Unit 3 Stacks and Queues

Structure:

3.1 Introduction

In the previous unit, we have learnt about the linear data structure linked list. In linked list the insertion and deletion can be performed at any place in the list at the beginning, at the end, or in the middle.

In some situations one wants to restrict the insertion and deletion at the beginning or the end of the list, not in the middle. Two of the data structures used in these situations are stack and queue. In this unit we will discuss about the various implementation of stacks and queues and their applications.

Objectives:

After studying this unit, you should be able to:

- discuss the array implementation of stack
- describe the linked list implementation of stack
- list some of the applications of stack
- describe the array and linked list of queue

3.2 Stack

A stack is a data structures in which insertion and deletion of items are made at the one end, called the *top* of the stack.

We have two basic operations in stack they are *push* and *pop.*

Push Operation: Push is used to insert an item into a stack.

Pop Operation: Pop is used to delete an item from a stack.

The figure 3.1 shows the stack with five items and the top pointer were the insertion and deletion are performed. From the figure 3.1 you can see that 5 is the current top item. If any new items are added, they are placed on the top of 5 and if any item are to be deleted, then 5 is the first element to be deleted. This means that the last item entered or inserted is the first one to be removed or deleted. So, stacks are also called as last- in first- out (LIFO).

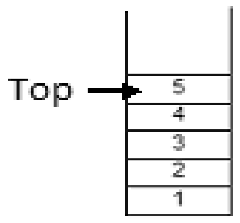


Figure 3.1: Stack

The figure 3.2 shows how the stack expands and shrinks. 3.2 (a) is the actual stack with five elements. Now item 6 is inserted, according to the definition the one position where the item 6 can be inserted is top and now 6 will be the top item. Similarly the item 7 is also inserted and now it is the top item that you can see in figure 3.2 (b) and (c). In figure 3.2(d) you can see when an item has to be removed, then according to the definition the top item i.e. 7 is removed first.

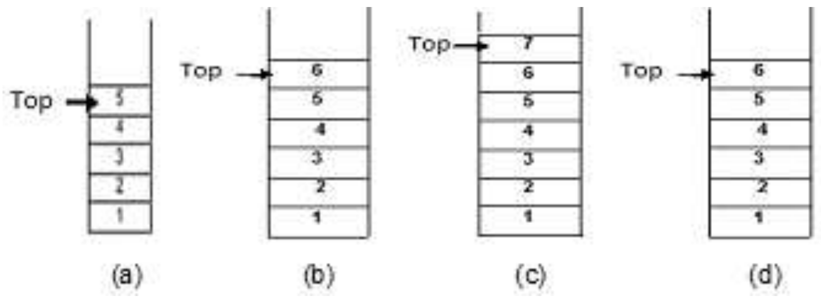


Figure 3.2: Stack- expands and shrinks

3.2.1 Array implementation of stack

Implementation of stack can be done either using array or one way list. The main disadvantage of array implementation of stack is that the size of the array has to be declared a head. So, the stack size is fixed and dynamically you cannot change the size of stack.

Each stack maintains a array STACK, a pointer variable TOP which contains the location of top element and a variable MAXSTK which gives the maximum number of elements that can be held by the stack.

The figure 3.3 shows the array representation of stack. Since TOP = 3, the stack has three elements and since MAXSTK =7, more 4 elements can be added to stack.

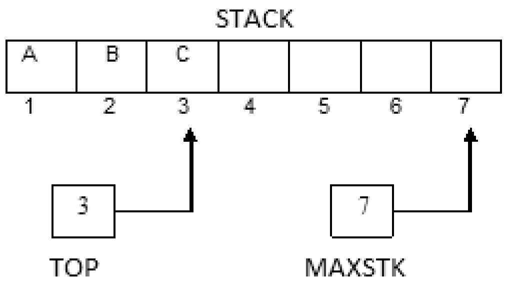


Figure 3.3: Array Representation of Stack

The following are the operations to add an item into stack and remove an item from a stack.

