Chapter 03
Stacks and Queues

01 Stack:

01.1 Definition:

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.

01.2 Operations:

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.

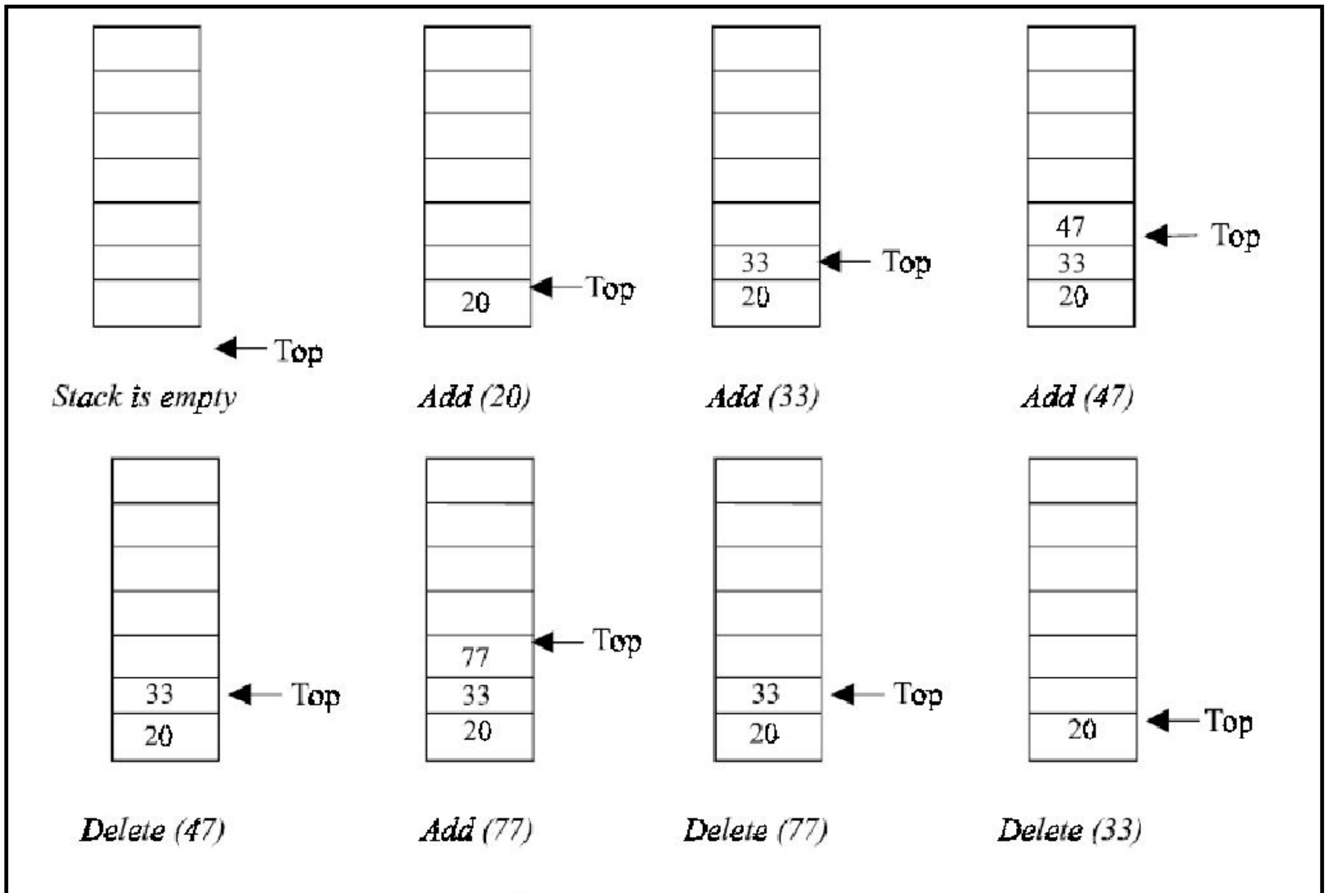


Fig. 3.1. Stack operation.

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.

01.3 Array representation of stack:

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full.

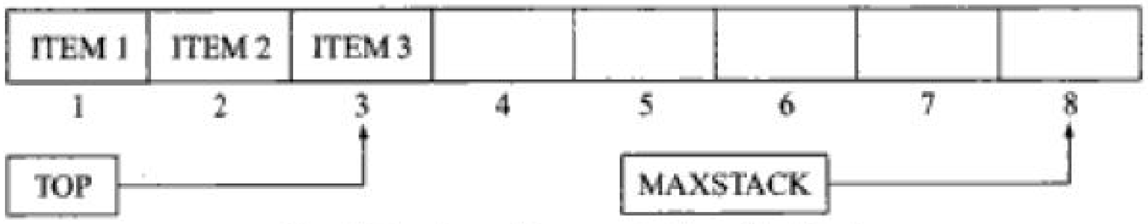


Fig. 6.1 *Array Representation of a Stack*

Algorithm: Push_Operation_on_stack
Input: VALUE: Insert element from user
Output: TOP

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Algorithm: Pop_operation_on_stack
Input: --
Output: TOP

