Unit 2: Linear Data Structure

Notes

Structure

2.0 Introduction

An array is a collection of variables that belong to the same data type. You can also store groups of data of the same data type within an array. An array might belong to any of the data types. It is in fact a data structure which can store a fixed-size collection of elements of the same data type. An array can also store a collection of data, and can be called a **collection of variables of the same kind**. The simplest type of a data structure is a linear array, which is also called a one dimensional

Notes

array. In computer science, an array type is a data type that is meant to describe a collection of elements. In this unit, you will learn about the arrays and its kinds.

A stack is a linear data structure in which an element can be added or removed only at one end called the top of the stack. In the terminology related to stacks, the insert and delete operations are known as **PUSH** and **POP** operations respectively. The last element added to the stack is the first element to be removed, that is, the elements are removed in the opposite order in which they are added to the stack. Hence, a stack works on the principle of last in first out, and is also known as a **last-in-first out** (LIFO) **list**. In this unit, you will learn about the stack organisation and operations on stack.

In the previous unit, you have learnt about stacks. This unit will teach you about representation of stacks. A stack is an abstract data type and it serves as a collection of elements, with two primary principal operations: push, and pop. Push adds an element to the collection and pop removes the most recently added element. As you go through this unit, you will be able to discuss the application and implementation of stacks.

In this unit, you will learn about the queues, their representation and applications. A queue is an abstract data structure which is somewhat similar to stacks. But unlike stacks, a queue is open at both its ends. One end of a queue is always used to insert data (called **enqueue**) and the other is used to remove data (called **dequeue**). queue follows the basic and simple First-In-First-Out methodology, which means that the data item stored first will be accessed first.

A list or sequence is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. A linked list is a sequence of data structures, which are held together by links. A linked list is a sequence of links which contains items. Each link contains a connection to another link. A linked list is the second most-used data structure after an array.

In the previous unit you have studied about lists. Now that you have got an understanding of the basic concepts behind linked list and their types, its time to dive into the common operations that can be performed. This unit will basically discuss the insertion and deletion operations on the linked lists.

In the previous unit, you have learnt about the insertion and deletion operations on linked lists. In this unit, you will learn about the traversing and representation of linked lists. Traversing means to access the elements of the lists for searching an element, find the position for insertion, etc.

2.1 Learning Objectives

After going through this unit, you will be able to:

- discuss Primitive data types,
- explain single-dimensional arrays,
- discuss the memory representation of a single-dimensional arrays,
- explain the various stack related terms,
- discuss the application of stacks,
- explain about priority queues,

- understand merging list and header linked list,
- analyse insertion and deletion of operators in linked lists,
- explain the representation of linked lists.

2.2 Arrays (Ordered Lists)

Notes

Array is one of the data types that can be used for storing a list of elements. When programmers want to store a list of elements under a single variable name, but still want to access and manipulate an individual element of the list, then arrays are used. Arrays can be defined as a fixed-size sequence of elements of the same data type. These elements are stored at contiguous memory locations and can be accessed sequentially or randomly. The programmer can access a particular element of an array by using one or more indices or subscripts. If only one subscript is used then the array is known as a **single-dimensional array**. If more than one subscript is used then the array is known as a **multi-dimensional array**.

Single-Dimensional Arrays

A single-dimensional array is defined as an array in which only one subscript value is used to access its elements. It is the simplest form of an array. Generally, a single-dimensional array is denoted as follows:

```
array_name[L:U]
```

where,

```
array_name = the name of the array
L = the lower bound of the array
U = the upper bound of the array
```

Before using an array in a programme, it needs to be declared. The syntax of declaring a single-dimensional array in C is as follows:

```
data_type array_name[size];
where,
data_type = data type of elements to be stored
in array
array_name = name of the array
size = the size of the array indicating that
the lower bound of the array is 0 and the
upper bound is size-1. Hence, the value of the
subscript ranges from 0 to size-1.
```

For example, in the statement int abc[5], an integer array of five elements is declared and the array elements are indexed from 0 to 4. Once the compiler reads a single-dimensional array declaration, it allocates a specific amount of memory for the array. Memory is allocated to the array at the compile-time before the programme is executed.

