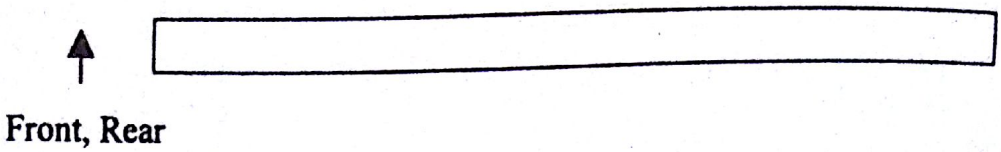


*Figure 4.2(a) Empty Queue*

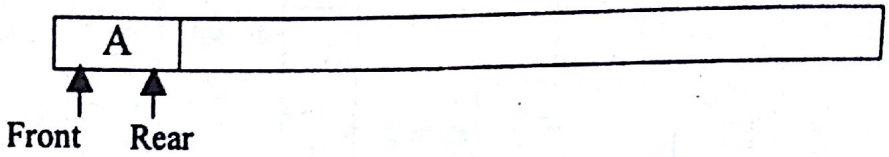


*Figure 4.2 (b) The queue after insertion of 'A'*

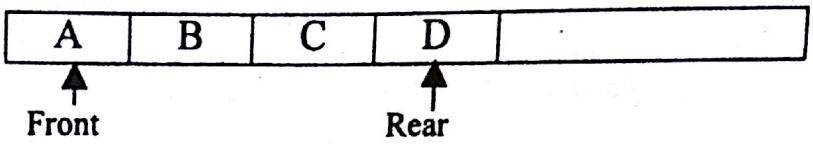


*Figure 4.2 (c) The queue after insertion of 'B', 'C', 'D'*

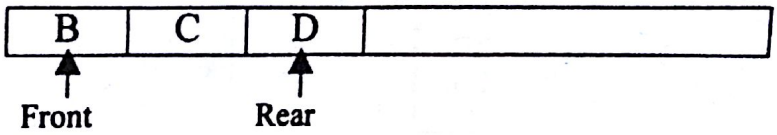


*Figure 4.2 (d) The queue after deletion of 'A'*

*Figure 4.2 Operations on a queue*

Stacks and queues can also be defined as Abstract Data Types (ADTs). A stack of elements of type T is a finite sequence of elements of T together with the following operations.

a. **Initialize** a stack as an empty stack.
b. Determine if a stack is **empty** or not.
c. Determine if a stack is **full** or not.
d. If a stack is not full, then **push** a new node at one end of the stack, called its top.
e. If a stack is not empty, then retrieve the node at its top.
f. If a stack is not empty, then **pop** the node at its top.
g. Copy one stack to another stack.

A queue of elements of type T is a finite sequence of elements of T together with the following operations.

a. Initialize a queue to be empty.
b. Determine if a queue is empty or not.
c. Determine if a queue is full or not.
d. Add a new node after the last node in a queue (i.e., at the "rear" end of the queue) if it is not full.
e. Retrieve the first node of a queue, if it is not empty.
f. Remove the first node in a queue (i.e., at the "front" end of the queue), if it is not empty.

## 4.2. Array representation

The most primitive representation of a stack can be done by using a one-dimensional array of data records. In addition to the one-dimensional array, which is used to hold the elements of the stack, another variable "top" is required to keep track of the index of the last inserted element (i.e. the topmost element). Formally, a stack may be defined as follows:

```
typedef struct stack {
        dataType AnArray[MAX_STACK_SIZE];
        int top;
}stack;
stack aStack;
```

