UNIT - 2

Stack and Queue

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are:

- Stack.
- Queue.

Linear lists and arrays allow one to insert and delete elements at any place in the list i.e., at the beginning, at the end or in the middle.

4.1. 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.

4.1.1. Representation of Stack:

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 an element is added to a stack, the operation is performed by push(). Figure 4.1 shows the creation of a stack and addition of elements using push().
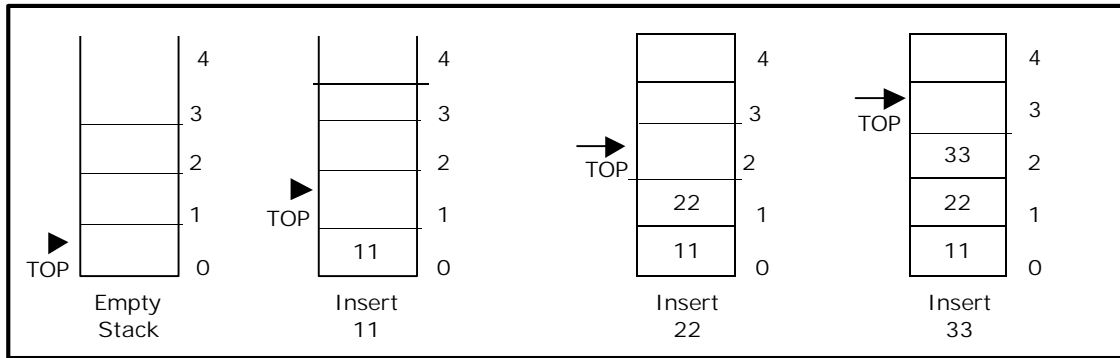
Figure 4.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure 4.2 shows a stack initially with three elements and shows the deletion of elements using pop().
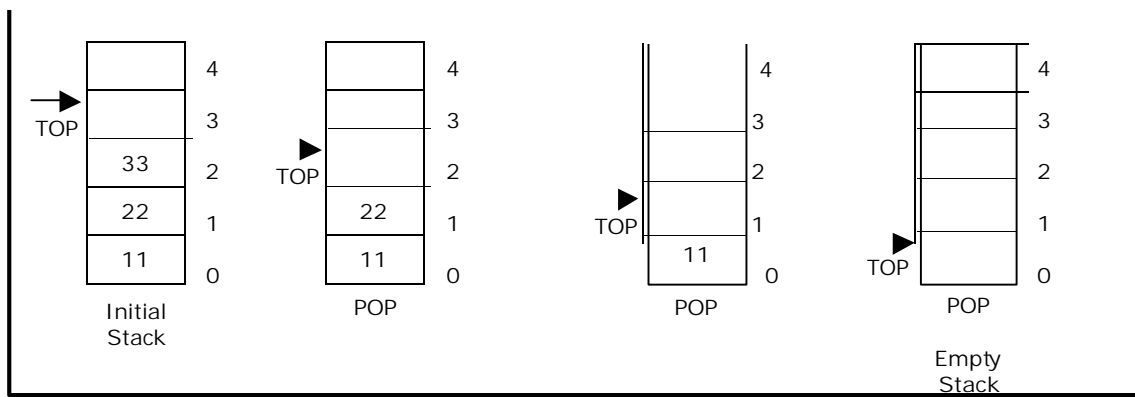

Figure 4.2. Pop operations on stack

4.1.2.   Source code for stack operations, using array:

```c
# 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..");
```

```c
                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);
}
```

4.1.3. Linked List Implementation 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 4.3.
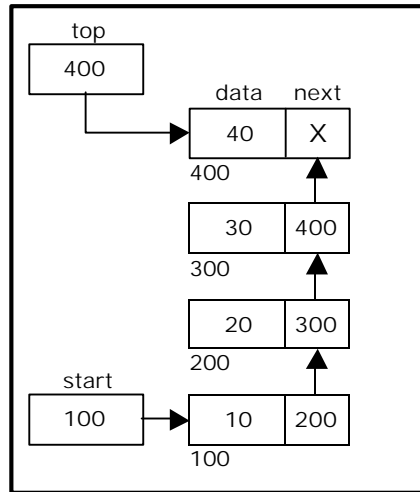

Figure 4.3. Linked stack representation

4.1.4. 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()
{
        node *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;
        }
```

```c
            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' );
}
```

4.2.   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).

<center>Example: * + A B – C D</center>

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.

<center>Example: A B + C D - *</center>

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

4.3. Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

4.3.1. Conversion from infix to postfix:

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 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) $\uparrow$ (E + F) infix expression to postfix form:

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

Example 2:

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

SYMBOL	POSTFIX STRING	STACK	REMARKS
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.	

Example 3:

Convert the following infix expression A + B * C − D / E * H into its equivalent postfix expression.

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

Example 4:

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.	

4.3.2.   Program to convert an infix to postfix expression:

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 != '(' )
                {
                        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++;
```

/* 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 */

```
        }
        postfix[j] = '\0';
        printf("\n Infix  Expression    :  %s ", infix);
        printf("\n Postfix Expression  :  %s ", postfix);
        getch();
}
```

4.3.3.   Conversion from infix to prefix:

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.