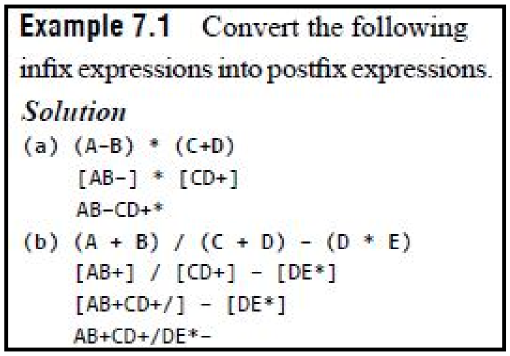
```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

01.4 Applications of stack:

1. Reversing a list: A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

2. Parentheses checker: Stacks can be used to check the validity of parentheses in any algebraic expression. For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket.For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid.

3. Conversion of an infix expression into a postfix expression:



Example 7.1 Convert the following infix expressions into postfix expressions.

Solution

```
(a) (A-B) * (C+D)
    [AB-] * [CD+]
    AB-CD+*
(b) (A + B) / (C + D) - (D * E)
    [AB+] / [CD+] - [DE*]
    [AB+CD+/] - [DE*]
    AB+CD+/DE*-
```

4. Evaluation of a postfix expression: The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation. That is, given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.

Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool.
The infix expression 9 – ((3 * 4) + 8) / 4 can be written as 9 3 4 * 8 + 4 / – using postfix
notation. Look at Table 7.1, which shows the procedure.

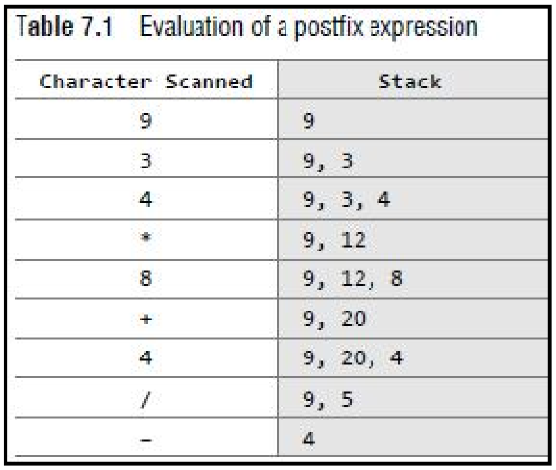
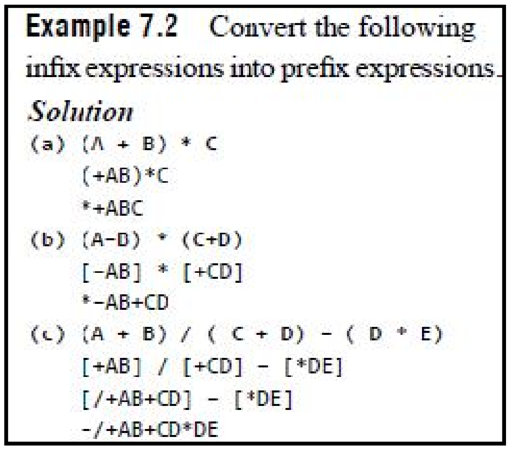


Table 7.1 Evaluation of a postfix expression

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
–	4

5. Conversion of an infix expression into a prefix expression:



Example 7.2 Convert the following infix expressions into prefix expressions.

Solution

