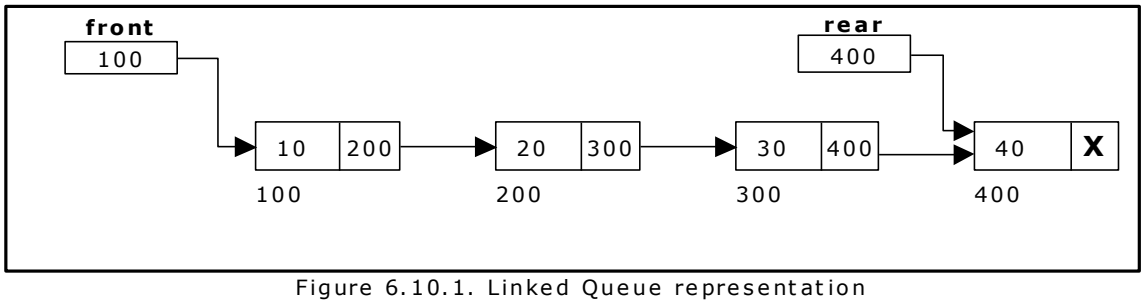


Figure 6.10.1. Linked Queue representation

Source code for queue operations using linked list:

```
# include <stdio.h>
# include <stdlib.h>
# include <conio.h>
struct queue
{
        int data;
        struct queue *next;
};

typedef struct queue node;
node *front = NULL;
node *rear =  NULL;

node* getnode()
{
        node *temp;
        temp = (node *) malloc(sizeof(node)) ;
        printf("\n Enter data ");
        scanf("%d", &temp -> data);
        temp -> next = NULL;
        return temp;
}
void insertQ()
{
        node *newnode;
        newnode = getnode();
        if(newnode == NULL)
        {
                printf("\n Queue Full");
                return;
        }
        if(front == NULL)
        {
                front = newnode;
                rear = newnode;
        }
        else
        {
                rear -> next = newnode;
                rear = newnode;
        }
        printf("\n\n\t Data Inserted into the Queue..");
}
void deleteQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t Empty Queue..");
                return;
        }
        temp = front;
```

```
        front = front -> next;
        printf("\n\n\t Deleted element from queue is %d ", temp -> data);
        free(temp);
}

void displayQ()
{
        node *temp;
        if(front == NULL)
        {
                printf("\n\n\t\t Empty Queue ");
        }
        else
        {
                temp = front;
                printf("\n\n\n\t\t Elements in the Queue are: ");
                while(temp != NULL )
                {
                        printf("%5d  ", temp -> data);
                        temp = temp -> next;
                }
        }
}
char menu()
{
        char ch;
        clrscr();
        printf("\n \t..Queue operations using pointers.. ");
        printf("\n\t ----------**********----------\n");
        printf("\n 1. Insert ");
        printf("\n 2. Delete ");
        printf("\n 3. Display");
        printf("\n 4. Quit ");
        printf("\n Enter your choice: ");
        ch = getche();
        return ch;
}
void main()
{
        char ch;
        do
        {
                ch = menu();
                switch(ch)
                {
                        case '1' :
                                insertQ();
                                break;
                        case '2' :
                                deleteQ();
                                break;
                        case '3' :
                                displayQ();
```

```
                    break;
                case '4':
                    return;
            }
            getch();
        } while(ch != '4');
    }
```

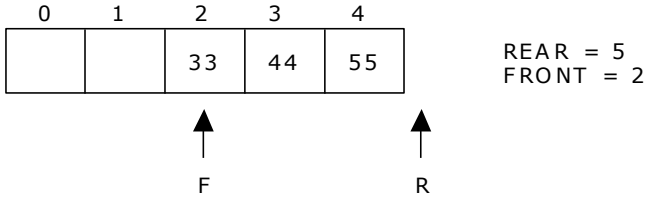
Circular Queue:

A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue, so long as items were also being taken off. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

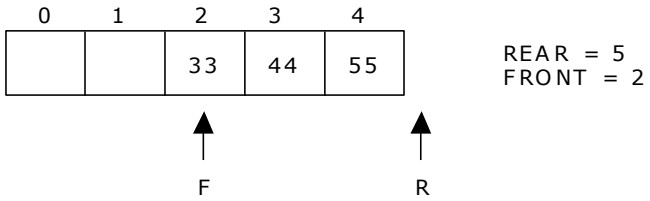
There are two problems associated with linear queue. They are:

- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- Signaling queue full: even if the queue is having vacant position.