Example 2:

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

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

Example 3:

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.	

4.3.4.　　　　Program to convert an infix to prefix expression:

```
# 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++;
}
```

```c
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();
}
```

4.3.5.   Conversion from postfix to infix:

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

1.   Scan the postfix expression from left to right.

2.   If the scanned symbol is an operand, then push it onto the stack.

3.   If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.

4.   Repeat steps 2 and 3 till the end of the expression.

Example:

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

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

4.3.6.   Program to convert postfix to infix expression:

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
# define MAX 100

void pop (char*);
void push(char*);

char stack[MAX] [MAX];
int top = -1;
```

```c
void main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2],temp[2];
        int i=0;
        clrscr( ) ;
        printf("\Enter the postfix expression; ");
        gets(s);
        while (s[i]!='\0')
        {
                if(s[i] == ' ' )                    /*skip whitespace, if any*/
                        i++;
                if (s[i] == '^' || s[i] == '*'|| s[i] == '-' || s[i] == '+' || s[i] == '/')
                {
                        pop(str1);
                        pop(str2);
                        temp[0] ='(';
                        temp[1] ='\0';
                        strcpy(str, temp);
                        strcat(str, str2);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcat(str,temp);
                        strcat(str, str1);
                        temp[0] =')';
                        temp[1] ='\0';
                        strcat(str,temp);
                        push(str);
                }
                else
                {
                        temp[0]=s[i];
                        temp[1]='\0';
                        strcpy(s1, temp);
                        push(s1);
                }
                i++;
        }
        printf("\nThe Infix expression is: %s", stack[0]);

}

void pop(char *a1)
{
        strcpy(a1,stack[top]);
        top--;
}

void push (char*str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.3.7.   Conversion from postfix to prefix:

Procedure to convert postfix expression to prefix expression is as follows:

1.   Scan the postfix expression from left to right.

2.   If the scanned symbol is an operand, then push it onto the stack.

3.   If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in front of the operands and push it onto the stack.

5.   Repeat steps 2 and 3 till the end of the expression.

Example:

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.

4.3.8. Program to convert postfix to prefix expression:

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

#define MAX 100
void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top =-1;

main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2], temp[2];
        int i = 0;
        clrscr();
        printf("Enter the postfix expression; ");
        gets (s);
        while(s[i]!='\0')
        {
        /*skip whitespace, if any */
                if (s[i] == ' ')
                        i++;
                if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i]== '+' || s[i] == '/')
                {
                        pop (str1);
                        pop (str2);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcpy (str, temp);
                        strcat(str, str2);
                        strcat(str, str1);
                        push(str);
                }
                else
                {
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcpy (s1, temp);
                        push (s1);
                }
                i++;
        }
        printf("\n The prefix expression is: %s", stack[0]);
}

void pop(char*a1)
{
        if(top == -1)
        {
                printf("\nStack is empty");
                return ;
        }
        else
        {
                strcpy (a1, stack[top]);
                top--;
        }
}
```

```
void push (char *str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.3.9.   Conversion from prefix to infix:

Procedure to convert prefix expression to infix expression is as follows:

1.   Scan the prefix expression from right to left (reverse order).

2.   If the scanned symbol is an operand, then push it onto the stack.

3.   If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.

4.   Repeat steps 2 and 3 till the end of the expression.

Example:

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.	

4.3.10. Program to convert prefix to infix expression:

```c
# include <string.h>
# define MAX 100

void pop (char*);
void push(char*);
char stack[MAX] [MAX];
int top = -1;

void main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2],temp[2];
        int i=0;
        clrscr( ) ;
        printf("\Enter the prefix expression; ");
        gets(s);
        strrev(s);
        while (s[i]!='\0')
        {
                /*skip whitespace, if any*/
                if(s[i] == ' ' )
                        i++;
                if (s[i] == '^' || s[i] == '*'|| s[i] == '-' || s[i] == '+' || s[i] == '/')
                {
                        pop(str1);
                        pop(str2);
                        temp[0] ='(';
                        temp[1] ='\0';
                        strcpy(str, temp);
                        strcat(str, str1);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcat(str,temp);
                        strcat(str, str2);
                        temp[0] =')';
                        temp[1] ='\0';
                        strcat(str,temp);
                        push(str);
                }
                else
                {
                        temp[0]=s[i];
                        temp[1]='\0';
                        strcpy(s1, temp);
                        push(s1);
```

```
                }
                i++;
        }
        printf("\nThe infix expession is: %s", stack[0]);
}

void pop(char *a1)
{
        strcpy(a1,stack[top]);
        top--;
}

void push (char*str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.3.11.   Conversion from prefix to postfix:

Procedure to convert prefix expression to postfix expression is as follows:

1.   Scan the prefix expression from right to left (reverse order).

2.   If the scanned symbol is an operand, then push it onto the stack.

3.   If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator after the operands and push it onto the stack.

4.   Repeat steps 2 and 3 till the end of the expression.

Example:

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.

4.3.12.   Program to convert prefix to postfix expression:

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

#define MAX 100

void pop (char *a1);
void push(char *str);
char stack[MAX][MAX];
int top =-1;

void main()
{
        char s[MAX], str1[MAX], str2[MAX], str[MAX];
        char s1[2], temp[2];
        int i = 0;
        clrscr();
        printf("Enter the prefix expression; ");
        gets (s);
        strrev(s);
        while(s[i]!='\0')
        {
                if (s[i] == ' ')                  /*skip whitespace, if any */
                        i++;
                if(s[i] == '^' || s[i] == '*' || s[i] == '-' || s[i]== '+' || s[i] == '/')
                {
                        pop (str1);
                        pop (str2);
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcat(str1,str2);
                        strcat (str1, temp);
                        strcpy(str, str1);
                        push(str);
                }
```

```
                else
                {
                        temp[0] = s[i];
                        temp[1] = '\0';
                        strcpy (s1, temp);
                        push (s1);
                }
                i++;
        }
        printf("\nThe postfix expression is: %s", stack[0]);
}
void pop(char*a1)
{
        if(top == -1)
        {
                printf("\nStack is empty");
                return ;
        }
        else
        {
                strcpy (a1, stack[top]);
                top--;
        }
}
void push (char *str)
{
        if(top == MAX - 1)
                printf("\nstack is full");
        else
        {
                top++;
                strcpy(stack[top], str);
        }
}
```