```
(a) (A + B) * C
    (+AB)*C
    *+ABC
(b) (A-D) * (C+D)
    [-AB] * [+CD]
    *-AB+CD
(c) (A + B) / ( C + D) - ( D * E)
    [+AB] / [+CD] - [*DE]
    [/+AB+CD] - [*DE]
    -/+AB+CD*DE
```

6. Evaluation of a prefix expression: For example, The infix expression 9 – ((3 * 4) + 8) / 4 can be consider the prefix expression + – 9 2 7 * 8 / 4 12.

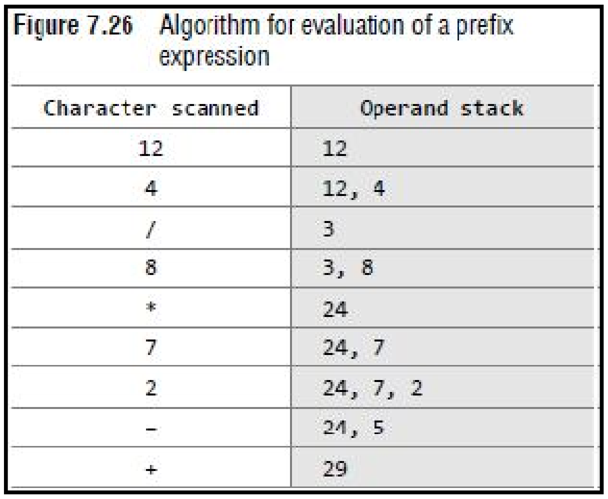


Figure 7.26 Algorithm for evaluation of a prefix expression

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
–	24, 5
+	29

7. Recursion: A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.

8. Tower of Hanoi: Look at Fig. 7.33 which shows three rings mounted on pole A. The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk.

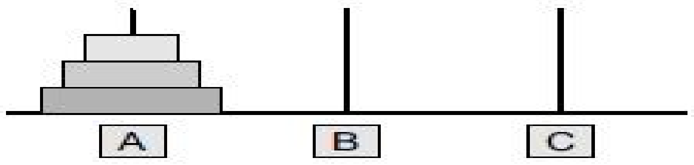


Figure 7.33 Tower of Hanoi

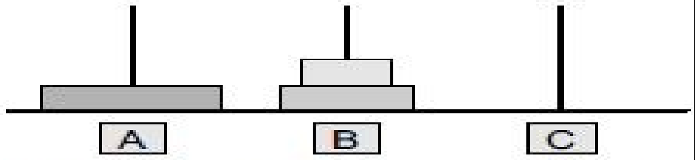


Figure 7.34 Move rings from A to B

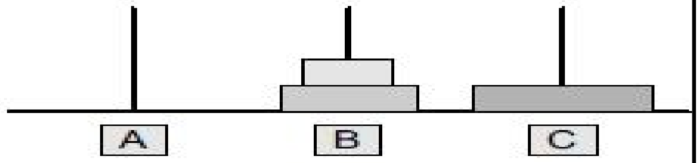


Figure 7.35 Move ring from A to C

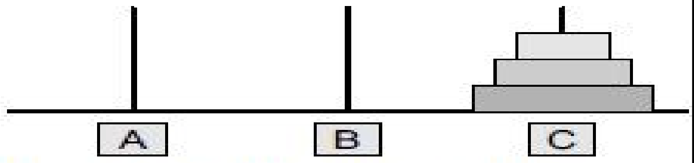


Figure 7.36 Move ring from B to C

Write a program to perform Push, Pop, and Peek operations on a stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 3 // Altering this value changes size of stack created

int st[MAX], top=-1;
void push(int st[], int val);
int pop(int st[]);
int peek(int st[]);
void display(int st[]);

int main()
{
    int val, option;
    do
    {
        printf("\n *****MAIN MENU*****");
        printf("\n 1. PUSH");
        printf("\n 2. POP");
        printf("\n 3. PEEK");
        printf("\n 4. DISPLAY");
        printf("\n 5. EXIT");
        printf("\n Enter your option: ");
        scanf("%d", &option);
        switch(option)
        {
            case 1:
                printf("\n Enter the number to be pushed on stack: ");
                scanf("%d", &val);
                push(st, val);
                break;
            case 2:
                val = pop(st);
                if(val != -1)
                    printf("\n The value deleted from stack is: %d",
                val);
                break;
            case 3:
                val = peek(st);
                if(val != -1)
                    printf("\n The value stored at top of stack is: %d",
                val);
                break;
            case 4:
                display(st);
                break;
        }
    }while(option != 5);
    return 0;
}

void push(int st[], int val)
{
    if(top == MAX-1)
    {
        printf("\n STACK OVERFLOW");
    }
    else
    {
        top++;
        st[top] = val;
    }
}

int pop(int st[])
{
    int val;
    if(top == -1)
    {
        printf("\n STACK UNDERFLOW");
        return -1;
    }
    else
    {
        val = st[top];
        top--;
        return val;
    }
```

```
}

void display(int st[])
{
     int i;
     if(top == -1)
          printf("\n STACK IS EMPTY");
     else
     {
          for(i=top;i>=0;i--)
               printf("\n %d",st[i]);
          printf("\n"); // Added for formatting purposes
     }
}

int peek(int st[])
{
     if(top == -1)
     {
          printf("\n STACK IS EMPTY");
          return -1;
     }
     else
          return (st[top]);
}


Output:

*****MAIN MENU*****
1. PUSH
2. POP
3. PEEK
4. DISPLAY
5. EXIT
Enter your option : 1
Enter the number to be pushed on stack : 500
```

02 Queue:

02.1 Definition:

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.

02.2 Operations:

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

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. Total number of elements present in the queue is front-rear+1, when implemented using arrays.

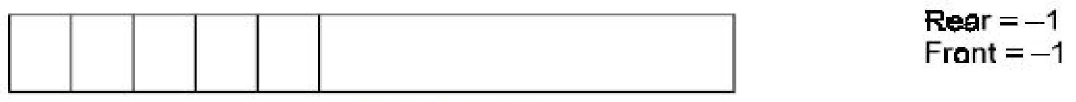


Fig. 4.1. Queue is empty.

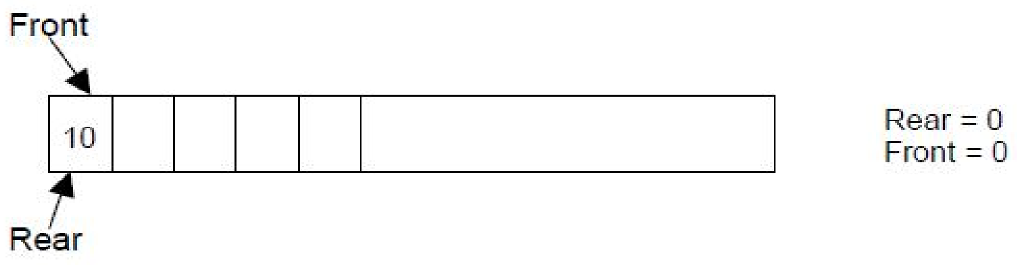


Fig. 4.2. push(10)

