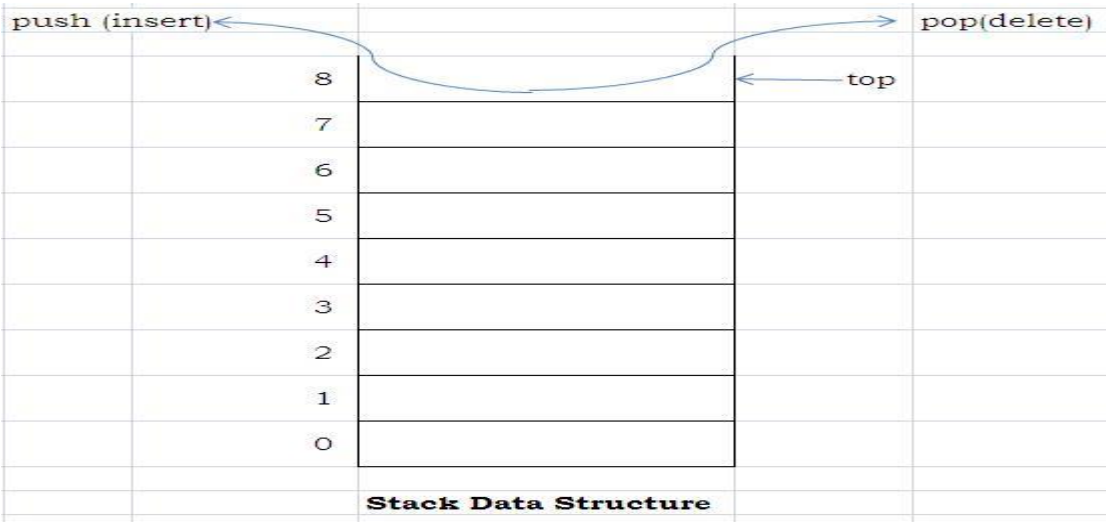
UNIT – II

STACK AS ABSTRACT DATATYPE

A stack is a data structure in which addition of new element or deletion of an existing element always takes place at the same end. This end is known as top of stack. When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop. Stack is also called as Last-In-First-Out (LIFO) list which means that the last element that is inserted will be the first element to be removed from the stack.



Abstract Datatype Stack

{

instances:

Linear list of elements, one end is called top and other end is called bottom.

operations:

empty()	– returns true if stack is empty otherwise false
size()	– returns the number of elements in the stack
top()	– returns top element of the stack
push(x)	– add element x at the top of the stack
pop()	– remove top element from the stack

}

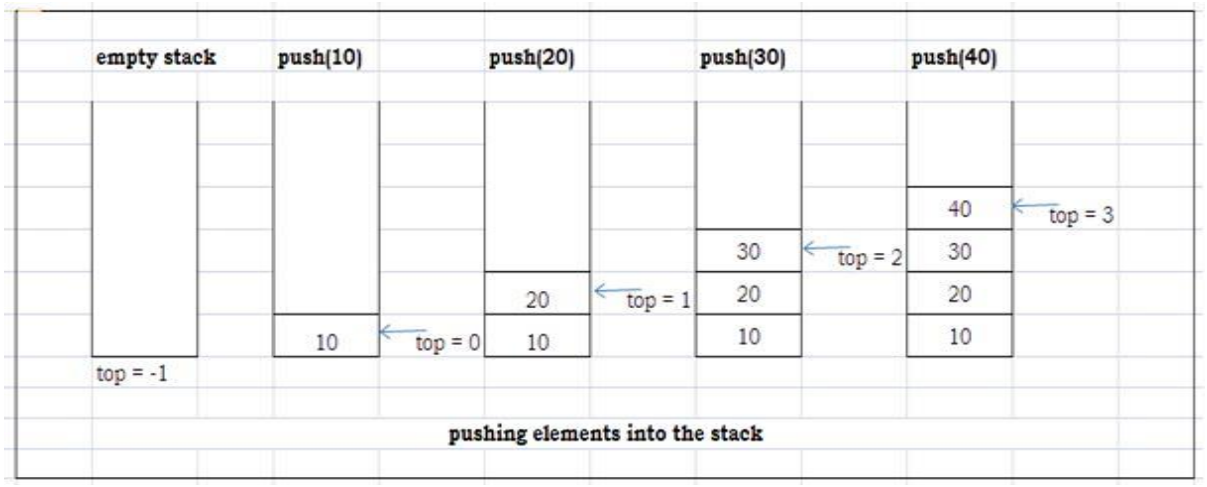
Representation of stacks (operations performed on stacks)