Type of each element in the stack is assumed to be "dataType", a predefined type. There must exist an upper limit of the number of elements which may be stored in the stack. In the given declaration, the upper limit is assumed to be MAX_STACK_SIZE, a predefined constant. In this case, "aStack" is a variable of type "stack" and aStack.AnArray[top] is the topmost element. When "aStack" is initialized, aStack.top is set to −1. After each insertion (deletion) aStack.top increases (decreases) by 1. Therefore, to check whether a stack is empty or not one can simply compare aStack.top with −1. If aStack.top is equal to −1 then the stack is empty; otherwise, the stack is non-empty. "C" functions for pushing into or popping from a stack are presented next.

```
int push (dataType a, stack *x)
{
        if (x->top == MAX_STACK_SIZE)
                return 0;
        else
        {
                x->top += 1;
                x->AnArray[x->top] = a;
```

```
                return 1;
        }
}
int pop (stack *x, dataType *a)
{
        if (x->top == -1)
                return 0;
        else
        {
                *a = x->AnArray [x->top];
                x->top  = x->top - 1;
                return 1;
        }
}
```

The above functions take a constant amount of time, being independent of the number of elements of the stack.

Similar to a stack, a queue may also be conveniently represented by a one-dimensional array which may hold the data items. Two other variables "front" and "rear" are also required to point to the beginning and end of the queue. A queue data type may be defined formally as follows.

```
typedef struct queue {
        dataType Elements[MAX_QUEUE_SIZE+1];
        int front, rear;
}queue;
```

During the initialization of a queue, Q, its "front" and "rear" are set to (-1). On each addition to a queue, which logically takes place at the rear end of the queue, "rear" is incremented by one. On the other hand, after each removal operation, "front" is incremented by one. It has to be mentioned here that addition must not be done if the queue is full and similarly removal should not be done if the queue is empty. Therefore, conditions for empty queue and full queue have to be carefully formulated. The following points have to be observed in this context.

i. if the queue is non-empty then, "front" always denotes one less than the index of the beginning of the queue.
ii. the length of the queue is ("rear"-"front"), and
iii. If the condition ("front" == "rear") is TRUE then the queue is empty.

First confirm that the above observations hold for an initialized queue which is obviously empty as well. Whenever an item is added then "rear" is increased by

one and "front" does not get affected. Thus, length of the queue increases by one as length = "rear"-"front" which is correct. Whenever an item is deleted, "front" is incremented. Length of the queue decreases by one. If the new "front" becomes equal to "rear" then the queue becomes empty.

"C" functions for adding an element to a queue and removing an element from a queue are presented next.

```
int addq (dataType item, queue *q)
{
        if (q->rear == MAX_QUEUE_SIZE)
                return 0;
        else
        {
                q->rear += 1;
                q->Elements [q->rear] = item;
                return 1;
        }
}
int deleteq (dataType *data, queue *q)
{
        if (q->front  != q->rear)
        {
                (*data) =  q->Elements [ ++ (q->front)];
                return 1;
        }
        return 0;
}
```

If insertions and deletions are implemented as above then a particular problem is encountered. Consider the operations "addq" and "deleteq" being repeatedly performed on a queue in the sequence as shown in Figure 4.3.