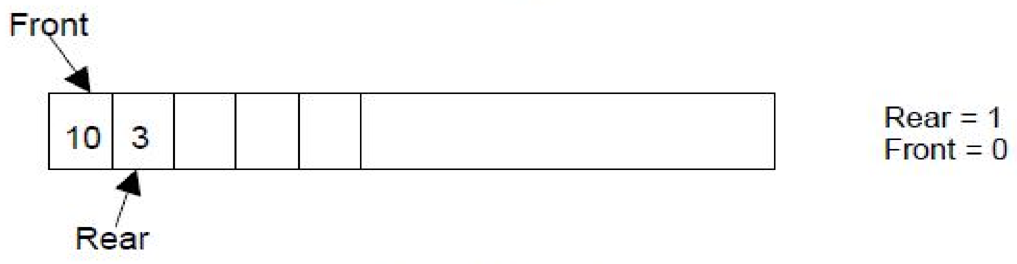


Fig. 4.3. push(3)

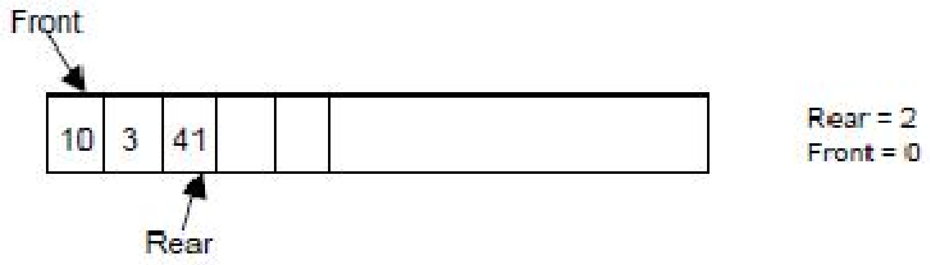


Fig. 4.4. push(41)

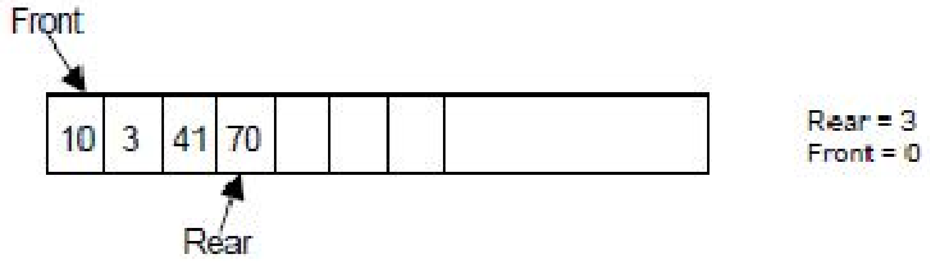


Fig. 4.5. push(70)

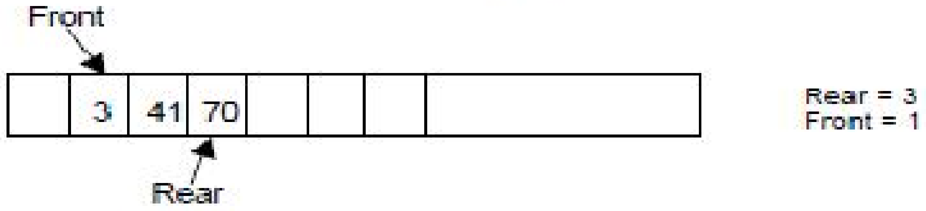


Fig. 4.6. x = pop() (*i.e.*; x = 10)

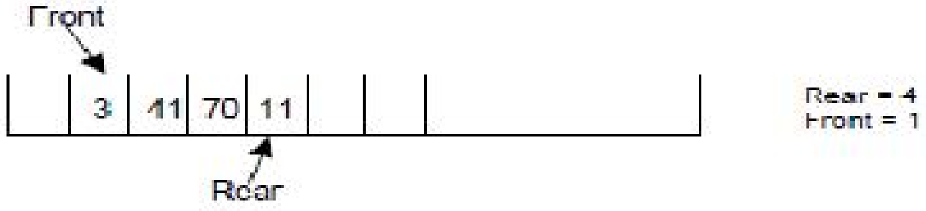


Fig. 4.7. push(11)

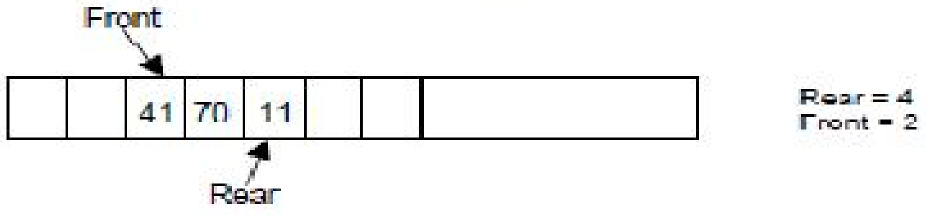


Fig. 4.8. x = pop() (*i.e.*: x = 3)