There are two possible operations performed on a stack. They are push and pop.

- ✓ Push: Allows adding an element at the top of the stack.
- ✓ Pop: Allows removing an element from the top of the stack.

Algorithm for PUSH Operation

1. stack overflow? If top=max_stacksize then write overflow and exit.
2. read item
3. set top = top + 1
4. set stack[top] = item
5. exit

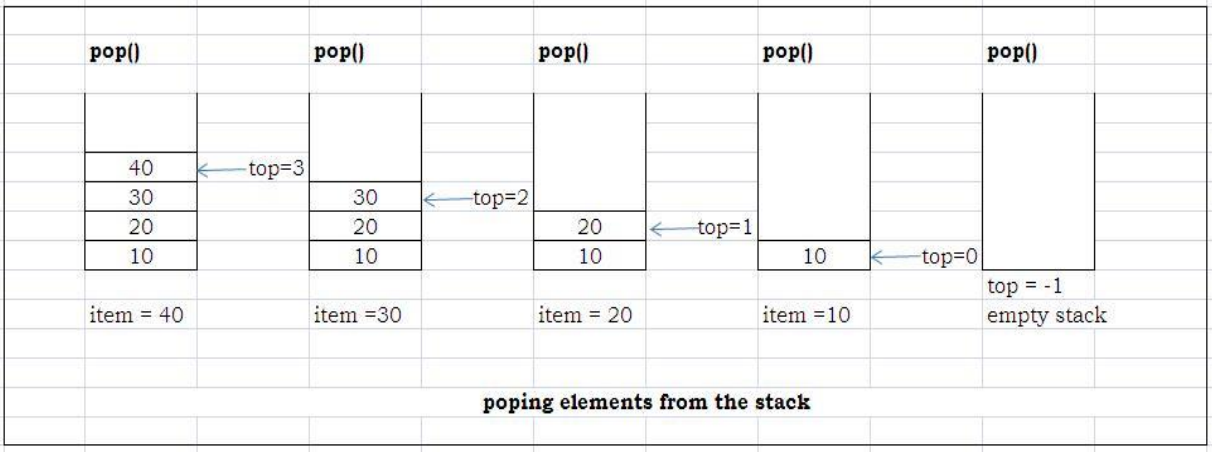


If the elements are added continuously to the stack using the push operation then the stack grows at one end. Initially when the stack is empty the top = -1. The top is a variable which indicates the position of the topmost element in the stack.

Algorithm for POP Operation

On deletion of elements the stack shrinks at the same end, as the elements at the top get removed.

1. stack underflow? If top = -1 then write underflow and exit
2. repeat step 3 to 5 until top >= 0
3. set item = stack[top]
4. top = top – 1
5. write deleted item
6. exit



Applications of Stacks

- ✓ Stack is used by compilers to check for balancing of parentheses, brackets and braces.
- ✓ Stack is used to evaluate a postfix expression.
- ✓ Stack is used to convert an infix expression into postfix/prefix form.
- ✓ In recursion, all intermediate arguments and return values are stored on the processor's stack.
- ✓ During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

Implementation of Stacks using Arrays

The stacks can be implemented by using arrays and linked lists. If arrays are used for implementing the stacks, it would be very easy to manage the stacks. But the problem with an array is that we are required to declare the size of the array before using it in a program. This means the size of the stack should be fixed.