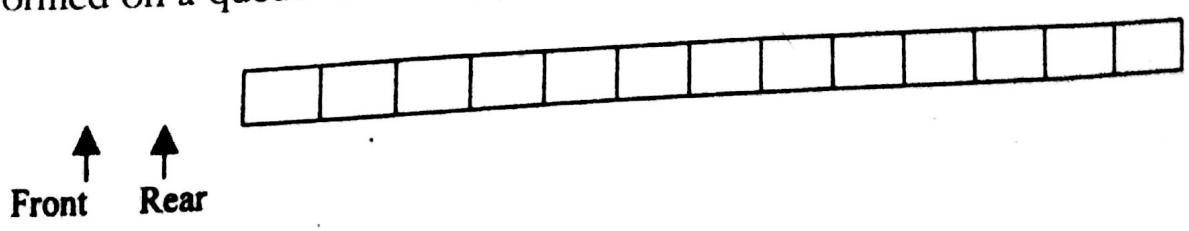


*Figure 4.3 (a) An empty queue*

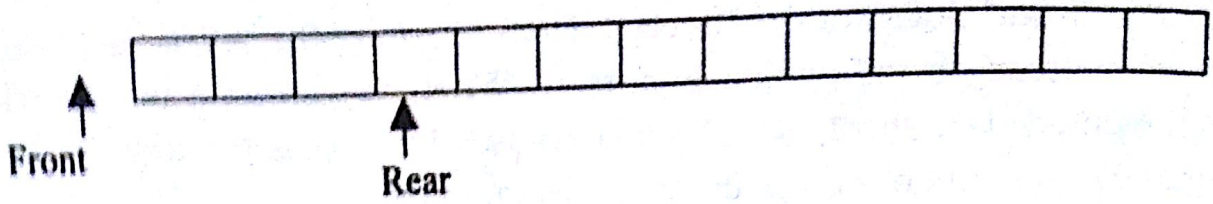


*Figure 4.3 (b) The queue after 4 successive insertions*

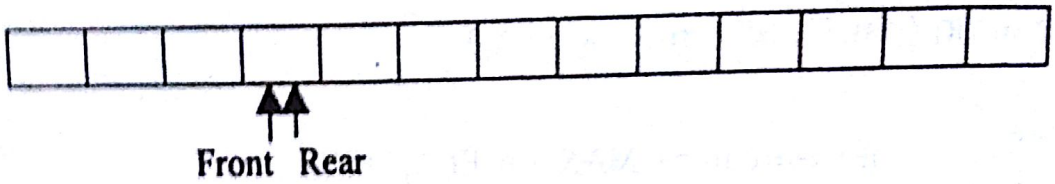


*Figure 4.3 (c) The queue after 4 successive deletions*

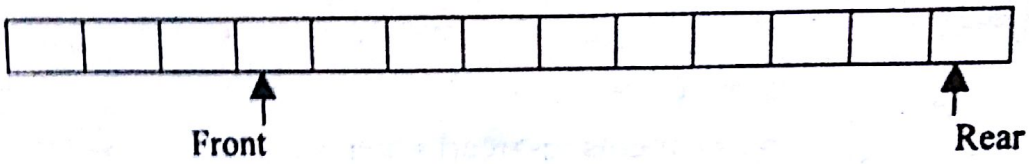


*Figure 4.3 (d) The queue after 9 successive insertions*

One more insertion at this point will become impossible as the "rear" has attained MAX_QUEUE_SIZE –1, although there are four empty spaces in the queue. Also note that the queue tends to shift towards the right end as "front" always tries to catch up with the "rear" of the queue. One simple way to overcome this problem is possible by resetting the "front" and "rear" to -1 once the queue is found to be empty. This can be achieved by rewriting the "deleteq" function in the following way.

```
int deleteq (queue *q, dataType *data)
{
        if (q->front == q->rear)
        {
                printf ("Queue is empty \n");
                q->front = -1;
                q->rear = -1;
                data = NULL;
                return 0;
        }
        (q -> front) += 1;
        (*data) = q->Elements [q->front];
        return 1;
}
```

Even with this version of "deleteq" function, it is quite possible that "addq" function does not add an item to the queue because of queue-full condition although there is some room in the queue. To overcome this problem, one may shift the "Elements" array towards left whenever "rear" is seen to be equal to "MAX_QUEUE_SIZE-1". But such a solution is certainly time consuming especially if the number of elements in a queue is large at the time when "queue_full" condition is fulfilled.

A more efficient queue representation using arrays is achieved by regarding the array of elements of a queue to be circular. Assume that the index of the array increases in clockwise direction except the index MAX_QUEUE_SIZE-1. Every element in the array has one clockwise and one anti-clockwise neighbour. The indices of the clockwise and the anti-clockwise neighbours of the i-th element of the array are (i+1)%MAX_QUEUE_SIZE and (i-1)%MAX_QUEUE_SIZE respectively. As before, "rear" points to the element which is at the rear end of the queue and "front" points to anti-clockwise neighbour of the beginning of the queue. Initially, let "front" and "rear" be initialized to zero. In this case also, "front" is equal to "rear" if and only if the queue is empty. Figure 4.4 shows a circular queue with four elements where front = 1 and rear = 5. Every insertion implies that "rear" of the queue increases by one except when rear = MAX_QUEUE_SIZE-1. If "rear" is equal to MAX_QUEUE_SIZE-1 then after one more insertion "rear" is made zero provided the queue is not yet full.