Push operation

The push operation is used to add an item into a stack. Before executing push operation one must check for the OVERFLOW condition, i.e. check whether there is room for the new item in the stack. The following procedure performs the PUSH operation.

```
Procedure: PUSH (STACK, TOP, MAXSTK, ITEM)
    1. [Stack already full?]
       If TOP = MAXSTK, then: print: OVERFLOW, and Return.
    2. Set TOP := TOP + 1. [Increases TOP by 1]
    3. Set STACK [TOP]:= ITEM. [Insert ITEM in new TOP position]
    4. Return.
```

As per the procedure first it checks for the OVERFLOW condition. Since the stack is not full the TOP pointer is incremented by 1, TOP= TOP + 1. So, now TOP points to a new location and then the ITEM is inserted into that position.

Pop operation

The pop operation is used to remove an item from a stack. Before executing the pop operation one must check for the UNDERFLOW condition, i.e. check whether stack has an item to be removed. The following procedure performs the POP operation.

```
Procedure: POP (STACK, TOP, ITEM)
  1. [Stack is empty?]
     If TOP = 0, then: Print: UNDERFLOW, and Return.
  2. Set ITEM := STACK [TOP] . [Assign Top element to ITEM]
  3. Set TOP := TOP – 1. [Decreases Top by 1]
  4. Return.
```

As per the procedure, first it checks for the underflow condition. Since the stack is not empty the top element in the stack is assigned to ITEM, ITEM: = STACK [TOP]. Then the top is decremented by 1.

3.2.2 Linked list implementation of stack

Let us see how stack is implemented using linked list. As we have seen in previous section the advantages of linked list over array is that, it is not necessary to declare the size of linked list a head and the size can be changed dynamically. So, at any point we can expand or shrink the stack size. One more advantage is that cost of insertion and deletion is less compared to array implementation.

Linked list implementation of stack uses singly linked list or one- way list where the DATA field contains the ITEM to be stored in stack and the link field contains the pointer to the next element in the stack. Here TOP points to the first node of linked list which contains the last entered element and null pointer of the last node indicates the bottom of stack. The figure 3.4 shows the linked list representation of stack.

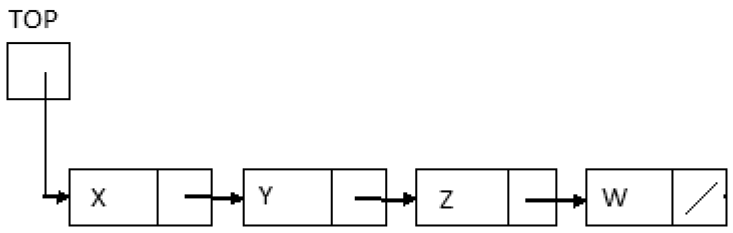


Figure 3.4: Linked List Implementation of stack

The following are the operations to add an item into stack and remove an item from a stack.

Push operation

Push operation is performed by inserting a node into the front or start of the list. Even though in this we don't want to declare the size of linked list a head we have to check for the OVERFLOW condition of the free- storage list. The following procedure performs the push operation.

```
Procedure: PUSH (DATA, LINK, TOP, VAIL, ITEM)
   1. [Available space?]
      If AVAIL = NULL, then Write OVERFLOW and Exit
   2. [Remove first node from AVAIL list]
      Set NEW := AVAIL and AVAIL := LINK [AVAIL].
   3. Set DATA [NEW] := ITEM [Copies ITEM into new node.]
   4. Set LINK [NEW] := TOP [New node points to the original top
      node in the stack]
   5. Set TOP = NEW [Reset TOP to point to the new node at the
      top of the stack]
   6. Exit.
```

First check the available space in the free storage list. If free space is available then make the pointer variable NEW to point the first node of AVAIL list, NEW: = AVAIL. Get the data for the new node, DATA [NEW]:= ITEM and make the link filed of new node to point the actual top node or first node in the list, LINK [NEW]:= TOP. Now new node becomes the first node in the list and the TOP pointer point to the NEW node. The figure 3.5 shows the PUSH operation in linked list implementation of stack.