C++ program to illustrate about stacks using arrays

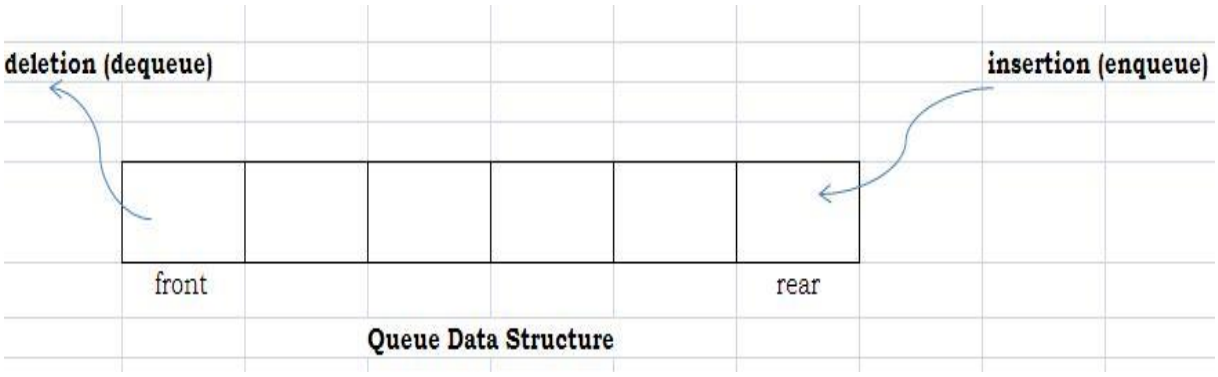
```
#include<iostream.h>
#include<conio.h>

class stack
{
        int stk[5], top;
    public:
        stack( )
        {
            top = -1;
        }
        void push(int x)
        {
            if(top>4)
            {
                cout<<"stack overflow";
                return;
```

```
                }
                stk[++top]=x;
                cout<<"Inserted "<<x;
            }
        void pop()
        {
            if(top<0)
            {
                cout<<"stack empty";
                return;
            }
            cout<<"deleted"<<stk[top--];
        }
        void display( )
        {
            if(top<0)
            {
                cout<<"stack empty";
                return;
            }

            for(int i=top; i>=0; i--)
                cout<<stk[i]<<" ";
        }
};
main( )
{
        int opt, ele;
        stack st;
        while(1)
        {
            cout<<"\n 1. push 2. pop 3. display 4. exit";
            cout<<" enter the option";
            cin>>opt;
            switch(opt)
            {
        case 1:                     cout<<" enter the element";
                                    cin>>ele;
                                    st.push(ele);
                                    break;
                case 2:             st.pop();
                                     break;
                case 3:             st.display();
                                     break;
                default:            exit(0);

            }
        }
}
```

QUEUE AS ABSTRACT DATA TYPE

Queue is a linear data structure that permits insertion of new element at one end and deletion of an element at the other end. The end at which the deletion of an element take place is called front, and the end at which insertion of a new element can take place is called rear. The deletion or insertion of elements can take place only at the front or rear end called dequeue and enqueue. The first element that gets added into the queue is the first one to get removed from the queue. Hence the queue is referred to as

First-In-First-Out list (FIFO).



Abstract Datatype Queue
{
instances:
Linear list of elements, one end is called front and other end is called rear.
operations:

empty()	– returns true if queue is empty otherwise false
size()	– returns the number of elements in the queue
front(x)	– returns first element of queue pointed by front
rear(x)	– add element x at the rear of the queue

}

Representation of Queue (operations performed on Queue)

There are two possible operations performed on a queue. They are enqueue and dequeue.