4.4.   Evaluation of postfix expression:

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

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

4.4.1.   Program to evaluate a postfix expression:

```
# include <conio.h>
# include <math.h>
# define MAX 20

int isoperator(char ch)
{
        if(ch == '+' || ch == '-' ||  ch == '*' || ch == '/' ||  ch == '^')
                return 1;
        else
                return 0;
}
```

```
void main(void)
{
        char postfix[MAX];
        int val;
        char ch;
        int i = 0, top = 0;
        float val_stack[MAX], val1, val2, res;
        clrscr();
        printf("\n Enter a postfix expression: ");
        scanf("%s", postfix);
        while((ch = postfix[i]) != '\0')
        {
                if(isoperator(ch) == 1)
                {
                        val2 = val_stack[--top];
                        val1 = val_stack[--top];
                        switch(ch)
                        {
                                case '+':
                                        res = val1 + val2;
                                        break;
                                case '-':
                                        res = val1 - val2;
                                        break;
                                case '*':
                                        res = val1 * val2;
                                        break;
                                case '/':
                                        res = val1 / val2;
                                        break;
                                case '^':
                                        res = pow(val1, val2);
                                        break;
                        }
                        val_stack[top] = res;
                }
                else
                        val_stack[top] = ch-48;  /*convert character digit to integer digit */
                top++;
                i++;
        }
        printf("\n Values of %s is : %f ",postfix, val_stack[0] );
        getch();
}
```

4.5. Applications of stacks:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.

2. Stack is used to evaluate a postfix expression.

3. Stack is used to convert an infix expression into postfix/prefix form.

4. In recursion, all intermediate arguments and return values are stored on the processor's stack.

5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.
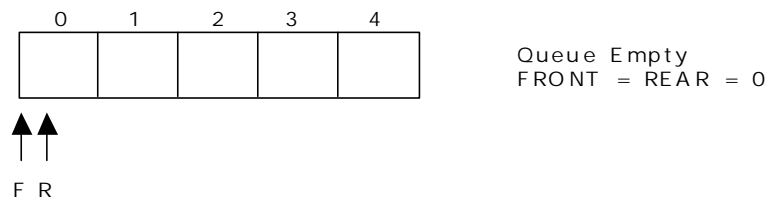
4.6. Queue:

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:
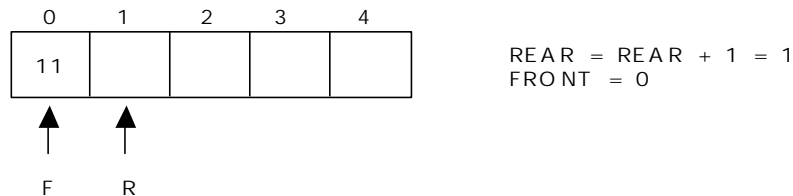
- enqueue: which inserts an element at the end of the queue.

- dequeue: which deletes an element at the start of the queue.
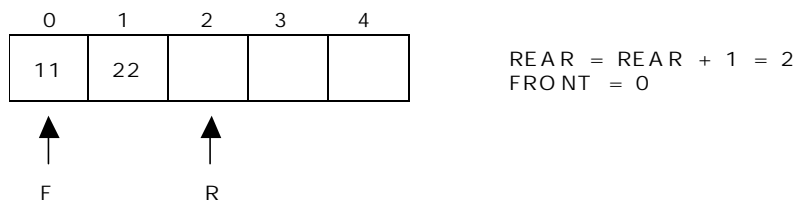
4.6.1.   Representation of Queue:

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.
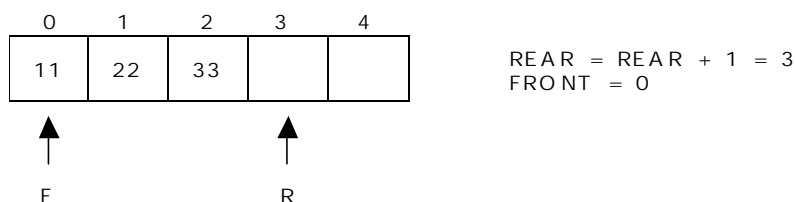
```
      0     1     2     3     4
   +-----+-----+-----+-----+-----+          Queue Empty
   |     |     |     |     |     |          FRONT = REAR = 0
   +-----+-----+-----+-----+-----+
     ↑↑
     F R
```

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

```
      0     1     2     3     4
   +-----+-----+-----+-----+-----+          REAR = REAR + 1 = 1
   | 11  |     |     |     |     |          FRONT = 0
   +-----+-----+-----+-----+-----+
     ↑     ↑
     F     R
```

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

```
      0     1     2     3     4
   +-----+-----+-----+-----+-----+          REAR = REAR + 1 = 2
   | 11  | 22  |     |     |     |          FRONT = 0
   +-----+-----+-----+-----+-----+
     ↑           ↑
     F           R
```

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

```
      0     1     2     3     4
   +-----+-----+-----+-----+-----+          REAR = REAR + 1 = 3
   | 11  | 22  | 33  |     |     |          FRONT = 0
   +-----+-----+-----+-----+-----+
     ↑                 ↑
     F                 R
```

Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:
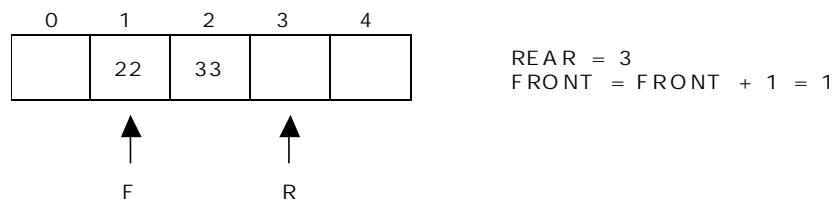
0	1	2	3	4
	22	33		

F (at index 1) R (at index 3)

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:

0	1	2	3	4
		33		

F (at index 2) R (at index 3)

REAR = 3
FRONT = FRONT + 1 = 2
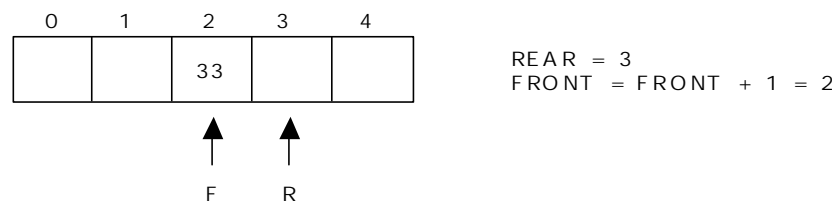
Now, insert new elements 44 and 55 into the queue. The queue status is:
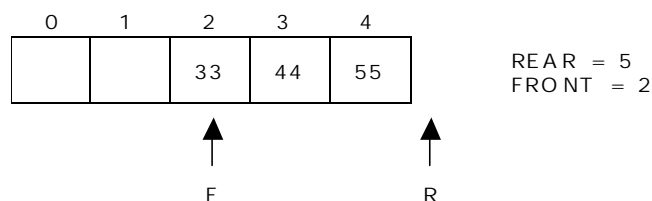
0	1	2	3	4
		33	44	55

F (at index 2) R (at index 4)

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:

0	1	2	3	4
		33	44	55

F (at index 2) R (at index 4)

REAR = 5
FRONT = 2
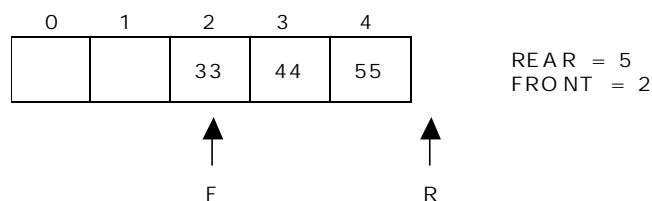
Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:

0	1	2	3	4
33	44	55	66	

F (at index 0) R (at index 3)

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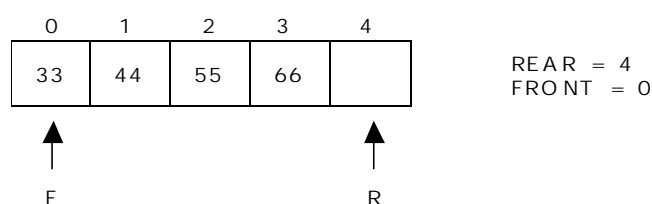
This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a circular queue.

4.6.2. Source code for Queue operations using array:

In order to create a queue we require a one dimensional array Q(1:n) and two variables front and rear. The conventions we shall adopt for these two variables are that front is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0. The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.