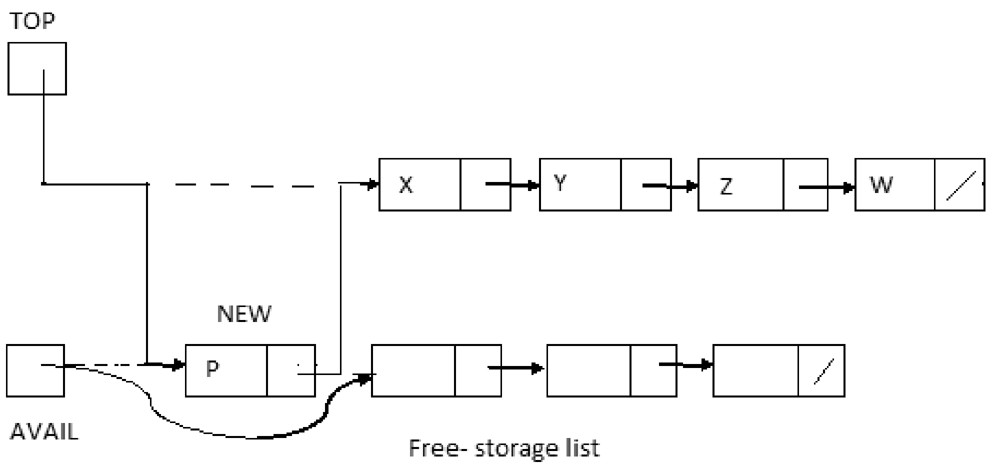


Figure 3.5: Push Operation

Pop operation

Pop operation is performed by removing the first or the start node of a linked list. Before executing the pop operation one must check for the UNDERFLOW condition, i.e. check whether stack has an item to be removed. The following procedure performs the pop operation.

```
Procedure: POP (DATA, LINK, TOP, AVAIL, ITEM)
    1. [Stack is empty?]
       If TOP = NULL then Write: UNDERFLOW and Exit.
    2. Set ITEM := DATA [TOP] [copies the top element of stack into
       ITEM]
    3. Set TEMP := TOP and TOP = LINK [TOP]
       [Remember the old value of the TOP pointer in TEMP and reset
       TOP to point to the next element in the stack.]
    4. [Return deleted node to the AVAL list]
       Set LINK [TEMP] = AVAIL and AVAIL = TEMP.
    5. Exit.
```

First check the UNDERFLOW condition. If stack is not empty then the element in the top node is copied to the ITEM, ITEM := DATA [TOP]. A temporary variable TEMP is made to point the top node, TEMP := TOP and the top pointer now points to the next node in the list, Top = LINK [TOP]. Now the node pointed by TEMP is moved to the AVAIL list, LINK [TEMP] =

AVAIL and AVAIL = TEMP. The figure 3.6 shows the POP operation in linked list implementation of stack.

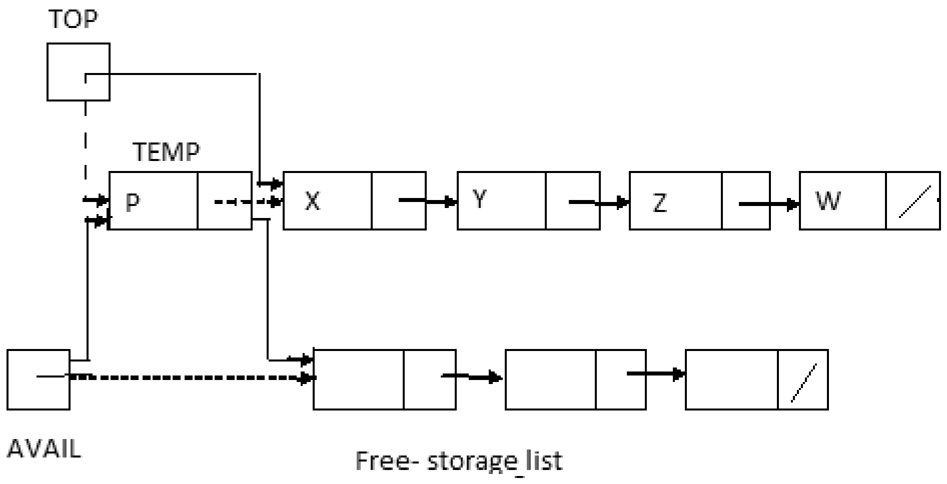


Figure 3.6: Pop Operation

Self Assessment Questions

1. Stack allows insertion and deletion at one end called _______________.
2. Insertion and deletion of element from the stack is performed with ______ and ________ operation.
3. Before every insertion into the stack __________ condition need to be checked.

3.3 Applications of Stack

Arithmetic Expression

Generally in all arithmetic expression the operators are placed in between the operands, this is called infix notation.

A+B and (X+Y)* Z

In some type of notation the operator is placed before its two operands, this is called prefix notation or polish notation.