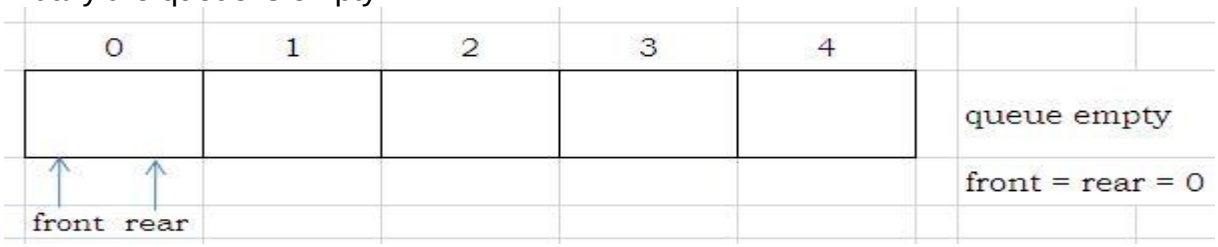
- ✓ enqueue: Allows inserting an element at the rear of the queue.
- ✓ dequeue: Allows removing an element from the front of the queue.

Algorithm for Enqueue (inserting an element)

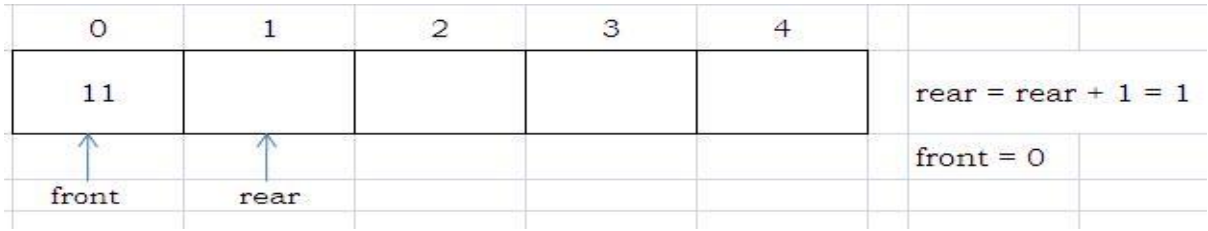
1. initialize front = 0,rear = -1.
2. check overflow condition? If front = 0and rear = max_size then write overflow and exit.

3. if front = NULL then set front = 0 and rear = 0 else if rear = max_size then set rear = 0
4. set rear = rear + 1
5. queue[rear] = item
6. exit

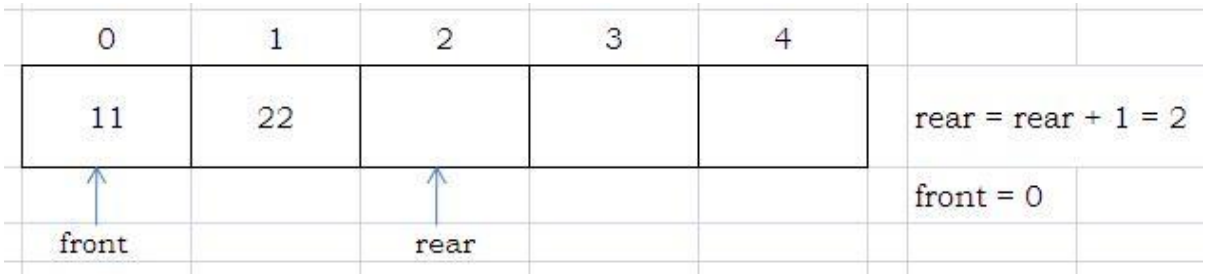
Let us consider a queue, which can hold maximum of five elements.
Initially the queue is empty.