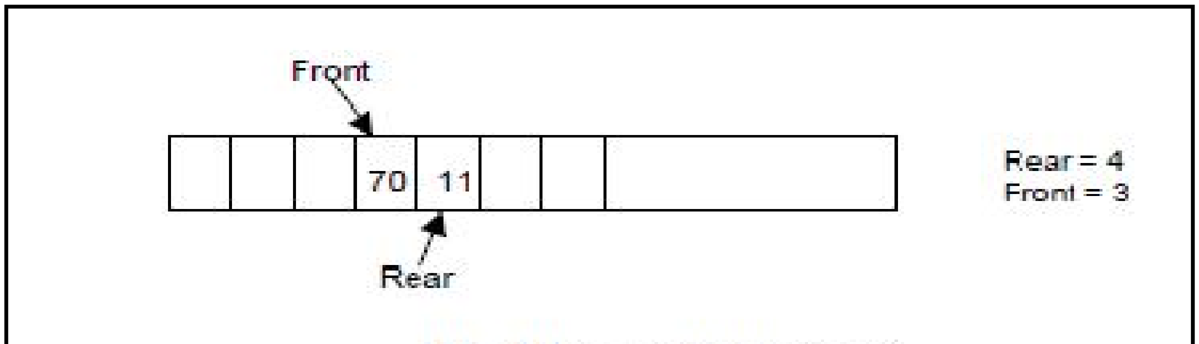


Fig. 4.9. $x = \text{pop}()$ (*ie.*, $x = 41$)

02.3 Array representation of queue:

Queues can be easily represented using linear arrays. As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

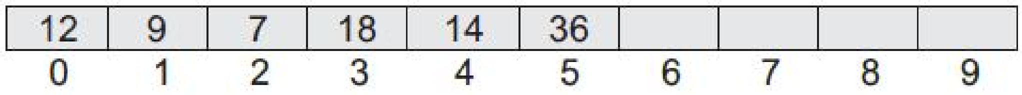


Figure 8.1 Queue

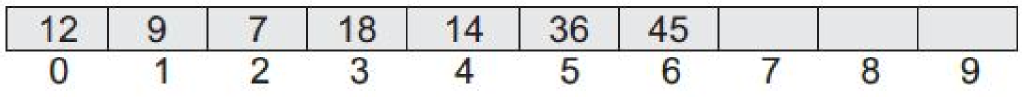


Figure 8.2 Queue after insertion of a new element

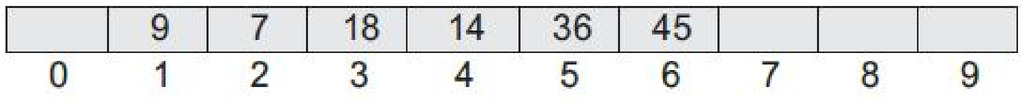


Figure 8.3 Queue after deletion of an element

In Fig. 8.1, FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR. The queue after addition would be as shown in Fig. 8.2. Here, FRONT = 0 and REAR = 6. Every time a new element has to be added, we repeat the same procedure.

If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue. The queue after deletion will be as shown in Fig. 8.3. Here, FRONT = 1 and REAR = 6.

However, before inserting an element in a queue, we must check for overflow conditions. An overflow will occur when we try to insert an element into a queue that is already full. When REAR = MAX – 1, where MAX is the size of the queue, we have an overflow condition. Note that we have written MAX – 1 because the index starts from 0.

Similarly, before deleting an element from a queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If FRONT = –1 and REAR = –1, it means there is no element in the queue. Let us now look at Figs 8.4 and 8.5 which show the algorithms to insert and delete an element from a queue.

Algorithm: insert an element in a queue

Input: NUM: Input from user

Output: FRONT and REAR

```
Step 1: IF REAR = MAX-1
             Write OVERFLOW
             Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
             SET FRONT = REAR = 0
        ELSE
             SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm: Delete element from a queue

Input: --

Output: FRONT and REAR

```
Step 1: IF FRONT = -1 OR FRONT > REAR
             Write UNDERFLOW
        ELSE
             SET VAL = QUEUE[FRONT]
             SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

02.4 Applications of Queue:

1 Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.

2 Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.

3 Queues are used as buffers on MP3 players and portable CD players, iPod playlist.

4 Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.

5 Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.

Write a program to implement a linear queue.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10 // Changing this value will change length of array

int queue[MAX];
int front = -1, rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);

int main()
{
     int option, val;
     do
     {
          printf("\n\n ***** MAIN MENU *****");
          printf("\n 1. Insert an element");
          printf("\n 2. Delete an element");
          printf("\n 3. Peek");
          printf("\n 4. Display the queue");
          printf("\n 5. EXIT");
          printf("\n Enter your option : ");
          scanf("%d", &option);

          switch(option)
          {
               case 1:
                    insert();
                    break;
               case 2:
                    val = delete_element();
                    if (val != -1)
                         printf("\n The number deleted is : %d", val);
                    break;
               case 3:
                    val = peek();
                    if (val != -1)
                         printf("\n The first value in queue is : %d", val);
                    break;
```

