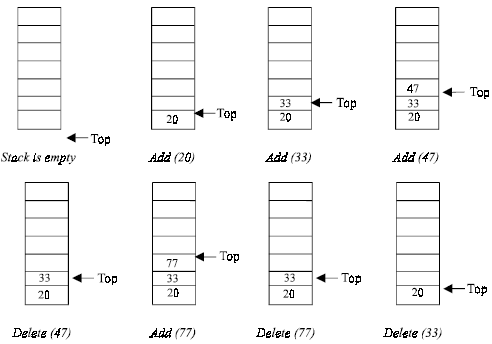
**Chapter Five**

**Stack and Queues**

A **stack** is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added /inserted and from which items may be deleted at only one end, called the *top* of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called ***Last-in-First-out (LIFO)***. Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.

The operation of the stack can be illustrated as in Fig.



**Fig.** *Stack operation****.***

The insertion (or addition) operation is referred to as *push*, and the deletion (or remove) operation as *pop*. A stack is said to be *empty* or *underflow*, if the stack contains no elements. At this point the top of the stack is present at the bottom of the stack. And it is *overflow* when the stack becomes full, *i.e.*, no other elements can be pushed onto the stack. At this point the top pointer is at the highest location of the stack.

**Operations performed on stack**

The primitive operations performed on the stack are as follows:

***Push:***The process of adding (or inserting) a new element to the top of the stack is called Push operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

***Pop****:* The process of deleting (or removing) an element from the top of stack is called Pop operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

**Stack Implementation**

Stack can be implemented in two ways:

1. Static implementation (using arrays)

2. Dynamic implementation (using pointers)

Static implementation uses arrays to create stack. Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (*i.e.,* increased or decreased). Moreover static implementation is not an efficient method when resource optimization is concerned (*i.e.,* memory utilization). For example a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be wasted, on the other hand if there are more number of elements to be stored in the stack (say 60) then we cannot change the size array to increase its capacity.

The above said limitations can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.

**Stack Using Arrays**

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stack is given below.

**Algorithm for push**

Suppose STACK [SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

1. If TOP = SIZE – 1, then:

(*a*) Display “The stack is in overflow condition”

(*b*) Exit

2. TOP = TOP + 1

3. STACK [TOP] = ITEM

4. Exit

**Algorithm for pop**

Suppose STACK [SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

1. If TOP < 0, then

(*a*) Display “The Stack is empty”

(*b*) Exit

2. Else remove the Top most element

3. DATA = STACK[TOP]

4. TOP = TOP – 1

5. Exit

**Application of Stack** (Reading Assignment)

**Expression**

Another application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; an expression is defined as the number of operands or data items combined with several operators.)

1. Infix notation

2. Prefix notation

3. Postfix notation

The *infix notation* is what we come across in our general mathematics, where the operator is written in-between the operands. For example: The expression to add two numbers A and B is written in infix notation as:

A + B

Note that the operator ‘+’ is written in between the operands A and B.

The *prefix notation* is a notation in which the operator is written before the operands, it is also called polish notation. The same expression when written in prefix notation looks like:

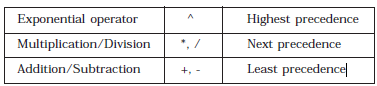
+ A B

As the operator ‘+’ is written before the operands A and B, this notation is called prefix (pre means before).

In the *postfix notation* the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as *suffix notation* or *reverse polish notation*. The above expression if written in postfix expression looks like:

A B +

***Operator Precedence***



**Converting Infix to Postfix Expression**

The method of converting infix expression A + B \* C to postfix form is:

A + B \* C Infix Form

A + (B \* C) Parenthesized expression

A + (B C \*) Convert the multiplication

A (B C \*) + Convert the addition

A B C \* + Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.

2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression

B \* C is parenthesized first before A + B.

3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.

4. Once the expression is converted to postfix form, remove the parenthesis.

**Problem1** Give postfix form **for A + [(B + C) + (D + E) \* F] / G**

**Solution**

A + { [ (BC +) + (DE +) \* F ] / G}

A + {[(BC +) + (DE + F \*] / G}

A + {[(BC + (DE + F \* +] / G}

A + [BC + DE + F \*+ G /]

ABC + DE + F \* + G / + Postfix Form

**Problem** Give postfix form for **(A + B) \* C / D + E ^ A / B**

**Solution**

[(AB +) \* C / D] + [(EA ^) / B]

[(AB +) \* C / D] + [(EA ^) B /]

[(AB +) C \* D /] + [(EA ^) B /]

(AB +) C \* D / (EA ^) B / +

AB + C \* D / EA ^ B / + Postfix Form

**Algorithm**

Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators, P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential ( ^ ), multiplication ( \* ), division ( / ), addition ( + ) and subtraction ( - ).

The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. the algorithm is completed when the stack is empty.

1. Push “(” onto stack, and add“)” to the end of P.

2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.