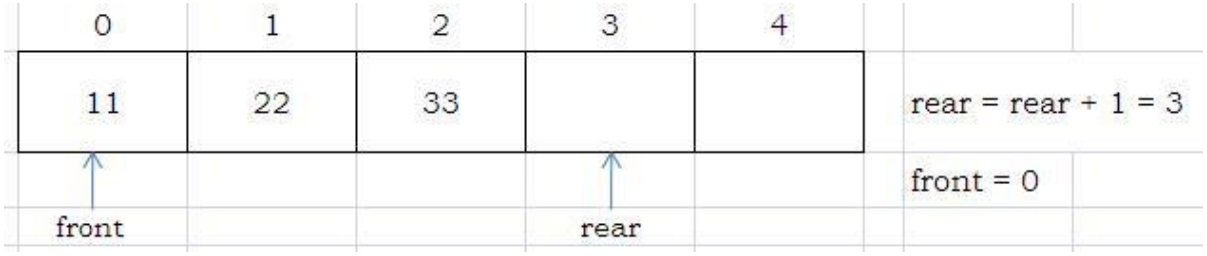
Now, insert 11 to the queue. Then queue status will be:



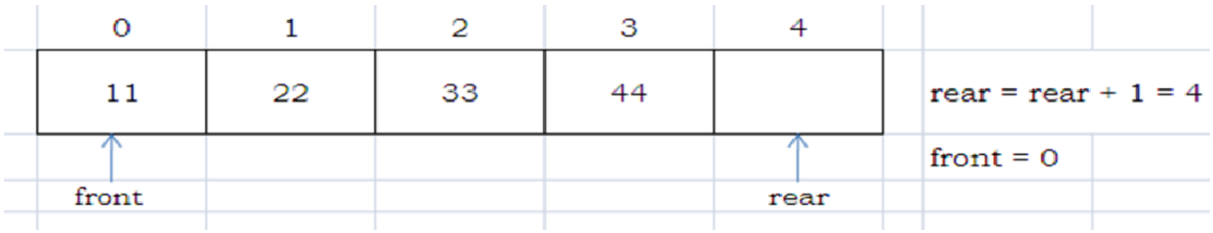
Next, insert 22 to the queue. Then the queue status is:



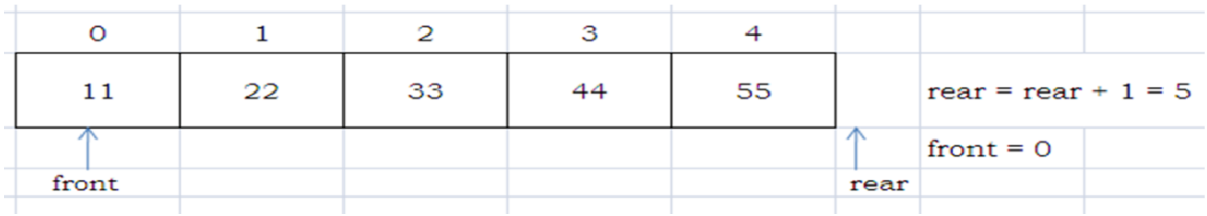
Again insert another element 33 to the queue. The status of the queue is:



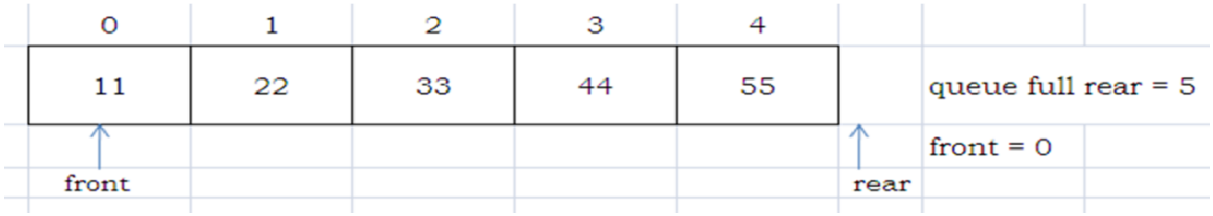
Again insert another element 44 to the queue. The status of the queue is:



Again insert another element 55 to the queue. The status of the queue is:



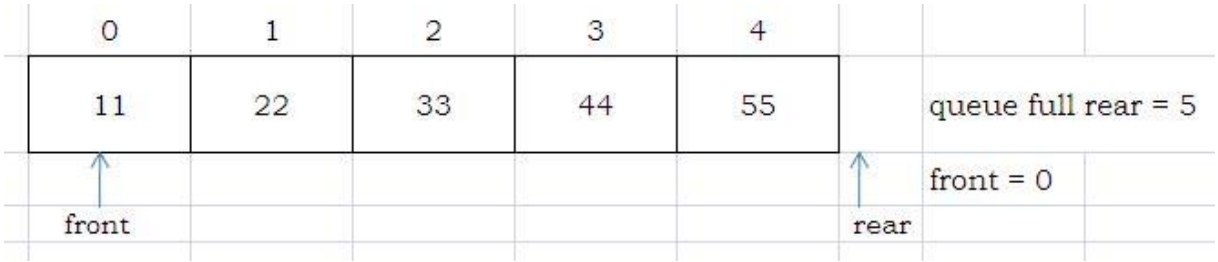
Again insert another element 66 to the queue. The status of the queue is:



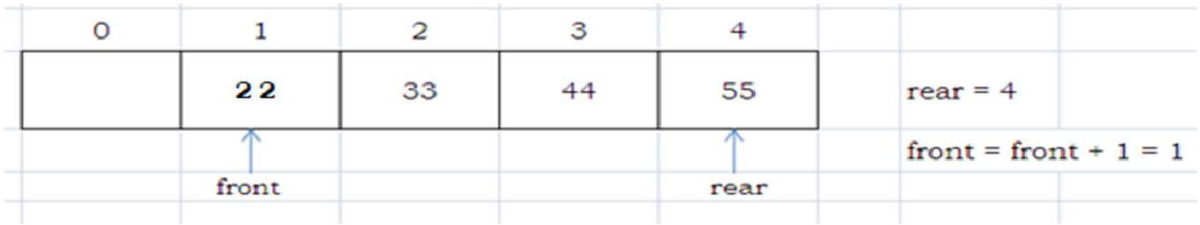
An element can be added to the queue only at the rear end of the queue. Before adding an element in the queue, it is checked whether queue is full. If the queue is full, then addition cannot take place. Otherwise, the element is added to the end of the list at the rear side.

Algorithm for Dequeue (deleting an element)

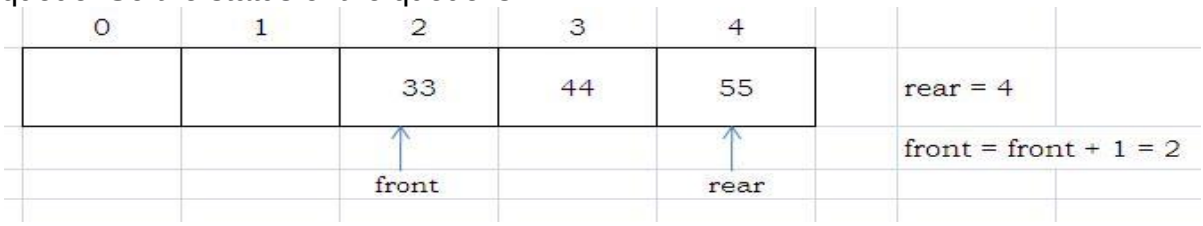
1. check underflow condition? if front < 0 then write underflow and exit
2. set item = queue[front]
3. if front = rear then set front = rear = NULL else if front = max_size then set front = 0
4. set front = front + 1
5. exit



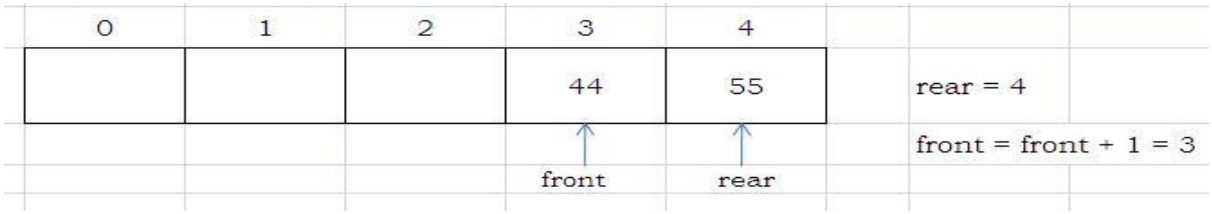
Now, delete an element 11. The element deleted is the element at the front of the queue. So the status of the queue is:



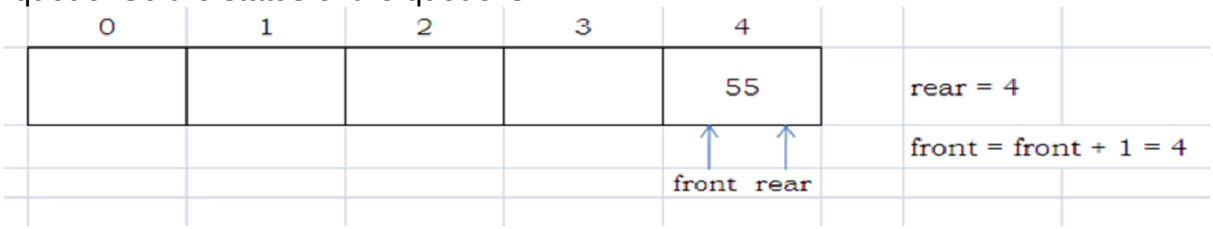
Now, delete an element 22. The element deleted is the element at the front of the queue. So the status of the queue is:



Now, delete an element 33. The element deleted is the element at the front of the queue. So the status of the queue is:



Now, delete an element 44. The element deleted is the element at the front of the queue. So the status of the queue is:



Now, delete an element 55. The element deleted is the element at the front of the queue. So the status of the queue is empty.

The dequeue operation deletes the element from the front of the queue. Before deleting and element, it is checked if the queue is empty. If not the element pointed by front is deleted from the queue and front is now made to point to the next element in the queue.

Implementation of Queues using Arrays

The stacks can be implemented by using arrays and linked lists. If arrays are used for implementing the queues, it would be very easy to manage the queues. But the problem with an array is that we are required to declare the size of the array before using it in a program. This means the size of the queue should be fixed.

C++ program to illustrate about queues using arrays

```
#include<iostream.h>
#include<conio.h>

class queue
{
        int que[5];
        int front, rear;

    public:
        queue()
        {
            front = rear = -1;
        }

        void enqueue(int x)
        {
            if(rear > 4)
            {
                cout<<"queue overflow";
                front = rear = -1;
                return;
            }
            que[++rear]=x;
            cout<<"inserted"<<x;
        }

        void dequeue()
        {
         if(front = rear)
         {
            cout<<"queue empty";
            return;
         }
         cout<<"deleted"<<que[front++];
        }
        void display()
        {
         if(rear = front)
         {
            cout<<"queue empty";
            return;
         }
           for(int i=front + 1; i<=rear; i++)
              cout<<que[i]<<" ";
    }
};
```

```
main()
{
        int opt, ele;
        queue qt;
        while(1)
        {
                cout<<"\n 1. enqueue 2. dequeue 3. display 4. exit";
                cout<<" enter the option";
                cin>>opt;
                switch(opt)
                {
                        case 1:         cout<<" enter the element";
                                        cin>>ele;
                                        qt.enqueue(ele);
                                        break;

                        case 2:         qt.dequeue();
                                         break;

                        case 3:         qt.display();
                                         break;

                        default:        exit(0);
                }
        }
}
```

EVALUATION OF EXPRESSIONS

"An expression is defined as the combination of operators and operands**"**.

"An expression is defined as the combination of variables, constants and operators arranged as per the syntax of the language".

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 performs a mathematical or logical operation between the operands. Examples of operators include +, -, *, /, ^ etc.