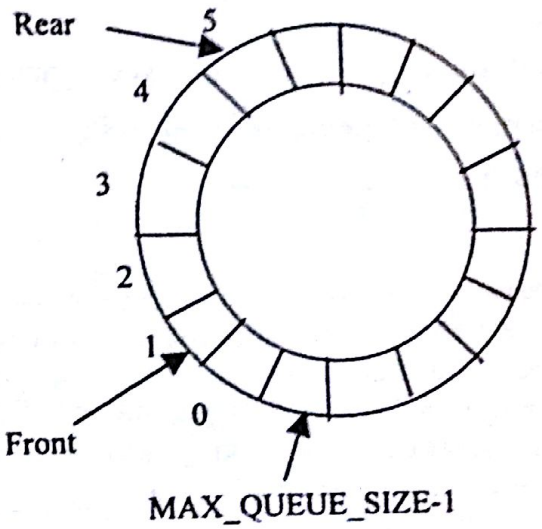


*Figure 4.4. An example of a circular queue*

The modified versions of the addition and deletion functions are given in the following.

```
int addToCircq (dataType item, queue *q)
{
        int t;
```

```
        t = (q->rear + 1)% MAX_QUEUE_SIZE;
        if (t == q->front)
                return 0;
        q->rear = t;
        q->Elements [q->rear] = item;
        return 1;
}
int  deleteCircq (dataType *item, queue *q)
{
        if (q->front == q->rear)
        {
                printf ("Queue is empty. \n");
                item = NULL;
                return 0;
        }
        else
        {
                q->front = (q -> front +1)% MAX_QUEUE_SIZE;
                *item =  (q->Elements [q->front]);
                return 1;
        }
}
```

It is interesting to note that apparently the conditions for "queue_full" and "queue_empty" are the same. But there is a subtle difference. During addition, the condition (q->front == q->rear) is checked after incrementing q->rear. This further means that all the available MAX_QUEUE_SIZE number of elements are not used in this implementation. In fact, the total capacity of the queue is now rendered one less than MAX_QUEUE_SIZE. In order to use all of the MAX_QUEUE_SIZE entries, a special variable "count" may be maintained for each queue to keep track of the number of elements in a queue. "count" may be made a component of the data type queue and "count" should be set to zero when a queue is initialized. During addition, "count" is checked against MAX_QUEUE_SIZE to determine whether the queue is full. Similarly, if "count" is zero then it implies that the queue is empty. Writing algorithm for addition and deletion operations on a circular queue using "count" is left as an exercise.

## 4.3 Linked list representation of stacks and queues

An alternative and efficient representation of stacks and queues is possible by using linked lists instead of arrays for storing their data elements. The advantages of such representations over array representation are similar to those of linked lists over arrays in general. Specifically, in such cases, the stack or queue is never full as long as the system has enough space for dynamic memory allocation. A stack can be simply represented by a singly connected linked list as shown in Figure 4.5. The pointer to the beginning of the list can serve the purpose of the "top" of the stack. To initialize a stack, one can simply set its "top" to NULL.