+AB and *+XYZ

In another type of notation the operator is placed after its two operands, this is called postfix notation or reverse polish notation.

AB+ and XY+Z*

Now we are going to discuss the role of stack, during the process of arithmetic expressions are

- Evaluation of a postfix expression
- Infix to postfix conversion

3.3.1 Evaluation of a postfix expression

Suppose we have an arithmetic expression written in postfix notation. By using STACK we are going to evaluate the expression. The following algorithm, which uses a stack to hold operands, evaluates the expression.

```
Algorithm:  Evaluation of Postfix Expression.
    1. Add a right parenthesis ")" at the end of X.
       [This acts as a sentinel]
    2. Scan X from left to right and repeat Steps 3 and 4 for each
       element of X 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 element of STACK,
                    where A is the top element and B is the
                    next- to-top element.
                (b) Evaluate B A.
                (c) Place the result of (b) back on STACK.
            [End of If structure]
       [End of Step 2 loop.]
    5. Set VALUE equal to the top element on STACK.
    6. Exit.
```

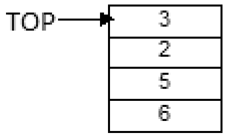
Let see an example for this. Consider X to be an arithmetic expression written in postfix notation. X = 6 5 2 3 + 8 * + 3 + *.

As per the algorithm, first we need to add the right parenthesis ")" at the end of X.

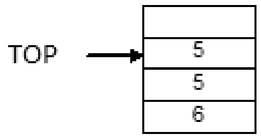
X= 6 5 2 3 + 8 * + 3 + *)

Then we have start the scan from left to right until we get the sentinel ")".

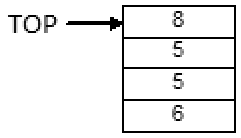
So the first four symbols are operands so they are place on the stack.



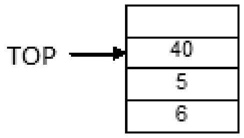
Next to that we have an operator '+', so we pop the top 2 elements i.e. 3 and 2 and their sum, 5 is pushed back into stack.



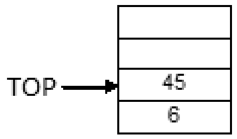
Next 8 is pushed into stack.



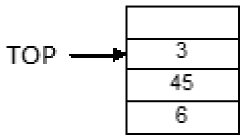
Next is '*', so 8 and 5 are popped and 5*8 = 40 is pushed.



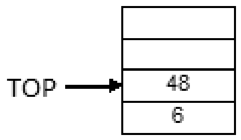
Next is '+', so 40 and 5 are popped and 5+40 = 45 is pushed.



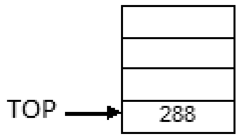
Next 3 is pushed into stack.



Next is '+', so 3 and 45 are popped and 45+3 = 48 is pushed.



Next is '*', so 48 and 6 are popped and 6*48 = 288 is pushed.



Next is ')', so it terminates the loop and the VALUE= 288.

3.3.2 Infix to postfix conversion

Generally the expression we use for algebraic notations are in infix form, now we are going to discuss how to convert this infix to postfix, the following algorithm helps us to proceed with. Infix is the input string and the out put what we get after the process is the postfix string.

Algorithm for conversion of infix to postfix

1. Initialize a STACK as empty in the beginning.
2. Inspect the infix string from left to right.

   While reading character from a string we may encounter
   Operand - add with the postfix string.

   Operator – if the stack is empty push the operator into stack, if any operator available with the stack compares the current operator precedence with the topStack if it has higher precedence over the current one pop the stack and add with the post string else push the current operator to the stack. Repeat this process until the stack is empty.

   Left Parenthesis: Push in to the STACK.

   Right Parenthesis: Pop everything until you get the left parenthesis or end of STACK.
3. Repeat Process with the infix string until all the characters are read.
4. Check for stack status if it is not empty add topStack to postfix string and repeat this process until the STACK is empty
5. Return the postfix string.
6. Exit.

Note : **Infix precedence**

- Parenthesis ()
- Exponentiation ^
- Multiplication *, Division /
- Addition +, Subtraction –

Example for infix to postfix expression

a + b * c - d

Infix String	Stack status	Postfix string
a + b * c - d	Null	Empty
+ b * c – d	Null	a
b * c - d	+	a
* c - d	+	ab
c - d	* +	ab
- d	* +	abc
d	-	abc*+
	-	abc*+d
		abc*+d-

3.4 Queue

A queue is a linear list of elements in which deletions can take place only at one end, called the *front* and insertions can take place only at the other end, called the *rear* as referred in the figure 3.7. The terms “front” and “rear” are used in describing a linear list only when it is implemented as a queue. Following are the two methods offered by queue for adding and deleting element from the queue.