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

3. displayQ(): displays the elements in the queue.

```c
# 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:


                        case 2:


                        case 3:


                        case 4:


                }
                getch();
        } while(1);
}
```

4.6.3. Linked List Implementation 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.
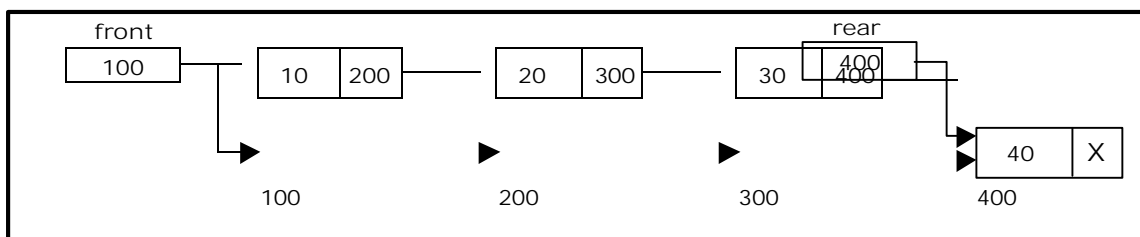
The linked queue looks as shown in figure 4.4:

Figure 4.4. Linked Queue representation

4.6.4.   Source code for queue operations using linked list:

```c
# 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);
}
```

```c
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');
}
```

4.7.   Applications of Queue:

1.   It is used to schedule the jobs to be processed by the CPU.

2.   When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.

3.   Breadth first search uses a queue data structure to find an element from a graph.

4.8.   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. 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:
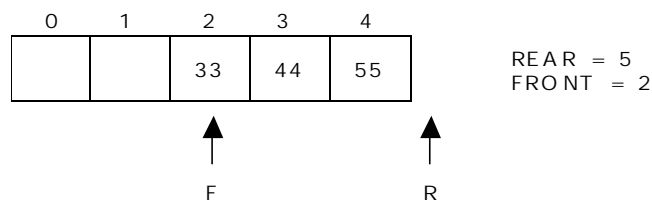
0	1	2	3	4
		33	44	55

F (at index 2)   R (at index 4)

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:

0	1	2	3	4
		33	44	55

F (at index 2)   R (at index 4)

REAR = 5
FRONT = 2
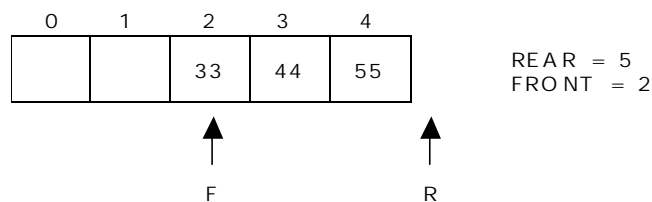
This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a circular queue.

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.
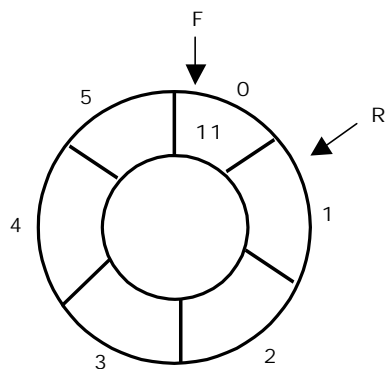
4.8.1.   Representation of Circular Queue:

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

F   R
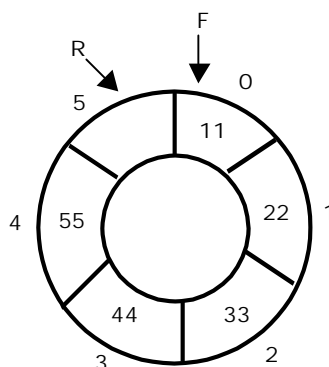
5   0
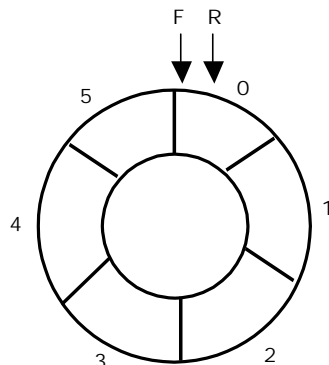
Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

4   1

3   2

Circular Queue

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

F

5   0

11   R

4   1

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

3   2

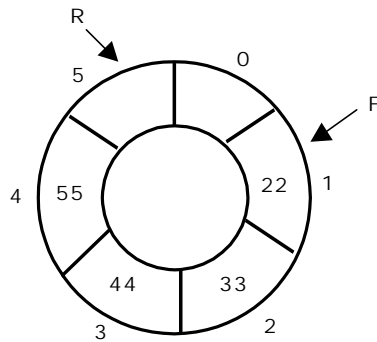Circular Queue

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

R   F

5   0

11

4   55   22   1

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

44   33

3   2

Circular Queue

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:

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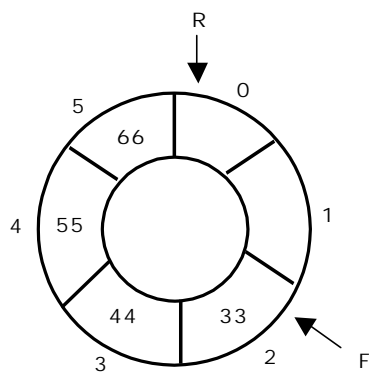COUNT = COUNT - 1 = 4