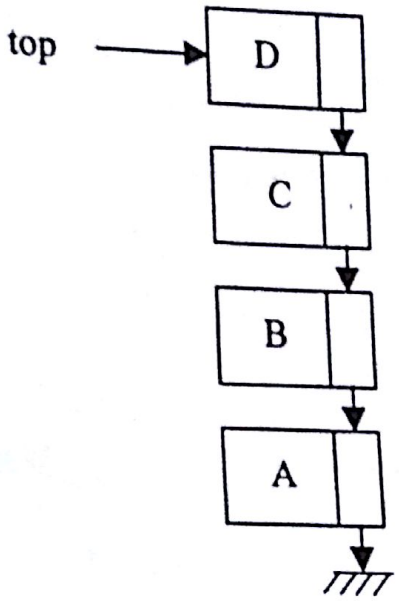


*Figure 4.5. Linked list representation of a stack after 'A', 'B', 'C', 'D' have been pushed onto it in that order.*

A stack can be defined as the following structure.

```
typedef struct stackElements {
        dataType item;
        struct stackElements *next;
}stackElements;
typedef struct stack {
        stackElements *top;
}stack;
```

If "s" is a variable of type "stack", then "s.top" points to the top of the stack "s". Push and pop operations on such a stack will typically involve manipulation of this "top" pointer. The functions for "push" and "pop" operations are described next.

```
void push (dataType a, stack *s)
{
        stackElements *x;
        x = (stackElements *) malloc (sizeof(stackElements));
        x->item = a;
        x->next = s->top;
        s->top = x;
}
int pop (dataType *a, stack *s)
{
        stackElements *t;
        if(s->top == NULL)
        {
                printf("Queue is empty \n");
                a = NULL;
                return 0;
        }
        *a = s->top->item;
        t = s->top;
        s->top = s->top->next;
        free(t);
        return 1;
}
```

Note that there is no question of stack overflow in this case.

A queue can be similarly represented by a singly connected linked list of queue elements. In addition to the list, two more pointer variables "front" and "rear" are also required to represent a queue. "front" and "rear" point to the elements at two extreme ends of the queue as shown in Figure 4.6.

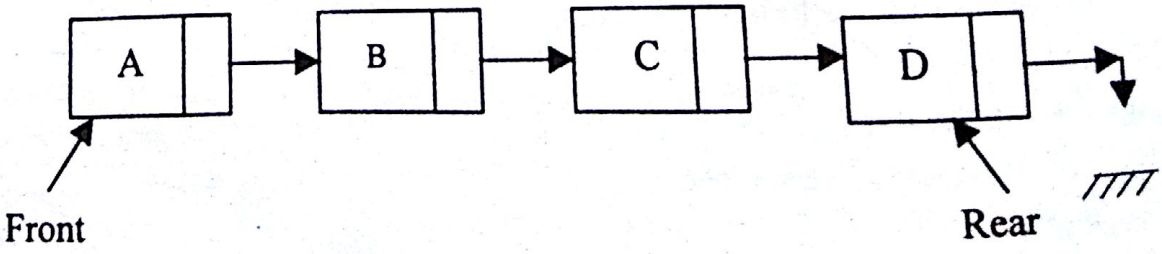


*Figure 4.6. Linked list representation of a queue.*

When a queue is initialized, both its pointers "front" and "rear" should be set to

NULL. The declaration of a queue using linked list is as follows.

```
typedef struct queueElements {
        dataType item;
        struct queueElements *next;
}queueElements;
typedef struct queue {
        queueElements *front;
        queueElements *rear;
}queue;
void addq(queue *q, dataType x)
{
        queueElements *t;
        t = (queueElements *) malloc (sizeof(queueElements));
        t->item = x;
        t->next = NULL;
        if(q->front == NULL)
        {
                q->front = t;
                q->rear = t;
                return;
        }
        q->rear->next = t;
        q->rear = t;
}
```

Note that special care has to be taken for insertion into an empty queue. In this case, "front" pointer has to be initialized to point to this first item in the queue. At the same time, "rear" pointer has to be updated also to point to the newly added item. Similar care is also necessary for deletion from a queue if the queue contains exactly one element, which is being pointed to simultaneously by "front" and "rear" pointers. The function for deletion operation is presented in the following.

```
int deleteq(queue *q, dataType *a)
{
        queueElements *t;
        if(q->front == NULL)
        {
                printf("Queue is empty \n");
```

```
            a = NULL;
            return 0;
        }
        t = q->front;
        q->front = q->front->next;
        *a = t->item;
        if(q->rear == t)
            q->rear = NULL;
        free(t);
        return 1;
}
```

## 4.4 Application of Stacks

There are numerous applications of stacks in computer algorithms. Some important uses of stacks will also be discussed in subsequent chapters. To begin with, consider the problem of checking whether a string of opening and closing parentheses is a well formed expression of parentheses or not. For example, the following strings are well formed.