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



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:

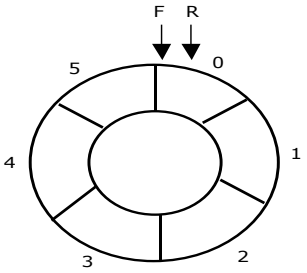


This difficulty can overcome if we treat queue position with index zero as a position that comes after position with index, four then we treat the queue as a **circular queue.**

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

Representation of Circular Queue:

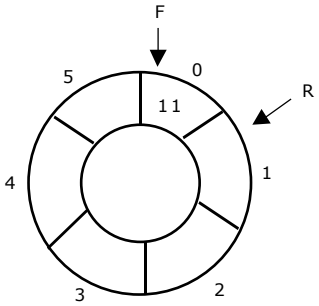
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Circular Queue

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

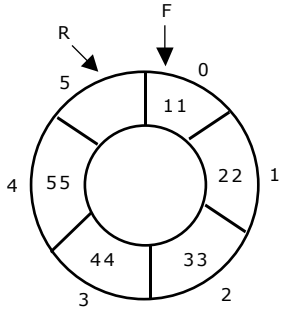
Now, insert 11 to the circular queue. Then circular queue status will be:



Circular Queue

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

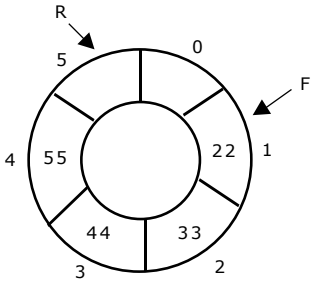
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 0, REAR = 5
REAR = REAR % 6 = 5
COUNT = 5

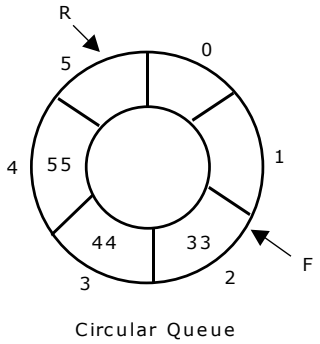
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

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

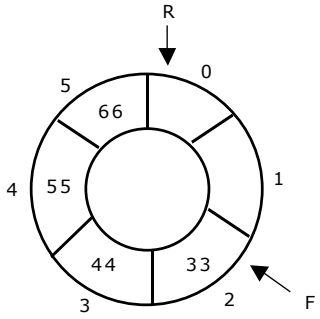
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

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

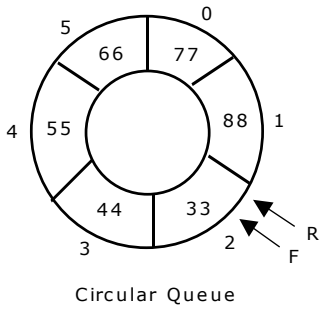
Again, insert another element 66 to the circular queue. The status of the circular queue is:



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*.

Implementation of Circular Queue

Below we have the implementation of a circular queue:

1. Initialize the queue, with size of the queue defined (maxSize), and head and tail pointers.
2. enqueue: Check if the number of elements is equal to maxSize - 1:
   - If **Yes**, then return **Queue is full**.
   - If **No**, then add the new data element to the location of tail pointer and increment the tailpointer.
3. dequeue: Check if the number of elements in the queue is zero:
   - If **Yes**, then return **Queue is empty**.

- If **No**, then increment the `head` pointer.