Circular Queue

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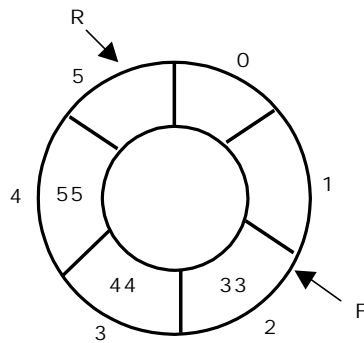
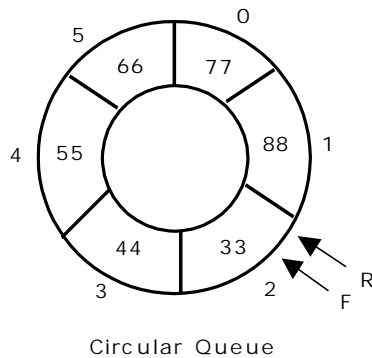FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3

Circular Queue

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

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

Circular Queue

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

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

Circular Queue

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.

4.8.2.   Source code for Circular Queue operations, using array:

```c
# 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 --;
        }
}
```

```c
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:


                        case 2:


                        case 3:


                        case 4:

                        default:

                }
                getch();
        } while(1);
}
```

4.9. Deque:

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a deque. The word deque is an acronym derived from double-ended queue. Figure 4.5 shows the representation of a deque.
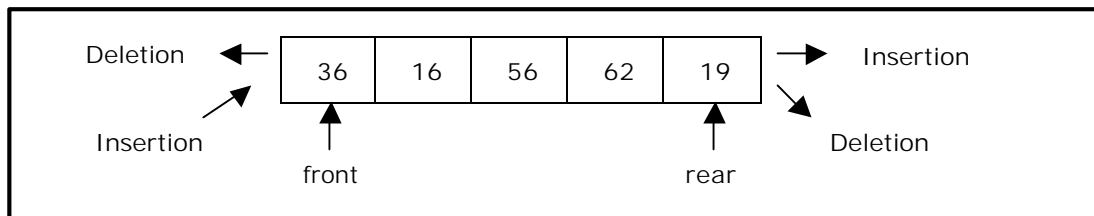

Figure 4.5. Representation of a deque.

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.
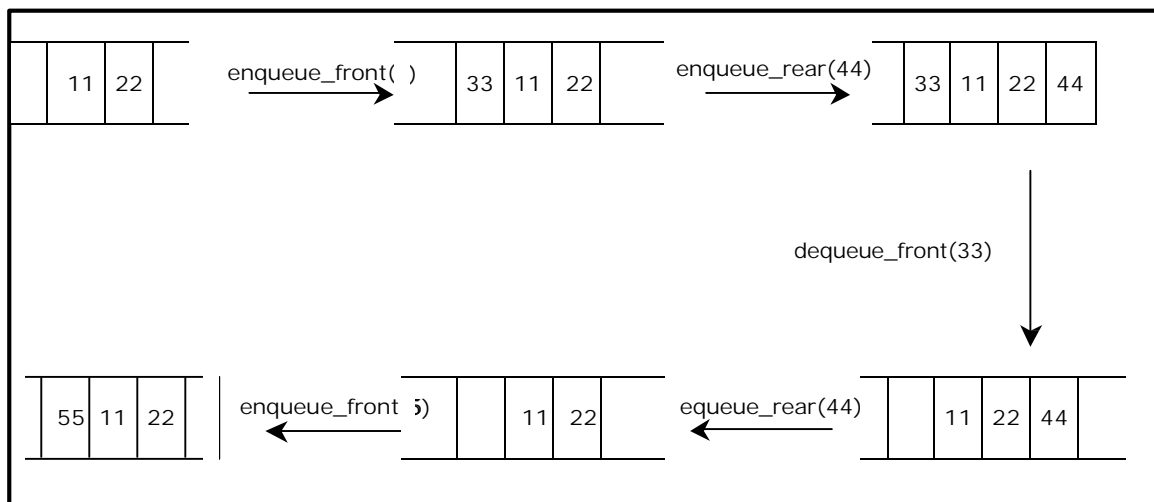

Figure 4.6. Basic operations on deque

There are two variations of deque. They are:

- Input restricted deque (IRD)
- Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

4.10. Priority Queue:

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1.   An element of higher priority is processed before any element of lower priority.

2.   two elements with same priority are processed   according to the order in which they were added to the queue.

A prototype of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

Exercises

1.   What is a linear data structure? Give two examples of linear data structures.

2.   Is it possible to have two designs for the same data structure that provide the same functionality but are implemented differently?