- *enqueue* - add a new item at the back of the queue
- *dequeue* - remove the item at the front of the queue

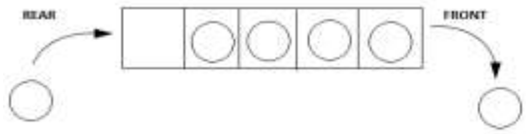


Figure 3.7: Queue representation

Queues are also called (FIFO) first-in-first-out, since the first element in a queue inserted will be deleted first. Queue can be implemented in the following two ways.

- Array implementation of queue
- Linked list implementation of queue

3.4.1 Array implementation of queue

Since a queue usually holds a bunch of items with the same type, we could implement by means of one way list or linear array. Here the queue in a linear way is maintained with two pointers called FRONT, containing the location of the front element of the queue and REAR, to hold the location which is at rear. Insertion and deletion of element is not handled as the normal array where we shift the elements forward or backward. Here, whenever the element is deleted from the queue the value of the FRONT is increased by 1 this can be implemented by FRONT: =FRONT+1. Similarly whenever an element is added to the queue, the value of REAR is increased by 1 by assigning REAR: =REAR +1 as referred in the figure 3.8.

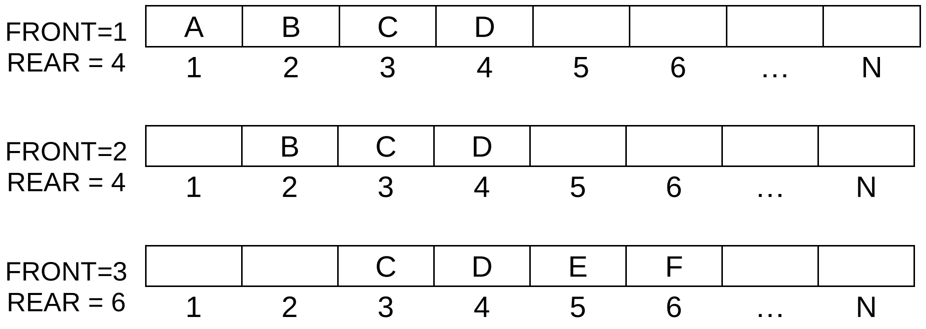


Figure 3.8: Array representation of queue

After N insertions, the rear element of the queue will occupy QUEUE[N] and this occurs even the array is not full. If we insert a new element when REAR =N, one way to handle this is to simply move the entire queue to the beginning of the array and change FRONT AND REAR accordingly an then insert an item. This procedure may be expensive, an alternate is consider the queue is circular that is QUEUE[1] comes after QUEUE[N] in the array. With this assumption we can insert ITEM into the queue by assigning ITEM to QUEUE[1], instead of increasing the REAR to N+1 we can reset REAR=1 then the assignment will be QUEUE[REAR]:=ITEM. Similarly if FRONT=N and an element of queue is deleted we can reset FRONT=1 instead of increasing FRONT to N+1.

Now we are going to discuss how to insert an element into the queue with the procedure QINSERT(), before insertion we need to check the overflow status of the queue and find the current position of REAR and increment that with one and this is the new location for inserting a new item .

```
QINSERT (QUEUE,N,FRONT,REAR,ITEM)
    1. If FRONT=1 and REAR =N, or FRONT=REAR+1 then
            Write OVERFLOW, and Return
    2. [Find new value of REAR]
        If FRONT :=NULL, then: [Queue is empty initially]
            Set FRONT :=1 and REAR:=1
        Else if REAR=N then:
        Set REAR :=1.
        Else
        Set REAR:=REAR+1
    3. Set QUEUE[REAR]:=ITEM
    4. Return
```