3. If an operand is encountered, add it to Q.

4. If a left parenthesis is encountered, push it onto stack.

5. If an operator ⊗is encountered, then:

(*a*) Repeatedly pop from stack and add P each operator (on the top of stack), which has the same precedence as, or higher precedence than ⊗.

(*b*) Add ⊗to stack.

6. If a right parenthesis is encountered, then:

(*a*) Repeatedly pop from stack and add to P (on the top of stack until a left parenthesis is encountered.

(*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]

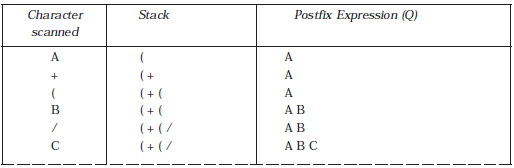
7. Exit.

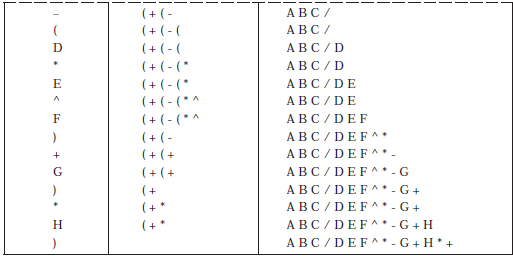
**Note.** Special character ⊗is used to symbolize any operator in P.

Consider the following arithmetic infix expression P

P = A + (B / C - (D \* E ^ F) + G) \* H

Fig. shows the character (operator, operand or parenthesis) scanned, status of the stack and postfix expression Q of the infix expression P.





**Evaluating Postfix Expression**

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

**Algorithm**

1. Add a right parenthesis “)” at the end of P. [This acts as a sentinel.]

2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel “)” is encountered.

3. If an operand is encountered, put it on STACK.

4. If an operator ⊗ is encountered, then:

(*a*) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.

(*b*) Evaluate B ⊗ A.

(*c*) Place the result on to the STACK.

5. Result equal to the top element on STACK.

6. Exit.

**Queues**

A queue is logically a *first in first out (FIFO or first come first serve)* linear data structure. The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service centre.

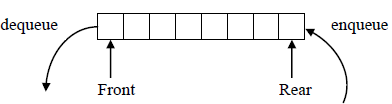
It is a homogeneous collection of elements in which new elements are added at one end called *rear*, and the existing elements are deleted from other end called *front*.

The basic operations that can be performed on queue are

1. Insert (or add) an element to the queue (enqueue).

2. Delete (or remove) an element from a queue (dequeue)

Push operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. Pop operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Following figure will illustrate the basic operations on queue.



* If a queue is empty, then front=rear=-1
* If a queue have elements and if no element is dequeued (removed) then front =0
* In each dequeue operation front will be incremented by one

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Content of queue | Front | Rear |
|  | Empty | -1 | -1 |
| Enqueue(B) | B | 0 | 0 |
| Enqueue(C) | B, C | 0 | 1 |
| Dequeue() | C | 1 | 1 |
| Enqueue(G) | C, G | 1 | 2 |
| Enqueue (F) | C, G, F | 1 | 3 |
| Dequeue() | G, F | 2 | 3 |
| Enqueue(A) | G, F, A | 2 | 4 |
| Dequeue() | F, A | 3 | 4 |

**Queue Implementation**

Queue can be implemented in two ways:

* Using array (statically)
* Using linked list (dynamically)

**Array implementation of Queue**

**Algorithm for Queue Operations**

Let Q be the array of some specified size say SIZE

**Inserting an Element into the Queue**

1. Initialize front=0 rear = –1

2. Input the value to be inserted and assign to variable “data”

3. If (rear >= SIZE)

(*a*) Display “Queue overflow”

(*b*) Exit

4. Else

(*a*) Rear = rear +1

5. Q[rear] = data

6. Exit

**Deleting an Element from Queue**

1. If (rear< front)

(*a*) Front = 0, rear = –1

(*b*) Display “The queue is empty”

(*c*) Exit

2. Else

(*a*) Data = Q[front]

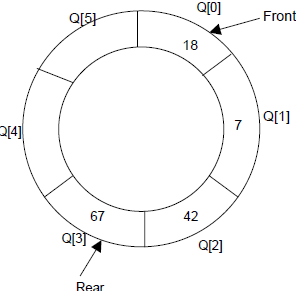
3. Front = front +1

4. Exit

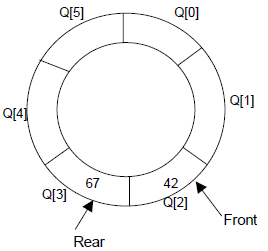
**CIRCULAR QUEUE**

In circular queues the elements Q[0],Q[1],Q[2] .... Q[*n* – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.

Suppose Q is a queue array of 6 elements. Push and pop operation can be performed on circular. The following figures will illustrate the same.