3.   What is the difference between the logical representation of a data structure and the physical representation?

4.   Transform the following infix expressions to reverse polish notation:
    a) $A \uparrow B * C - D + E / F / (G + H)$
    b) $((A + B) * C - (D - E)) \uparrow (F + G)$
    c) $A - B / (C * D \uparrow E)$
    d) $(a + b \uparrow c \uparrow d) * (e + f / d))$
    f) $3 - 6 * 7 + 2 / 4 * 5 - 8$
    g) $(A - B) / ((D + E) * F)$
    h) $((A + B) / D) \uparrow ((E - F) * G)$

5.   Evaluate the following postfix expressions:
    a) P_1: 5, 3, +, 2, *, 6, 9, 7, -, /, -
    b) P_2: 3, 5, +, 6, 4, -, *, 4, 1, -, 2, $\uparrow$, +
    c) P_3 : 3, 1, +, 2, $\uparrow$, 7, 4, -, 2, *, +, 5, -

6.   Consider the usual algorithm to convert an infix expression to a postfix expression. Suppose that you have read 10 input characters during a conversion and that the stack now contains these symbols:

+
(
*

bottom
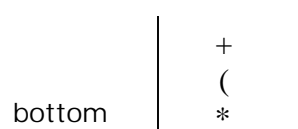
Now, suppose that you read and process the 11th symbol of the input. Draw the stack for the case where the 11th symbol is:
    A. A number:
    B. A left parenthesis:
    C. A right parenthesis:
    D. A minus sign:
    E. A division sign:

7. Write a program using stack for parenthesis matching. Explain what modifications would be needed to make the parenthesis matching algorithm check expressions with different kinds of parentheses such as (), [] and {}'s.

8. Evaluate the following prefix expressions:
   a) + * 2 + / 14 2 5 1
   b) - * 6 3 – 4 1
   c) + + 2 6 + - 13 2 4

9. Convert the following infix expressions to prefix notation:
   a) ((A + 2) * (B + 4)) -1
   b) Z – ((((X + 1) * 2) – 5) / Y)
   c) ((C * 2) + 1) / (A + B)
   d) ((A + B) * C – (D - E)) ↑ (F + G)
   e) A – B / (C * D ↑ E)

10. Write a "C" function to copy one stack to another assuming
    a) The stack is implemented using array.
    b) The stack is implemented using linked list.

11. Write an algorithm to construct a fully parenthesized infix expression from its postfix equivalent. Write a "C" function for your algorithm.

12. How can one convert a postfix expression to its prefix equivalent and vice-versa?

13. A double-ended queue (deque) is a linear list where additions and deletions can be performed at either end. Represent a deque using an array to store the elements of the list and write the "C" functions for additions and deletions.

14. In a circular queue represented by an array, how can one specify the number of elements in the queue in terms of "front", "rear" and MAX-QUEUE-SIZE? Write a "C" function to delete the K-th element from the "front" of a circular queue.

15. Can a queue be represented by a circular linked list with only one pointer pointing to the tail of the queue? Write "C" functions for the "add" and "delete" operations on such a queue

16. Write a "C" function to test whether a string of opening and closing parenthesis is well formed or not.

17. Represent N queues in a single one-dimensional array. Write functions for "add" and "delete" operations on the ith queue

18. Represent a stack and queue in a single one-dimensional array. Write functions for "push", "pop" operations on the stack and "add", "delete" functions on the queue.

1. Which among the following is a linear data structure: [D]
    A. Queue
    B. Stack
    C. Linked List
    D. all the above

2. Which among the following is a Dynamic data structure: [A]
    A. Double Linked List   C. Stack
    B. Queue   D. all the above

3. Stack is referred as: [A]
    A. Last in first out list   C. both A and B
    B. First in first out list   D. none of the above

4. A stack is a data structure in which all insertions and deletions of entries   [A]
    are made at:
    A. One end   C. Both the ends
    B. In the middle   D. At any position

5. A queue is a data structure in which all insertions and deletions are made   [A]
    respectively at:
    A. rear and front   C. front and rear
    B. front and front   D. rear and rear

6. Transform the following infix expression to postfix form: [D]
         (A + B) * (C − D) / E
    A. A B * C + D / -   C. A B + C D * - / E
    B. A B C * C D / - +   D. A B + C D - * E /

7. Transform the following infix expression to postfix form: [B]
         A - B / (C * D)
    A. A B * C D - /   C. / - D C * B A
    B. A B C D * / -   D. - / * A B C D

8. Evaluate the following prefix expression: * - + 4 3 5 / + 2 4 3   [A]

    A. 4   C. 1
    B. 8   D. none of the above

9. Evaluate the following postfix expression: 1 4 18 6 / 3 + + 5 / +   [C]
    A. 8   C. 3
    B. 2   D. none of the above

10. Transform the following infix expression to prefix form: [B]
         ((C * 2) + 1) / (A + B)
    A. A B + 1 2 C * + /   C. / * + 1 2 C A B +
    B. / + * C 2 1 + A B   D. none of the above

11. Transform the following infix expression to prefix form: [D]
         Z − ((((X + 1) * 2) − 5) / Y)
    A. / - * + X 1 2 5 Y   C. / * - + X 1 2 5 Y
    B. Y 5 2 1 X + * - /   D. none of the above

12. Queue is also known as: [B]
    A. Last in first out list   C. both A and B
    B. First in first out list   D. none of the above

13. One difference between a queue and a stack is: [C]
    A. Queues require dynamic memory, but stacks do not.
    B. Stacks require dynamic memory, but queues do not.
    C. Queues use two ends of the structure; stacks use only one.
    D. Stacks use two ends of the structure, queues use only one.