4. Finding the `size`:
   - If, **tail >= head**, `size = (tail - head) + 1`
   - But if, **head > tail**, then `size = maxSize - (head - tail) + 1`

Source code Circular Queue operations, using array:

```
# include <stdio.h>
# include <conio.h>
# define MAX 6
int CQ[MAX];
int front = 0;
int rear = 0;
int count = 0;
void insertCQ()
{
                                int data;
                                if(count == MAX)
                                {
                                 printf("\n Circular Queue is Full");
                                }
                                else
                                {
                                 printf("\n Enter data: ");
                                 scanf("%d", &data);
                                 CQ[rear] = data;
                                 rear = (rear + 1) % MAX;
                                 count ++;
                                 printf("\n Data Inserted in the Circular Queue ");
                                }
}
void deleteCQ()
{
if(count == 0)
{
printf("\n\nCircular Queue is Empty..");
}
else
{
printf("\n Deleted element from Circular Queue is %d ", CQ[front]);
front = (front + 1) % MAX;
count --;
}
}
void displayCQ()
{
                                int i, j;
                                if(count == 0)
                                {
                                 printf("\n\n\t Circular Queue is Empty ");
                                }
                                else
                                {
                                 printf("\n Elements in Circular Queue are: ");
```

```
                        j = count;
                        for(i = front; j != 0; j--)
                        {
                                printf("%d\t", CQ[i]);
                                i = (i + 1) % MAX;
                        }
                       }
}

int menu()
{
                       int ch;
                       clrscr();
                       printf("\n \t Circular Queue Operations using ARRAY..");
                       printf("\n ----------**********----------\n");
                       printf("\n 1. Insert ");
                       printf("\n 2. Delete ");
                       printf("\n 3. Display");
                       printf("\n 4. Quit ");
                       printf("\n Enter Your Choice: ");
                       scanf("%d", &ch);
                       return ch;
}

void main()
{
                       int ch;
                       do
                       {
                        ch = menu();
                        switch(ch)
                        {
                                case 1:
                                        insertCQ();
                                        break;
                                case 2:
                                        deleteCQ();
                                        break;
                                case 3:
                                        displayCQ();
                                        break;
                                case 4:
                                        return;
                                default:
                                        printf("\n Invalid Choice ");
                        }
                        getch();
                       } while(1);
}
```

Deque:

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.In previous post we had discussed introduction of deque. Now in this post we see how we implement deque Using circular array.

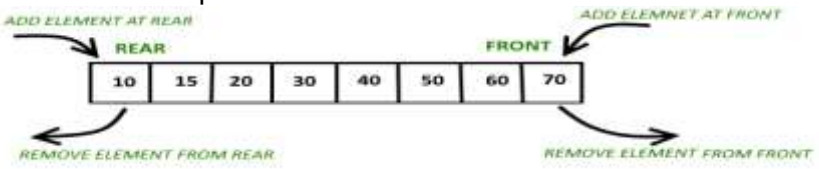
Operations on Deque:

Mainly the following four basic operations are performed on queue:

insetFront(): Adds an item at the front of Deque.
insertRear(): Adds an item at the rear of Deque.
deleteFront(): Deletes an item from front of Deque.
deleteRear(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

getFront(): Gets the front item from queue.
getRear(): Gets the last item from queue.
isEmpty(): Checks whether Deque is empty or not.
isFull(): Checks whether Deque is full or not.



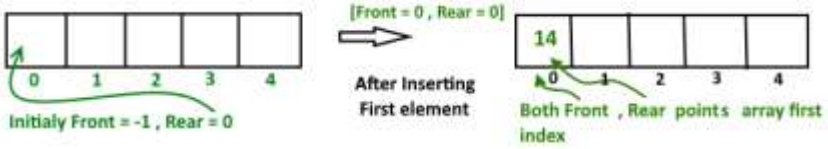
Circular array implementation deque

For implementing deque, we need to keep track of two indices, front and rear. We enqueue(push) an item at the rear or the front end of qedue and dequeue(pop) an item from both rear and front end.