```
                case 4:
                     display();
                     break;
           }
      }while(option != 5);
      getch();
      return 0;
}

void insert()
{
      int num;
      printf("\n Enter the number to be inserted in the queue : ");
      scanf("%d", &num);
      if(rear == MAX-1)
           printf("\n OVERFLOW");
      else if(front == -1 && rear == -1)
           front = rear = 0;
      else
           rear++;
      queue[rear] = num;
}

int delete_element()
{
      int val;
      if(front == -1 || front>rear)
      {
           printf("\n UNDERFLOW");
           return -1;
      }
      else
      {
           val = queue[front];
           front++;
           if(front > rear)
                front = rear = -1;
           return val;
      }
}

int peek()
{
      if(front==-1 || front>rear)
      {
           printf("\n QUEUE IS EMPTY");
           return -1;
      }
      else
      {
           return queue[front];
      }
}
```

```
void display()
{
      int i;
      printf("\n");
      if(front == -1 || front > rear)
            printf("\n QUEUE IS EMPTY");
      else
      {
            for(i = front;i <= rear;i++)
                  printf("\t %d", queue[i]);
      }
}
```

Output

```
***** MAIN MENU *****"
1. Insert an element
2. Delete an element
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 50
```

02.5 Circular queue:

Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elements popped.

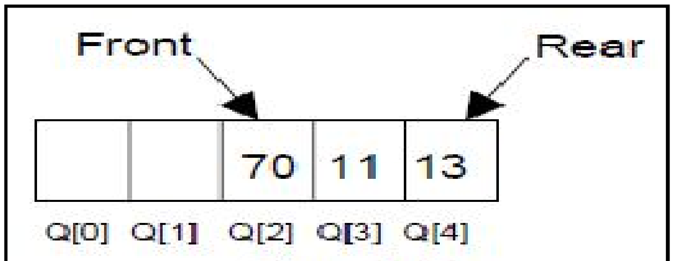


Fig. 4.10

Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed. Because in a queue, elements are always inserted at the *rear* end and hence *rear* points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty. This limitation can be overcome if we use 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.

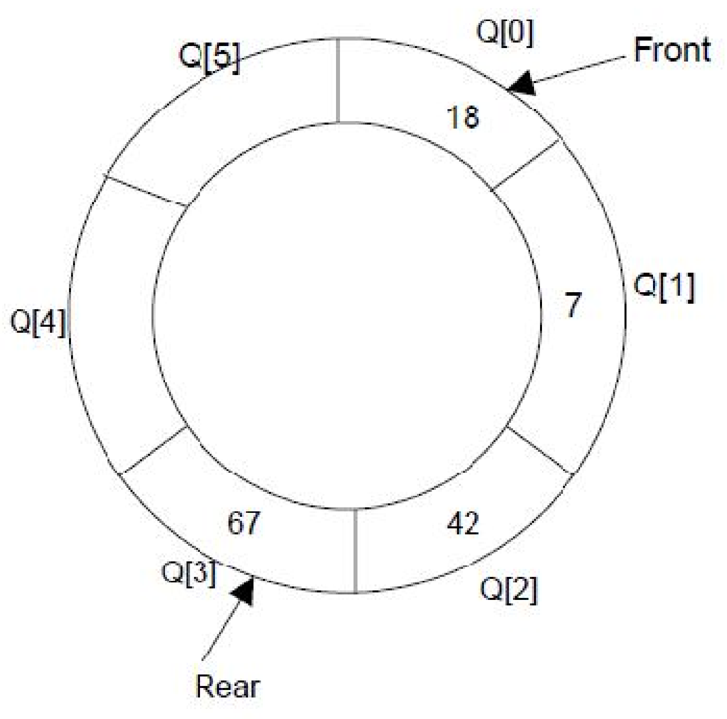


Fig. 4.11. A circular queue after inserting 18, 7, 42, 67.

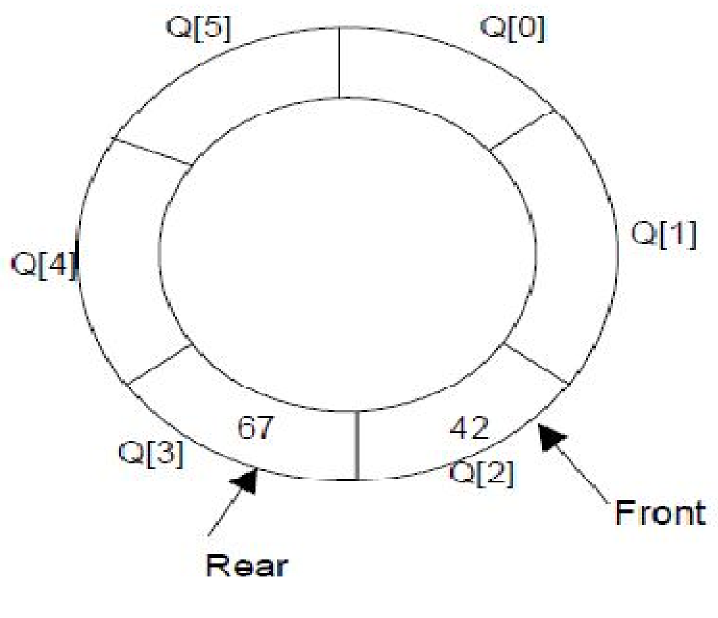


Fig. 4.12. A circular queue after popping 18, 7

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.

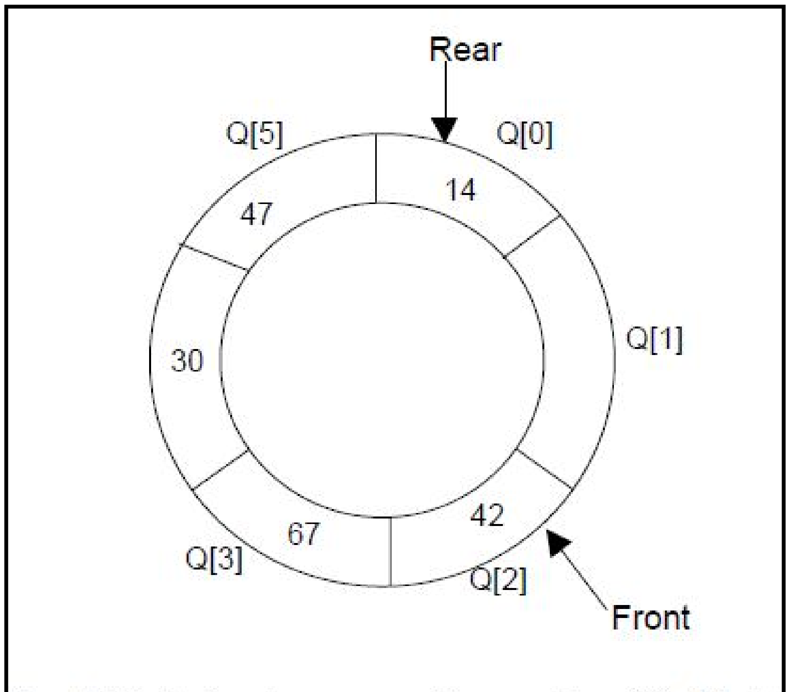


Fig. 4.13. A circular queue after pushing 30, 47, 14

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.

Algorithm: Algorithm to insert an element in a circular queue

Input: VAL: data from user

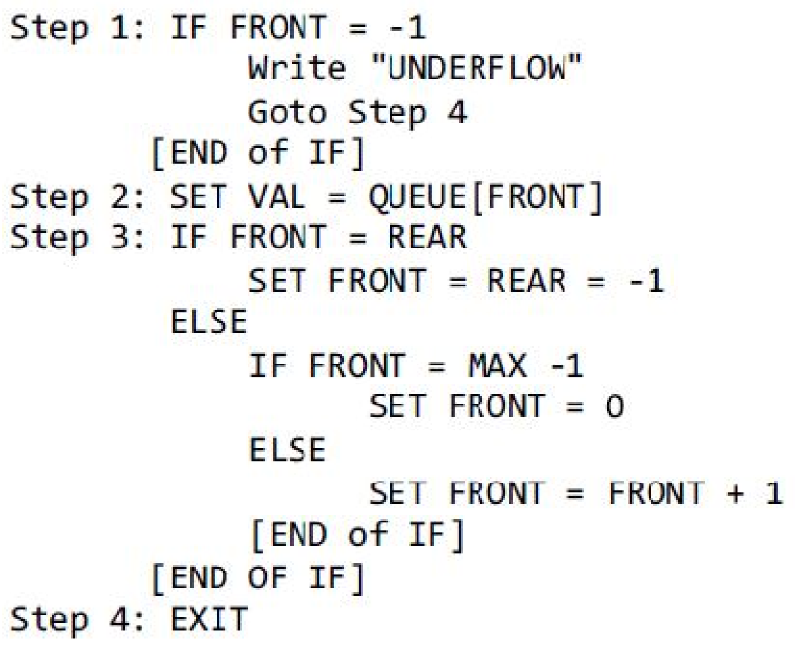
Output: FRONT & REAR

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW"
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Algorithm: Algorithm to delete an element from a circular queue

Input:--

Output: FRONT & REAR