Now we will discuss how to delete an item from the queue with the procedure QDELETE() which checks for the underflow status if queue contains items it deletes an first element from the queue by assigning it to the variable ITEM and calculate new value for REAR.

```
QDELETE (QUEUE, N, FRONT, REAR, ITEM)
    1. If FRONT: =NULL, then write : UNDERFLOW and Return.
    2. Set ITEM:=QUEUE[FRONT].
    3. If FRONT = REAR, then:          // only one element.
            Set FRONT: =NULL and REAR: =NULL.
       Else if FRONT := N then:
            Set FRONT: =1.
       Else:
            Set FRONT: = FRONT +1.
    4. Return.
```

3.4.2 Linked implementation of queue

In this section we are going to discuss how to represent a queue in linked list. Queues are very much like linked list except the ability to manipulate items on the lists, a linked queue is a queue implemented as a linked list with two pointer variables FRONT and REAR pointing to the nodes which is in the FRONT and REAR of the queue as referred in figure 3.9.

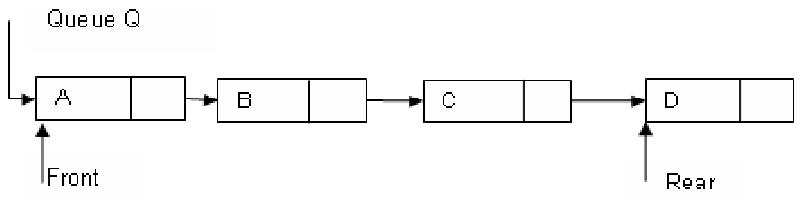


Figure 3.9: Linked representation of queue

Insertion of element in linked queue

The array representation of queue had a disadvantage of limited queue capacity that is, every time of insertion we need to check for OVERFLOW status and then insert a new element. But in linked queue representation while inserting an element a new node will be availed from the AVAIL list which holds the ITEM, will be inserted as the last node of the linked list representing queue. The rear point will be updated in order to point the node which is recently entered.

Figure 3.10 is illustrating how to insert a new node into the queue, new node is availed from the AVAIL list and ITEM D is assigned with the new node. Before insertion the value of the REAR was REAR = 3 and having NULL pointer. As we know insertion can happen only with REAR, the NEW node is linked with the LINK [REAR] and the Value of the REAR is incremented to 4.

```
LINKQ_INSERT (INFO,LINK,FRONT,REAR,AVAIL,ITEM)
  1. If AVAIL = NULL, then write OVERFLOW and Exit
  2. Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
       [Remove first node from AVAIL list]
  3. Set INFO[NEW] :=ITEM and LINK[NEW]=NULL
       [copies ITEM to new node]
  4. If (FRONT=NULL) then FRONT=REAR=NEW
       [If queue is empty then ITEM is the first element in the
       queue Q]
       Else Set LINK[REAR] :=NEW and REAR = NEW
        [REAR points to the new node appended to the end of list]
  5. Exit.
```