Working

1. Create an empty array 'arr' of size 'n'
initialize **front = -1** , **rear = 0**

Inserting First element in deque, at either front or rear will lead to the same result.



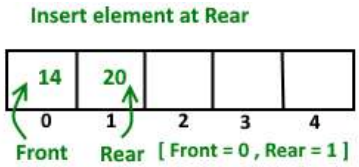
After insert Front Points = 0 and Rear points = 0

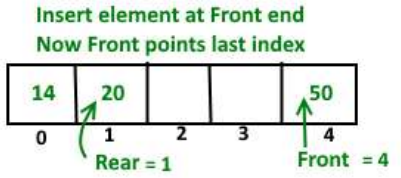
Insert Elements at Rear end

a). First we check deque if Full or Not
b). IF Rear == Size-1
then reinitialize Rear = 0 ;
Else increment Rear by '1'
and push current key into Arr[rear] = key
Front remain same.

Insert Elements at Front end

a). First we check deque if Full or Not
b). IF Front == 0 || initial position, move Front to points last index of array
front = size - 1
Else decremented front by '1' and push
current key into Arr[Front] = key
Rear remain same.





Delete Element From Rear end

a). first Check deque is Empty or Not
b). If deque has only one element
front = -1 ; rear =-1 ;
Else IF Rear points to the first index of array
it's means we have to move rear to points
last index [now first inserted element at
front end become rear end]
rear = size-1 ;
Else || decrease rear by '1'
rear = rear-1;

Delete Element From Front end

a). first Check deque is Empty or Not
b). If deque has only one element
front = -1 ; rear =-1 ;
Else IF front points to the last index of the array
it's means we have no more elements in array so
we move front to points first index of array
front = 0 ;
Else || increment Front by '1'
front = front+1;

Priority Queue:

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

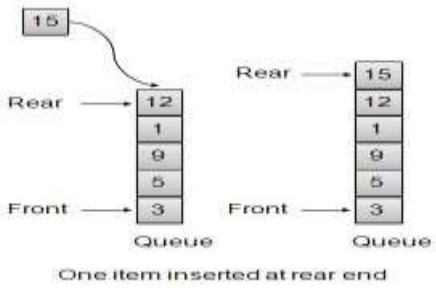
Basic Operations

- **insert / enqueue** – add an item to the rear of the queue.

- **remove / dequeue** – remove an item from the front of the queue.

There is few more operations supported by queue which are following.

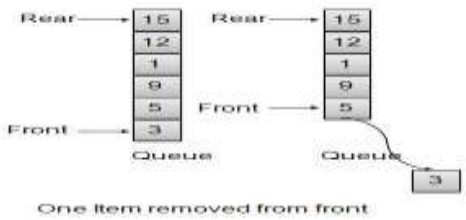
- **Peek** – get the element at front of the queue.
- **isFull** – check if queue is full.
- **isEmpty** – check if queue is empty.
- Insert / Enqueue Operation
- Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.




```
void insert(int data)
{
    int i = 0;
    if(!isFull())
    {
            // if queue is empty, insert the data
        if(itemCount == 0)
        {
            intArray[itemCount++] = data;
            }
        else
        {
            // start from the right end of the queue
            for(i = itemCount - 1; i >= 0; i-- )
            {
                    // if data is larger, shift existing item to right end
                    if(data > intArray[i]){
                            intArray[i+1] = intArray[i];
            }
        else
        {
            break;
        }
}
    // insert the data
    intArray[i+1] = data;
    itemCount++;
}
}
}
```

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.




```
int removeData()
{
  return intArray[--itemCount];
}
```

Applications of Queue

- **1)** When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
  2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

1. Computer controlled **Traffic Signal System** uses circular queue.
2. CPU scheduling and Memory management.
3. **Applications of Priority Queue:**

   1) CPU Scheduling

   2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc

   3) All queue applications where priority is involved.