```
Step 1: IF FRONT = -1
             Write "UNDERFLOW"
             Goto Step 4
       [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
             SET FRONT = REAR = -1
        ELSE
             IF FRONT = MAX -1
                   SET FRONT = 0
             ELSE
                   SET FRONT = FRONT + 1
             [END of IF]
       [END OF IF]
Step 4: EXIT
```

Write a program to implement a circular queue.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10

int queue[MAX];
int front=-1, rear=-1;

void insert(void);
int delete_element(void);
int peek(void);
void display(void);

int main()
{
	int option, val;
	clrscr();
	do
	{
		printf("\n ***** MAIN MENU *****");
		printf("\n 1. Insert an element");
		printf("\n 2. Delete an element");
		printf("\n 3. Peek");
		printf("\n 4. Display the queue");
		printf("\n 5. EXIT");
		printf("\n Enter your option : ");
		scanf("%d", &option);
		switch(option)
		{
```

```
                case 1:
                     insert();
                     break;
                case 2:
                     val = delete_element();
                     if(val!=-1)
                          printf("\n The number deleted is : %d", val);
                     break;
                case 3:
                     val = peek();
                     if(val!=-1)
                          printf("\n The first value in queue is : %d", val);
                     break;
                case 4:
                     display();
                     break;
           }
      }while(option!=5);
      getch();
      return 0;
}


void insert()
{
      int num;
      printf("\n Enter the number to be inserted in the queue : ");
      scanf("%d", &num);
      if(front==0 && rear==MAX-1)
      printf("\n OVERFLOW");
      else if(front==-1 && rear==-1)
      {
           front=rear=0;
           queue[rear]=num;
      }
      else if(rear==MAX-1 && front!=0)
      {
           rear=0;
           queue[rear]=num;
      }
      else
      {
           rear++;
           queue[rear]=num;
      }
}

int delete_element()
{
      int val;
      if(front==-1 && rear==-1)
      {
           printf("\n UNDERFLOW");
           return -1;
```

```
	}
	val = queue[front];
	if(front==rear)
		front=rear=-1;
	else
	{
		if(front==MAX-1)
			front=0;
		else
			front++;
	}
	return val;
}

int peek()
{
	if(front==-1 && rear==-1)
	{
		printf("\n QUEUE IS EMPTY");
		return -1;
	}
	else
	{
		return queue[front];
	}
}

void display()
{
	int i;
	printf("\n");
	if (front ==-1 && rear= =-1)
		printf ("\n QUEUE IS EMPTY");
	else
	{
		if(front<rear)
		{
			for(i=front;i<=rear;i++)
				printf("\t %d", queue[i]);
		}
		else
		{
			for(i=front;i<MAX;i++)
				printf("\t %d", queue[i]);
			for(i=0;i<=rear;i++)
				printf("\t %d", queue[i]);
		}
	}
}
```