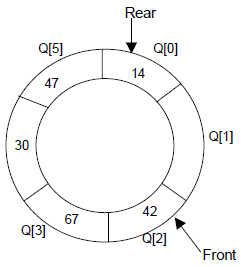


A circular queue after inserting 18, 7, 42, 67



A circular queue after popping 18, 7

After inserting an element at last location Q[5], the next element will be inserted at the very first location (*i.e.*, Q[0]) that is circular queue is one in which the first element comes just after the last element.



At any time the position of the element to be inserted will be calculated by the relation Rear = (Rear + 1) % SIZE

After deleting an element from circular queue the position of the front end is calculated by the relation Front= (Front + 1) % SIZE

After locating the position of the new element to be inserted, *rear*, compare it with *front*. If (rear = front), the queue is full and cannot be inserted anymore.

**Algorithms**

Let Q be the array of some specified size say SIZE. FRONT and REAR are two pointers where the elements are deleted and inserted at two ends of the circular queue. DATA is the element to be inserted.

**Inserting an element to circular Queue**

1. Initialize FRONT = – 1; REAR = 1

2. REAR = (REAR + 1) % SIZE

3. If (FRONT is equal to REAR)

(*a*) Display “Queue is full”

(*b*) Exit

4. Else

(*a*) Input the value to be inserted and assign to variable “DATA”

5. If (FRONT is equal to – 1)

(*a*) FRONT = 0

(*b*) REAR = 0

6. Q[REAR] = DATA

7. Repeat steps 2 to 5 if we want to insert more elements

8. Exit

**Deleting an element from a circular queue**

1. If (FRONT is equal to – 1)

(*a*) Display “Queue is empty”

(*b*) Exit

2. Else

(*a*) DATA = Q[FRONT]

3. If (REAR is equal to FRONT)

(*a*) FRONT = –1

(*b*) REAR = –1

4. Else

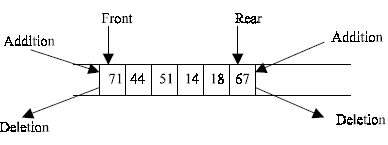
(*a*) FRONT = (FRONT +1) % SIZE

5. Repeat the steps 1, 2 and 3 if we want to delete more elements

6. Exit

**DEQUES (Double Ended Queues)**

A deque is a homogeneous list in which elements can be added or inserted (called push operation) and deleted or removed from both the ends (which is called pop operation). i.e.; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.



There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are

1. Input restricted deque

2. Output restricted deque

An input restricted deque is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both end, rear and front end of the lists.

An output-restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

The possible operation performed on deque is

1. Add an element at the rear end

2. Add an element at the front end

3. Delete an element from the front end

4. Delete an element from the rear end

Only 1st, 3rd and 4th operations are performed by input-restricted deque and 1st, 2nd and 3rd operations are performed by output-restricted deque.

**Algorithms for Inserting an Element**

Let Q be the array of MAX elements. *front* (or left) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. Let DATA be the element to be inserted. Before inserting any element to the queue *left* and *right* pointer will point to the – 1.

**Insert an Element at the Right Side of the De-Queue**

1. Input the DATA to be inserted

2. If ((left == 0 && right == MAX–1) || (left == right + 1))

(*a*) Display “Queue Overflow”

(*b*) Exit

3. If (left == –1)

(*a*) left = 0

(*b*) right = 0

4. Else

(*a*) if (right == MAX –1)

(*i*) left = 0

(*b*) else

(*i*) right = right+1

5. Q[right] = DATA

6. Exit

**Insert an Element at the Left Side of the De-Queue**

1. Input the DATA to be inserted

2. If ((left == 0 && right == MAX–1) || (left == right+1))

(*a*) Display “Queue Overflow”

(*b*) Exit

3. If (left == – 1)

(*a*) Left = 0

(*b*) Right = 0

4. Else

(*a*) if (left == 0)

(*i*) left = MAX – 1

(*b*) else

(*i*) left = left – 1

5. Q[left] = DATA

6. Exit

**Algorithms for Deleting an Element**

Let Q be the array of MAX elements. *front* (or *left*) and *rear* (or *right*) are two array index (pointers), where the addition and deletion of elements occurred. DATA will contain the element just deleted.

**Delete an Element from the Right Side of the De-Queue**

1. If (left == – 1)

(*a*) Display “Queue Underflow”

(*b*) Exit

2. DATA = Q [right]

3. If (left == right)

(*a*) left = – 1

(*b*) right = – 1

4. Else

(*a*) if(right == 0)

(*i*) right = MAX-1

(*b*) else

(*i*) right = right-1

5. Exit

**Delete an Element from the Left Side of the De-Queue**

1. If (left == – 1)

(*a*) Display “Queue Underflow”

(*b*) Exit

2. DATA = Q [left]

3. If(left == right)

(*a*) left = – 1

(*b*) right = – 1

4. Else

(*a*) if (left == MAX-1)

(*i*) left = 0

(*b*) Else

(*i*) left = left +1

5. Exit