(())(()())

((()))

(()(())()()()((()))()))

But the following strings are not well formed.

(()

)()(

(())(

A very simple algorithm based on stack can be formulated for solving the above problem. The algorithm is as follows. Scan the string from left to right. Whenever an opening parenthesis is found, push it into the stack and whenever a closing parenthesis is found, pop the stack once. If the stack is found empty when a pop operation is performed on the stack then it must be an error as it means that one closing parenthesis does not have a corresponding opening parenthesis. Once the string is exhausted, the stack must become empty as well because otherwise, there must be unmatched opening parenthesis. The algorithm is presented next. Input to the algorithm is a string of '(' and ')'. The output is +1 if the input is well formed; otherwise, the output is –1.

Algorithm checkParenthesis

Get the next character of the input string and assign it to x;

assign FALSE to error;

```
while((the input is not exhausted) && (not error))
{
        switch(x)
        {
                case '(':
                        push x into the stack;
                        break;
                case ')':
                        if the popped element is NULL
                                error = TRUE;
                        break;
        }
        Get the next character of the input string and assign it to x;
}
if(not error)
        if(stack is empty)
                return 1;
        else
                return -1;
return -1;
```

## 4.4.1 Evaluation of arithmetic expression

An arithmetic expression consists of operands and operators. The basic set of operations include addition, subtraction, multiplication, division and exponentiation. The symbols of the operators representing these operations are +, -, *, / and ^ respectively. The operands are either numeric constants or numeric variables. In the ensuing discussion, only numeric constants are taken into consideration as operands. For example,

$$9 + 5*7 - 6\hat{}2 + 15/3 \quad \ldots\ldots \quad (1)$$

is a valid arithmetic expression. In order to evaluate an expression, standard precedence rules for arithmetic expression are applied. For example, the exponentiation operation has the highest priority or precedence. This means that the exponentiation operations are carried out first in an arithmetic expression. So, in order to evaluate the expression (1), 6^2 is to be computed first. Hence, the resultant expression becomes

$$9 + 5*7 - 36 + 15/3 \quad \ldots\ldots \quad (2)$$

The priorities of multiplication and division operations are the same and this priority is less than that of exponentiation but more than those of addition or subtrac-

tion. In the expression (2), there is a multiplication and a division operation. These two operations are to be carried out next. Consequently the expression becomes

$$9 + 35 - 36 + 5 \quad \ldots\ldots (3)$$

The priorities of addition and subtraction operations are the same and therefore, these operations are carried out at the end. Hence, the result after computation of the above expression is 13. Default priority of an operation can be changed by enclosing the operation within parenthesis. The expression 1 + 2*3 will normally evaluate to 1 + 6 i.e. 7. But (1 + 2) *3 will evaluate to 3*3 i.e. 9. By enclosing the addition operation within parenthesis, priority of the operation becomes highest and hence, the operation is computed first. In general, if a sub-expression is enclosed within opening and closing parenthesis, the priority of the sub-expression becomes highest. In the following table, the priorities of different arithmetic operators are specified.

| Operator | Priority |
|---|---|
| ( ) | 4 |
| ^ | 3 |
| *, / | 2 |
| +, - | 1 |

In all expressions mentioned so far, the binary operators always appear between its operands. For example, if an expression is 2 + 3, then 2 and 3 are the operands of the operator +. Such expressions are known as infix expressions. Note that, in an infix expression, the unary operator always precedes its operand. As it has been already discussed, evaluation of an infix expression by a computer is not straight forward because the priorities of the operators involved are not explicit in such expressions. Only when an infix expression is fully parenthesized, the priorities of the operations are explicitly revealed. For example, consider the following expression.