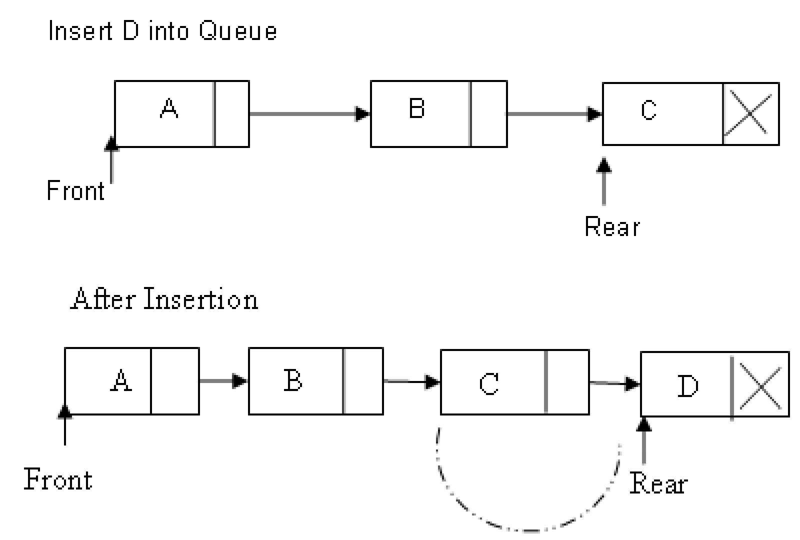


Figure 3.10: Insertion of node in linked queue

Deletion of element from linked queue

Now in this topic we are going to discuss how to delete an element from the linked queue, the deletion can happen only from the FRONT end. In case of deletion the first node of the list pointed to by FRONT is deleted and the FRONT pointer is updated to point to the next node in the list and the deleted node will be return to the AVAIL list. The figure 3.11 is showing the process of deletion, here the node which is in the FRONT carrying the value A is deleted and the FRONT pointer is updated to the next node carrying the value B.

```
LINKQ_DELETE (INFO, LINK, FRONT, REAR, AVAIL, ITEM)
    1. If (FRONT=NULL) then Write: UNDERFLOW and Exit
    2. Set TEMP=FRONT [if queue is not empty]
    3. ITEM=INFO(TEMP)
    4. FRONT=LINK(TEMP) [Reset FRONT to next element]
    5. LINK(TEMP)=AVAIL and AVAIL=TEMP
         [Return deleted node to AVAIL list]
    6. Exit.
```

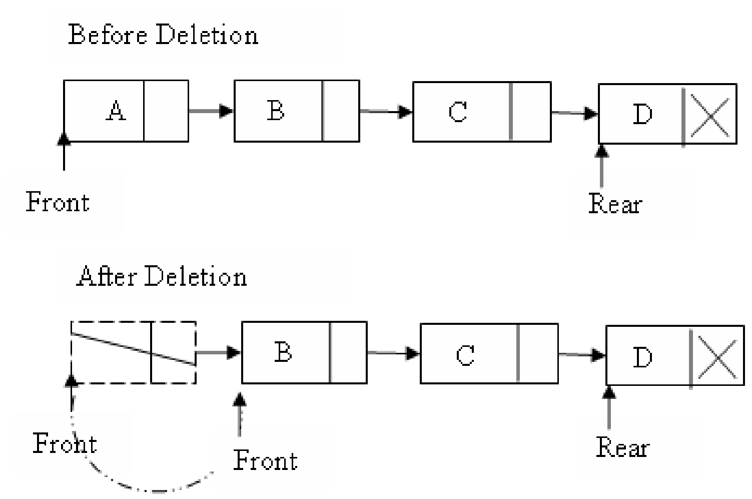


Figure 3.11: Deletion of node from linked queue

Self Assessment Questions

4. P+Q and (X+Y)* Z is the _______________ expression.
5. Addition is having higher precedence then Multiplication state. True/False.
6. Specify the method for deleting and element from the queue ________.
7. __________ and ___________ are the two pointers are used in queue.
8. New node can be availed from ___________ list while inserting new element into the queue.

3.5 Summary

This unit provided you the information about two special data structures that is Stack and Queue, which provides List -like capabilities in that they can store an arbitrary number of elements. The Queue and Stack differ from the List in the sense that while the List allows direct, random access to its elements, both the Queue and Stack limit how elements can be accessed. The Stack, offers LIFO access, which stands for last in, first out. Stacks provide this access scheme through its Push() and Pop() methods. Stacks are used in a number of areas in computer science, from code execution to parsing. The Queue on the other hand, uses a FIFO strategy, or first in, first out. That is, the order with which items can be removed from the Queue is precisely the order with which they were added to the Queue. To provide these semantics, the Queue offers two methods: Enqueue() and Dequeue(). Queues are useful data structures for job processing or other tasks where the order with which the items are processed is based by the order in which they were received. We have also discussed the representations and applications of stack and queues.

3.6 Terminal Questions

1. Discuss the stack data structure with Push() and Pop() operation.
2. Explain the array and linked list implementation of stack.
3. Explain the evaluation of postfix expression.
4. Discuss the process of Infix to postfix conversion.
5. Explain array implementation of queue.
6. Discuss the linked implementation of queue.

3.7 Answers

Self Assessment Questions

1. Top
2. Push and pop
3. OVERFLOW
4. Infix
5. False
6. Dequeue
7. FRONT and REAR
8. AVAIL

Terminal Questions

1. Stack is a data structure works with LIFO pattern. (Refer section 3.2 for detail)
2. Stack can be implemented two more different ways that is array and linked list. (Refer section 3.2 for detail)
3. Role of stack during evaluation of postfix expression. (Refer sub-section 3.3.1 for detail)
4. Role of stack in conversion of infix to postfix expression. (Refer sub-section 3.3.2 for detail)
5. Queue can be implemented using array. (Refer sub-section 3.4.1 for detail)
6. Queue can be implemented using linked list. (Refer sub-section 3.4.2 for detail)