14. If the characters 'D', 'C', 'B', 'A' are placed in a queue (in that order), and   [D]
    then removed one at a time, in what order will they be removed?
    A. ABCD   C. DCAB
    B. ABDC   D. DCBA

15. Suppose we have a circular array implementation of the queue class,   [D]
    with ten items in the queue stored at data[2] through data[11]. The
    CAPACITY is 42. Where does the push member function place the new
    entry in the array?
    A. data[1]   C. data[11]
    B. data[2]   D. data[12]

16. Consider the implementation of the queue using a circular array. What   [B]
    goes wrong if we try to keep all the items at the front of a partially-filled
    array (so that data[0] is always the front).
    A. The constructor would require linear time.
    B. The get_front function would require linear time.
    C. The insert function would require linear time.
    D. The is_empty function would require linear time.

17. In the linked list implementation of the queue class, where does the push   [A]
    member function place the new entry on the linked list?
    A. At the head
    B. At the tail
    C. After all other entries that are greater than the new entry.
    D. After all other entries that are smaller than the new entry.

18. In the circular array version of the queue class (with a fixed-sized array),   [   ]
    which operations require linear time for their worst-case behavior?
    A. front   C. empty
    B. push   D. None of these.

19. In the linked-list version of the queue class, which operations require   [   ]
    linear time for their worst-case behavior?
    A. front   C. empty
    B. push   D. None of these operations.

20. To implement the queue with a linked list, keeping track of a front   [B]
    pointer and a rear pointer. Which of these pointers will change during an
    insertion into a NONEMPTY queue?
    A. Neither changes   C. Only rear_ptr changes.
    B. Only front_ptr changes.   D. Both change.

21. To implement the queue with a linked list, keeping track of a front   [D]
    pointer and a rear pointer. Which of these pointers will change during an
    insertion into an EMPTY queue?
    A. Neither changes   C. Only rear_ptr changes.
    B. Only front_ptr changes.   D. Both change.

22. Suppose top is called on a priority queue that has exactly two entries   [B]
with equal priority. How is the return value of top selected?
A. The implementation gets to choose either one.
B. The one which was inserted first.
C. The one which was inserted most recently.
D. This can never happen (violates the precondition)

23. Entries in a stack are "ordered". What is the meaning of this statement?   [D]
A. A collection of stacks can be sorted.
B. Stack entries may be compared with the '<' operation.
C. The entries must be stored in a linked list.
D. There is a first entry, a second entry, and so on.

24. The operation for adding an entry to a stack is traditionally called: [D]
A. add   C. insert
B. append   D. push

25. The operation for removing an entry from a stack is traditionally called: [C]
A. delete   C. pop
B. peek   D. remove

26. Which of the following stack operations could result in stack underflow?   [A]
A. is_empty   C. push
B. pop   D. Two or more of the above answers

27. Which of the following applications may use a stack?   [D]
A. A parentheses balancing program.
B. Keeping track of local variables at run time.
C. Syntax analyzer for a compiler.
D. All of the above.

28. Here is an infix expression: 4 + 3 * (6 * 3 - 12). Suppose that we are   [D]
using the usual stack algorithm to convert the expression from infix to
postfix notation. What is the maximum number of symbols that will
appear on the stack AT ONE TIME during the conversion of this
expression?
A. 1   C. 3
B. 2   D. 4

29. What is the value of the postfix expression 6 3 2 4 + - *   [A]
A. Something between -15 and -100
B. Something between -5 and -15
C. Something between 5 and -5
D. Something between 5 and 15
E. Something between 15 and 100

30. If the expression ((2 + 3) * 4 + 5 * (6 + 7) * 8) + 9 is evaluated with *   [A]
having precedence over +, then the value obtained is same as the value
of which of the following prefix expressions?
A. + + * + 2 3 4 * * 5 + 6 7 8 9   C. * + + + 2 3 4 * * 5 + 6 7 8 9
B. + * + + 2 3 4 * * 5 + 6 7 8 9   D. + * + + 2 3 4 + + 5 * 6 7 8 9

31. Evaluate the following prefix expression: [B]
+ * 2 + / 14 2 5 1
A. 50   C. 40
B. 25   D. 15

32 Parenthesis are never needed prefix or postfix expression: [A]
   A. True   C. Cannot be expected
   B. False   D. None of the above

33 A postfix expression is merely the reverse of the prefix expression: [B]
   A. True   C. Cannot be expected
   B. False   D. None of the above

34 Which among the following data structure may give overflow error, even   [A]
   though the current number of elements in it, is less than its size:
   A. Simple Queue   C. Stack
   B. Circular Queue   D. None of the above

35. Which among the following types of expressions does not require   [C]
   precedence rules for evaluation:
   A. Fully parenthesized infix expression
   B. Prefix expression
   C. both A and B
   D. none of the above

36. Conversion of infix arithmetic expression to postfix expression uses: [D]
   A. Stack   C. linked list
   B. circular queue   D. Queue