$$4 + 2 * 3 \wedge 2 + 9/3 - 1*8$$

The sequence of evaluation of the individual operations in the given arithmetic expression can be obtained only after applying rules relating to the priorities and precedences of the arithmetic operators involved in the expression. Applying these rules, the sequence of evaluation of the above expression can be explicitly stated as shown in the following.

$$(((4 + (2 * (3 \wedge 2))) + (9 / 3)) - (1*8))$$

Another form of expression known as postfix expression is more amenable to mechanical evaluation. In a postfix expression the operators always appear after its operands. So, the infix expression "3 +2" becomes "3, 2 +" in its postfix form. The postfix form of "3 + 5 * 6 – 1" is "3, 5, 6, *, +, 1, -". If a postfix expression is to be evaluated, the expression is scanned from left to right. As soon as an operator is encountered, the last two operands are used to perform the operation and the

result is remembered. If the expression is "3, 5, 6, *, +, 1, -" then 3, 5, 6 are scanned and the operator '*' is encountered. The last two operands which are 5 and 6 are fetched and multiplication is performed and the result, 30, is stored. So the resultant expression becomes "3, 30, +, 1, -". Continuing the same procedure, 3 and 30 are scanned and the '+' operator is encountered. The result, 33 is stored. Consequently the expression becomes "33, 1, -". In the final scan, 33 and 1 become the operands for the subtraction operation and the result becomes 32. Although it may appear that repeated scans from the beginning of the expression are required, use of a stack may eliminate that need. Suppose the elements of the expression are scanned from left to right. If the element is a numeric value then it is pushed into a stack of integers. As soon as an operator is encountered the required number of integers are popped out from the stack, the operation is performed and the result is pushed back to the stack. Then, once again, elements are scanned from the remaining expression and the same process is continued. The following figure illustrates this procedure by showing the content of the stack and the remaining expression. Let the postfix expression be "3, 5, 6, *, +, 1, -". Initially the stack is empty as shown below.

| Stack content | Remaining expression |
|---|---|
| NIL | 3, 5, 6, *, +, 1, - |

Next, 3 is scanned and pushed into the stack.

| Stack content | Remaining expression |
|---|---|
| 3 | 5, 6, *, +, 1, - |

In the same way, 5, 6 are scanned and pushed into the stack.

| Stack content | Remaining expression |
|---|---|
| 3, 5, 6 | *, +, 1, - |

Now, the operator '*' is scanned. As '*' is a binary operator, the stack is popped twice to fetch its operands. So, 5, 6 are popped and then they are multiplied and the result, 30 is pushed into the stack.

| Stack content | Remaining expression |
|---|---|
| 3, 30 | +, 1, - |

The next element being scanned is '+'. As before, 30 and 3 are popped out and added. The result 33 is pushed back to stack. Condition of stack after the operation is shown in the following.

| Stack content | Remaining expression |
|---|---|
| 33 | 1, - |

Next, '1' is scanned and pushed into the stack resulting in the configuration as shown in the following.

| Stack content | Remaining expression |
|---|---|
| 33, 1 | - |

- The current element being scanned is '-'. 1 and 33 are taken out and subtraction operation is performed. The result 32 is pushed into the stack. Now the expression is exhausted. Therefore, the item popped up from the stack is the result of the evaluation of the expression. The condition of stack is shown in the following.

| Stack content | Remaining expression |
|---|---|
| 32 | NIL |

The algorithm for the postfix expression evaluation is described next. The input to the algorithm is an expression having operands and operators. The output is the result after evaluating the expression. The last element of the expression is denoted by a special symbol, say a '$'.

Algorithm evaluatePostfixExpression

Initialize a stack of real numbers.

Get the next element from the expression.

while the element obtained from the expression is not '$'

{

if the element represents a numeric value

push the numeric value to the stack.

else if the element represents an operator

{