Output

```
***** MAIN MENU *****
1. Insert an element
2. Delete an element
```

```
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 25
Enter your option : 2
The number deleted is : 25
Enter your option : 3
QUEUE IS EMPTY
Enter your option : 5
```

02.6 Priority queue:

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

1. An element with higher priority is processed before an element with a lower priority.

2. Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors. Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely. For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed.

However, CPU time is not the only factor that determines the priority, rather it is just one among several factors. Another factor is the importance of one process over another. In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

Array Representation of a Priority Queue

When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.

We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space. Look at the two-dimensional representation of a priority queue given below. Given the front and rear values of each queue, the two-dimensional matrix can be formed as shown in Fig. 8.29.

FRONT[K] and REAR[K] contain the front and rear values of row K, where K is the priority number. Note that here we are assuming that the row and column indices start from 1, not 0. Obviously, while programming, we will not take such assumptions.

Insertion To insert a new element with priority K in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element. For example, if we have to insert an element R with priority number 3, then the priority queue will be given as shown in Fig. 8.30.

Deletion To delete an element, we find the first nonempty queue and then process the front element of the first non-empty queue. In our priority queue, the first non-empty queue is the one with priority number 1 and the front element is A, so A will be deleted and processed first. In technical terms, find the element with the smallest K, such that FRONT[K] != NULL.

FRONT	REAR
3	3
1	3
4	5
4	1

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ 1 & & & A & & \\ 2 & B & C & D & & \\ 3 & & & & E & F \\ 4 & I & & & G & H \end{array}$$

Figure 8.29 Priority queue matrix

FRONT	REAR
3	3
1	3
4	1
4	1

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ 1 & & & A & & \\ 2 & B & C & D & & \\ 3 & R & & & E & F \\ 4 & I & & & G & H \end{array}$$

Figure 8.30 Priority queue matrix after insertion of a new element

02.7 Deque:

A deque (pronounced as 'deck' or 'dequeue') is a list in which the elements can be inserted ordeleted at either end. It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) end.

However, no element can be added and deleted from the middle. In the computer's memory, a deque is implemented using either a circular array or a circular doubly linked list. In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque. The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N–1] is followed by Dequeue[0].

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). ie; 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.

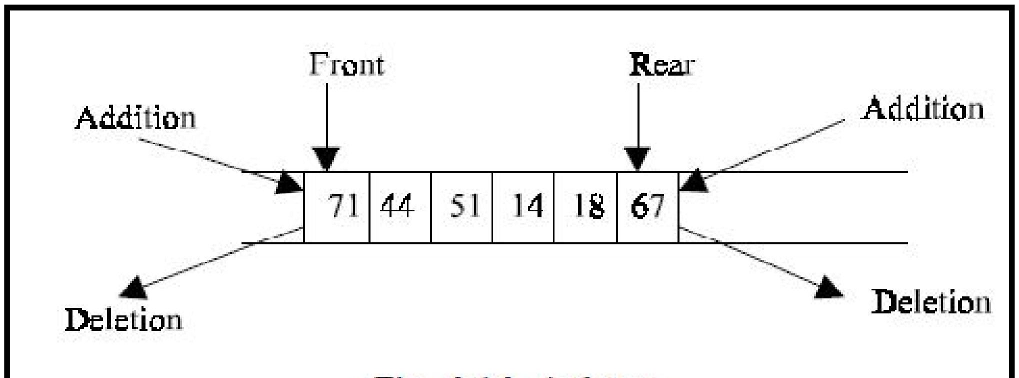


Fig. 4.14. A deque

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 ends, 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.