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

Infix: an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands. **Example**: (A + B) * (C - D)

Prefix: an arithmetic expression in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation. **Example**: * + A B – C D

Postfix: 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 unambiguously.

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 ($, ↑, ^)	Highest	1
*, /	next highest	2
+, -	Lowest	3

As programmers we write the expressions into two types. They are simple and complex expressions. Let us consider the complex expression as follows:

x = a / b – c + d * e – a * c

Description	Operator	Rank	Associatively
Function expression	( )	1	Left to Right
Array expression	[ ]		
Unary plus	+	2	Right to left
Unary minus	-		
Increment/Decrement	++/--		
Logical negation	!		
One's complement	~		
Pointer reference	*		
Address of	&		
Size of an object	Sizeof		
Type cast (conversion)	(type)		
Multiplication	*	3	Left to Right
Division	/		

Modulus	%		
Addition	+	4	Left to Right
Subtraction	-		
Left shift	<<	5	Left to Right
Right shift	>>		
Less than	<	6	Left to Right
Less than or equal to	<=		
Greater than	>		
Greater than or equal to	>=		
Equality	==	7	Left to Right
Not equal to	! =		
Bit wise AND	&	8	Left to Right
Bit wise XOR	^	9	Left to Right
Bit wise OR	|	10	Left to Right
Logical AND	&&	11	Left to Right
Logical OR	||	12	Left to Right
Conditional	? :	13	Right to Left
Assignment	=,*=,/=,%=,+=,-=,& etc	14	Right to Left
Comma operator	,	15	Left to Right

In the above expression we first understand the meaning of the expression and then the order of performing the operation. For example, a = 4, b = c = 2, d = e = 3 then the value of x is found as

((4 / 2) – 2) + (3 * 3) – (4 * 2)

=0 + 9 – 8

=1

Or

(4 / (2 – 2 + 3)) * (3 – 4) * 2

=(4 / 3) * (- 1) * 2

=- 2.66666

Mostly we prefer the first method because we know multiplication is performed before addition and division is performed before subtraction. In any programming language, we follow hierarchy of operators for evaluation of expressions. The operator precedence is shown in the above table.

EVALUATION OF POSTFIX EXPRESSION

The standard representation for writing expressions is infix notation which means that placing the operator in between the operands. But the compiler uses the postfix notation for evaluating the expression rather than the infix notation.

It is an easy task for evaluating the postfix expression than infix expression because there are no parentheses. To evaluate an expression we scan it from left to right. The postfix expression is evaluated easily by the use of a stack.

When an operand is seen, it is pushed onto the stack. When an operator is seen, the operator is applied to the two operands 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.

Example 1

Evaluate the postfix expression: 6 2 / 3 - 4 2 * +

Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	3			0
3	3	3		1
-	0			0
4	0	4		1
2	0	4	2	2
*	0	8		1
+	8			0

INFIX TO POSTFIX

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

1. Fully parenthesize the expression.
2. Move all the binary operators so that they replace their corresponding right parenthesis.
3. Delete all parenthesis.

For example

a / b – c + d * e – a * c

According to step1 of the algorithm ((((a / b) – c) + (d * e)) – (a * c)) Performing the step2 and step3 gives ab/c-de*+ac*-

Example (simple expression)

We have simple expression a + b * c, then the postfix expression is abc*+. The output translation of the given infix expression to postfix expression is as follows.

Token	Stack [0]	[1]	[2]	top	Output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

In the above example, we have stacked the operators as long as the precedence of operator at the top of the stack is less than the incoming operator.

Example (parenthesized expression)

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.

We have parenthesized expression a * (b + c) *d, then the postfix expression is abc+*d*.

Token	Stack			top	Output
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc+*
d	*			0	abc+*d
eos	*			0	abc+*d*

Parenthesis makes translation process more difficult because the equivalent postfix expression will be parenthesis free. The postfix of our example is abc+*d*. Here we stack the operators until we reach the right parenthesis. At that point we unstuck till we reach the left parenthesis.