Initializing and Accessing Single-Dimensional Array

An array can be initialized in two ways. It can be done by declaring and initializing it simultaneously or by accepting elements of the already declared array

from the user. Once an array is declared and initialized, the elements stored in it can be accessed any time. These elements can be accessed by using a combination of the name of an array and subscript value.

Notes

Example 1: A programme to illustrate the initialization of two arrays and display their elements is as follows:

```
#include<stdio.h>
#include<conio.h>
#define MAX 5
void main()
{
     int A[MAX]={1,2,3,4,5};
     int B[MAX], i;
     clrscr();
     printf("Enter the elements of array B:\n");
     for (i = 0;i < MAX;i++)
     {
          printf("Enter the element: ");
          scanf("%d", &B[i]);
     }
     printf("Elements of array A: \n");
     for (i = 0;i<MAX;i++)
          printf("%d\t", A[i]);
     printf("\nElements of array B: \n");
     for (i = 0;i<MAX;i++)
          printf("%d\t", B[i]);
     getch();
     }
The output of the programme is as follows:
Enter the elements of array b:
Enter a value: 6
Enter a value: 7
Enter a value: 8
Enter a value: 9
Enter a value: 10
Elements of array a:
1    2     3     4     5
Elements of array b:
6     7     8     9     10
```

In this example, an array A is declared and initialized simultaneously and the elements for the array B are accepted from the user, then the elements of both the arrays are displayed.

Once an array is declared and initialized, various operations such as traversing, searching, insertion, deletion, sorting, and merging can be performed on an array. To perform any operation on an array, the elements of the array need to be accessed. The process of accessing each element of an array is known as **traversal**. Generally, the traversal of an array is performed from the element at position 0 to element at position size-1.

Notes

Algorithm 1: Traversing an Array

```
traverse (ARR, size)
1. Set i = 0, sum = 0
2. Print "The elements of the array are: "
3. While i < size //size indicates number of elements
   in the array
        Print ARR[i]
         Set sum = sum + ARR[i]
         Set i = i + 1
   End While
4. Print "Sum of elements of an array: ", sum
5. End
```

Example 2: A programme to illustrate the traversal of an array is as follows:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
/*Function prototype*/
void traverse(int [], int);
void main()
{
    int ARR[MAX];
    int i, size;
    clrscr();
    printf("Enter the number of elements in
    array:\n"); scanf("%d", & size);
    printf("Enter the elements of the array:\n");
    for (i = 0;i < size;i++)
    {
    scanf("%d", & ARR[i]);
    }
    traverse(ARR, size);
    getch();
}
/*Function to find the sum of the elements of matrix*/
void traverse(int ARR[], int size)
```

Notes

```
{
    int i, sum = 0;
    printf("The elements of the array are:\n");
    for (i = 0;i < size; i++)
    {
    printf("%d ", ARR[i]);
    sum + = ARR[i];
}
    printf("\nSum of elements of an array: %d", sum);
}
    The output of the programme is as follows:
    Enter the number of elements in array: 5
    Enter the elements of the array:
    12   23   34   45   56
    The elements of the array are:
    12   23   34   45   56
    Sum of elements of an array: 170
```

Multi-Dimensional Arrays

Multi-dimensional arrays can be described as 'arrays of arrays'. A multi-dimensional array of dimension *n* is a collection of elements, which are accessed with the help of *n* subscript values. Most of the high-level languages, including C, support arrays with more than one dimension. However, the maximum limit of an array dimension is compiler dependent.

The syntax of declaring a multi-dimensional array in C is as follows:

```
    element_type array_name[a][b][c].....[n];
```

where,

```
    element_type = the data type of array
    array_name = name of the array
    [a][b][c].....[n] = array subscripts
```

The arrays of three or more dimensions are not often used because of their huge memory requirements and the complexity involved in their manipulation. Hence, only two-dimensional and three-dimensional arrays have been discussed in brief in this unit.

Two-dimensional arrays

A two-dimensional array is one in which two subscript values are used to access an array element. They are useful when the elements being processed are to be arranged in rows and columns (matrix form).

Generally, a two-dimensional array is represented as follows:

```
    A[Lr : Ur, Lc : Uc]
```

where,

Lr and Lc = the lower bounds of a row and column, respectively

Ur and Uc = the upper bounds of a row and column, respectively

The number of rows in a two-dimensional array can be calculated by using the formula (Ur – Lr + 1) and the number of columns can be calculated by using the formula (Uc – Lc + 1).

Like a single-dimensional array, a two-dimensional array also needs to be declared first. The syntax of declaring a two-dimensional array in C is as follows:

```
data _ type    array _ name
[row_size][column_size];
```

For example, in the statement int a[3][3], an integer array of three rows and three columns is declared. Once a compiler reads a two-dimensional array declaration, it allocates a specific amount of memory for this array.

Notes

Initializing and Accessing Two-dimensional Arrays

A two-dimensional array can be initialized in two ways just like a single-dimensional array, *i.e.,* by declaring and initializing the array simultaneously and by accepting array elements from the user.

Once a two-dimensional array is declared and initialized, the array elements can be accessed anytime. Same as one-dimensional arrays, two-dimensional array elements can also be accessed by using a combination of the name of the array and subscript values. The only difference is that instead of one subscript value, two subscript values are used. The first subscript indicates the row number and the second subscript indicates the column number of a two-dimensional arrays.

Algorithm 2: Traversing a Two-Dimensional Array

```
traverse(ARR, m, n)
1. Set i = 0, sum = 0 //sum stores sum of elements
   of two-dimensional array
2. While i < m //m is number of rows in two-dimensional
   array   Set j = 0
    While j < n //n is number of columns in two-
   dimensional array   Print ARR[i][j]
    Set sum = sum + ARR[i][j]
    Set j = j + 1
    End While
    Set i = i + 1
    End While
3. Print "Sum of the elements of a matrix is : ", sum
4. End
```

Example 3: A programme to illustrate the traversal of a matrix (two-dimensional array) and finding the sum of its elements is as follows:

```
    #include<stdio.h>
#include<conio.h>
#define MAX 10
/*Function prototype*/
void traverse(int [][MAX], int,int);
```

Notes

```
void main()
{

    int ARR[MAX][MAX], i, j, m, n;
    clrscr();
    printf("Enter the number of rows and columns of a
matrix A: ");
    scanf("%d%d", &m, &n);
    printf("Enter the elements of matrix A: \n"); for (i
= 0;i <m; i++)
        for (j = 0; j<n; j++)
            scanf("%d", &ARR[i][j]);
      traverse (ARR, m, n);
      getch();
    }
    /*Function to find sum of elements of the matrix*/
void traverse(int ARR[][MAX], int m, int n)
    {
        int i, j, sum = 0;
        printf("Matrix A is: ");
        for (i = 0;i <m; i++)
        {
            printf ("\n");
            for (j = 0; j<n; j++)
        {
            printf("%d ", ARR[i][j]);
            sum = sum + ARR[i][j];
        }
        }
        printf ("\n Sum of elements of a matrix is: %d",
sum);
    }
    The output of the programme is as follows:
    Enter the number of rows and columns of a matrix A:
3 3 Enter the elements of matrix A:
    1 2 3
    4 5 6
    7 8 9
    Matrix A is:
    1 2 3
    4 5 6
    7 8 9
    Sum of elements of a matrix is: 45
```

Three-Dimensional Arrays

A three-dimensional array is defined as an array in which three subscript values are used to access an individual array element. The three-dimensional array can be declared as follows:

```
int A[3][3][3];
```

Notes

Algorithm 3: Traversing a Three-Dimensional Array

```
traverse(ARR)
1.  Set i = 0, count = 0 //count is used to count the
    number of zeroes in
                         //three-dimensional array
2.  While i < MAX        //MAX is the size of three-
                         dimensional array
    Set j = 0
    While j < MAX
        Set k = 0
        While k < MAX
             If ARR[i][j][k] = 0
                Set count = count + 1
             End If
             Set k = k + 1
      End While
      Set j = j + 1
      End While
    Set i = i + 1
     End While
3.  Print "Number of zeroes in given array are: ",
    count
4.  End
```

Example 4: A programme to illustrate the traversal of a three-dimensional array and to find the number of zeroes in it is as follows:

```
#include<stdio.h>
#include<conio.h>
#define MAX 3
/*Function prototype*/
void traverse(int [][MAX][MAX]);

void main()
{
    int ARR[MAX][MAX][MAX], i, j, k;
    clrscr();
    printf("Enter the elements of an array A
```

Notes

```
        (3×3×3):\n"); for (i = 0; i < MAX; i++)
        for (j = 0; j<MAX; j++)
          for(k = 0; k<MAX; k++)
              scanf("%d", &ARR[i][j][k]);
        traverse (ARR);
        getch();
    }

    void traverse (int ARR[][MAX][MAX])
    {
        int i, j, k, count = 0;
        for (i = 0; i<MAX; i++)
          for (j = 0; j<MAX; j++)
              for (k = 0; k<MAX; k++)
                if(ARR[i][j][k] = = 0)
                  count++;
    printf ("Number of zeroes in given array are: %d",
count);
    }
    The output of the programme is as follows:
    Enter the elements of an array A (3×3×3):
    1 2 3   0 4 5   0 6 7
    0 7 3   2 0 5   6 0 8
    2 0 4   5 0 7   6 9 0
    Number of zeroes in given array are: 8
```

2.3 Representation of an Array

All the elements in an array are always stored next to each other. Also, the memory address of the first element of an array is contained in the name of the array. The memory location, where the first element of an array is stored, is known as the **base address**, which is generally referred to by the name of the array.

Memory Representation of a Single-Dimensional Array

Each element in a single-dimensional array is associated with a unique subscript value, starting from 0 to size-1. The subscript value for an element specifies its position starting from the base address and the difference between the memory addresses of two consecutive array elements is the size of an array's data type. For example, let us study a declaration as follows:

```
int A[5];
```

In this declaration, an integer array of five elements is declared. The array name A refers to the base address of the array. It must be noted that array elements, in **Figure 2.1** are indexed from 0 to 4.

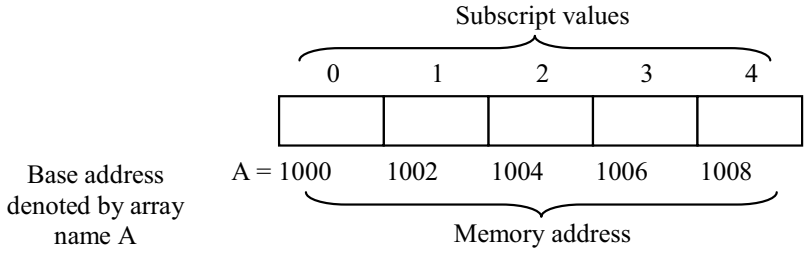


Fig. 2.1 Base Address and Subscript Values of an Array

Notes

Address Calculation in a Single-Dimensional Array

The base address of an array can be used to calculate the address of any element in an array. The formula to calculate the address of an element of an array is as follows:

```
Array[i] = Base + (I–L)*Size
where,
Array = name of an array
I = position of an array element whose address needs to be calculated
L = lower bound of an array
Base = base address of an array
Size = size of each element of an array, i.e., the number of bytes of
space occupied by the individual element of the array
```

Similarly, the length of an array, i.e., the number of elements in the array can be calculated by the formula:

```
Length of Array = U–L+1
where,
U = upper bound of an array
L = lower bound of an array
```

The concept of address calculation in a single-dimensional array will be clear from the examples as provided here.

Example 1: Calculate the address of the fourth element of a floating point array A[10] implemented in C. The base address of the array is 1000.

Solution: Here,

```
Base = 1000        Size = 4 (size of a float variable)
I = 3              L = 0 (in C)
```

Substituting these values in the formula Array [I] = Base + (I–L)*Size, the required address can be calculated as follows:

```
Address of A[3] = 1000 + (3–0) * 4
                = 1000 + 12
                = 1012
```

Example 2: Calculate the address of the element at the position –2 of an integer array A [–4 .. 4]. The base address of the array is 2000. Also, calculate the length of the array.

Notes

Solution: Here,

Base = 2000 Size = 2 (size of an integer variable)

I = –2 L = –4

Substituting these values in the formula Array [I] = Base + (I – L)*Size, the required address can be calculated as follows:

```
Address of A[-2] = 2000 + (-2-(-4))*2
                 = 2000 + (-2 + 4)*2
                 = 2000 + 4
                 = 2004
```

The length of the array is calculated by the formula U – L + 1.

```
Here, U = 4 and L = -4.
Hence, Length = 4-(-4)+1 = 9
```

Thus, the array contains 9 elements.

Memory Representation of a Two-Dimensional Array

Two-dimensional arrays are represented in a linear form to enable the storage of elements in the contiguous memory locations. There are two ways in which a two dimensional array can be represented in the linear form which are row-major order and column-major order.

In a row-major order representation of a two-dimensional array, first the elements of the first row are stored sequentially in the memory, then the elements of the second row are stored sequentially, and so on, as represented in **Figure 2.2**.

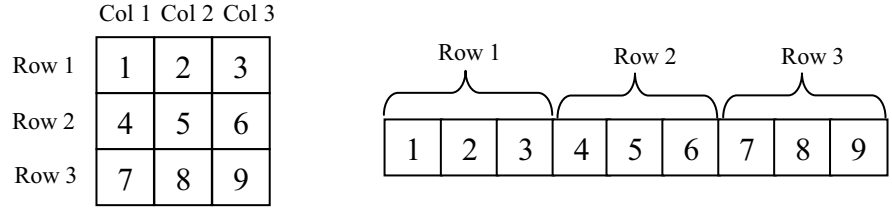


Fig. 2.2 Representation of a Two-Dimensional Array in Row-Major Order

In the column-major order of representation of a two-dimensional array, first the elements of the first column are stored sequentially in the memory, then the elements of the second column are stored sequentially, and so on, as represented in **Figure 2.3**.

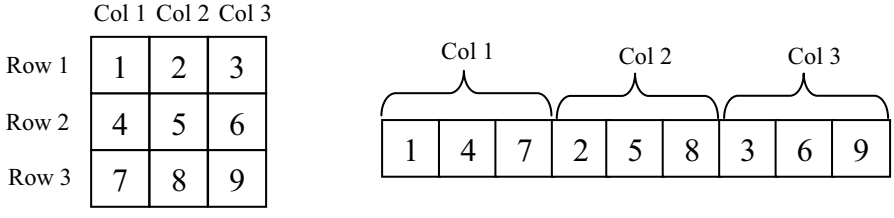


Fig. 2.3 Representation of a Two-Dimensional Arrays in Column-Major Order

Address Calculation in a Two-dimensional Array

The address of any element of a two-dimensional array stored in the row-major order can be calculated using the following formula:

```
A[I,J] = Base + Size*(c*(I-Lr) + (J-Lc))
```

where,

```
A = name of an array
I, j = position of the element whose address has to be calculated
Base = base address of the two-dimensional array
Size = size of an individual elements of the array
C = number of columns in each row
Lr = lower bound of rows
Lc = lower bound of columns
```

To understand the concept of calculation of an address in a two-dimensional array in the row-major order, a few examples have been provided here.

Example 3: If a two-dimensional array C[5 .. 10, –5 .. 9] is stored in a row-major order, calculate the address of C[8, –2] if the base address is 10 and each array element requires 2 bytes of memory space.

Solution: Here,

```
I = 8     J = –2       Base = 10    Size = 2
Lr = 5    Ur = 10      Lc = –5      Uc = 9
c = (Uc-Lc+1) = (9–(–5)+1) = 9+5+1 = 15
```

Substituting these values in the formula A[I, J] = Base + Size*(c*(I – Lr) + (J – Lc)), the required address can be calculated as follows:

```
Address of C[8, –2] = 10 + 2*(15*(8 – 5) + (–2 – (–5))
                    = 10 + 2*(15*3 + 3)
                    = 10 + 2*48
                    = 10 + 96
                    = 106
```

Example 4: The two-dimensional array A[5][10] is stored in the memory using a row major order. Calculate the address of the element A[2][3] if the base address is 150 and each element requires 4 bytes of memory space.

Solution: Here (assuming array is represented in C),

```
I = 2     J = 3        Base = 150   Size = 4
Lr = 0    Ur = 4       Lc = 0       Uc = 9
c = (Uc – Lc + 1) = (9 – 0 + 1) = 10
```

Substituting these values in the formula A[I, J] = Base + Size*(c*(I – Lr) + (J – Lc)), the required address can be calculated as follows:

```
Address of A[2][3] = 150 + 4*(10*(2 – 0) + (3 – 0))
                   = 150 + 4*23
                   = 150 + 92
                   = 242
```

Notes

Notes

The address of any element of a two-dimensional array stored in the column major order can be calculated using the following formula:

A[I, J] = Base + Size*((I – Lr) + r*(J – Lc))

where,

A = the array name

I, *j* = the position of the element whose address has to be calculated

Base = the base address of the two-dimensional array

Size = the size of the individual elements of the array

r = the number of rows in each column

Lr = the lower bound of rows

Lc = the lower bound of columns

To understand the concept of calculation in a two-dimensional array in a column major order, a few examples have been provided here.

Example 5: If a two-dimensional array C[5 .. 10, –5 .. 9] is stored in a column major order, calculate the address of C[8, –2] if the base address is 10 and each array element requires 2 bytes of memory space.

Solution: Here,

I = 8 J = –2 Base = 10 Size = 2

Lr = 5 Ur = 10 Lc = –5 Uc = 9

r = (Ur-Lr+1) = (10–5+1) = 6

Substituting these values in the formula A[I, J] = Base + Size*((I – Lr) + r*(J – Lc)), the required address can be calculated as follows:

Address of C[8, –2] = 10 + 2*((8 – 5) + 6*(–2–(–5))
= 10 + 2*(3+18)
= 10 + 2*21
= 10 + 42
= 52

Example 6: The two-dimensional array A[5][10] is stored in memory using the column-major order. Calculate the address of the element A[2][3] if the base address is 150 and each element requires 4 bytes of memory space.

Solution: Here (assuming array represented in C),

I = 2 J = 3 Base = 150 Size = 4

Lr = 0 Ur = 4 Lc = 0 Uc = 9

r = (Ur - Lr + 1) = (4 – 0 + 1) = 5

Substituting these values in the formula A[I, J] = Base + Size*((I – Lr) + r*(J – Lc)), the required address can be calculated as follows:

Address of A[2][3] = 150 + 4*((2 – 0) +5*(3 – 0))
= 150 + 4*17
= 150 + 68
= 218

2.4 Stack Related Terms and Operations on Stack

Notes

A stack can be organized (represented) in the memory either as an array or as a singly-linked list. In both the cases, insertion and deletion of elements is allowed only at one end. Insertion and deletion at the middle of an array or a linked list is not allowed. An array representation of a stack is static, but linked list representation is dynamic in nature. Though array representation is a simple technique, it provides less flexibility and is not very efficient with respect to memory utilization. This is because if the number of elements to be stored in a stack is less than the allocated memory then the memory space will be wasted. Conversely, if the number of elements to be handled by a stack is more than the size of the stack, then it will not be possible to increase the size of the stack to store these elements. In this section, only the array organization of a stack will be discussed.

When a stack is organized as an array, a variable named Top is used to point to the top element of the stack. Initially, the value of Top is set as –1 to indicate an empty stack. Before inserting a new element onto a stack, it is necessary to test the condition of overflow. Overflow occurs when a stack is full and there is no space for a new element and an attempt is made to push a new element. If a stack is not full then the push operation can be performed successfully. To push an item onto a stack, Top is incremented by one and the element is inserted at that position.

Similarly, before removing the top element from the stack, it is necessary to check the condition of underflow. Underflow occurs when a stack is empty and an attempt is made to pop an element. If a stack is not empty, POP operation can be performed successfully. To POP (or remove) an element from a stack, the element at the top of the stack is assigned to a local variable and then Top is decremented by one.

The total number of elements in a stack at a given point of time can be calculated from the value of Top as follows:

number of elements = Top + 1

Figure 2.4 shows an empty stack with size 3 and Top = –1.

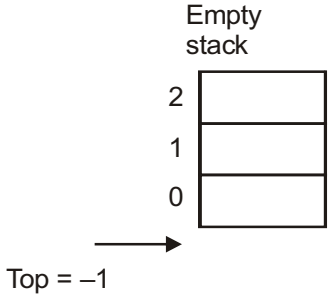
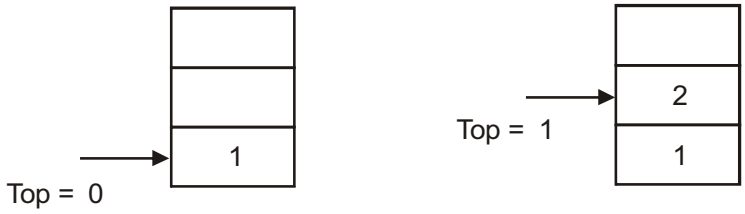


Fig. 2.4 An Empty Stack

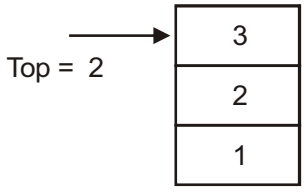
To insert an element 1 in a stack, Top is incremented by one and the element 1 is stored at stack [Top]. Similarly, other elements can be added to the same stack until Top reaches 2, as shown in **Figure 2.5**. To POP an element from the stack (data element 3), Top is decremented by one, which removes the element 3 from the stack. Similarly,

Notes

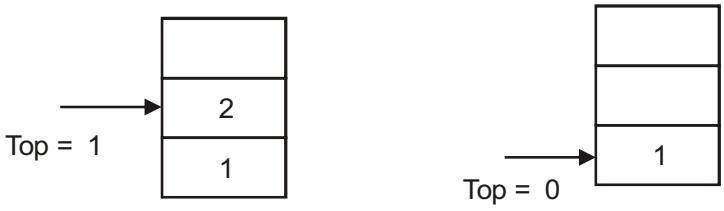
other elements can be removed from the stack until Top reaches –1. Figure 2.5 shows different states of stack after performing PUSH and POP operations on it.



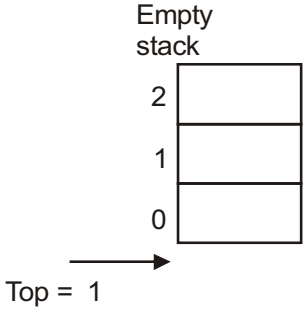
(a) Stack after pushing the element 1 **(b) Stack after pushing the element 2**



(c) Stack after pushing the element 3



(d) Stack after popping the element 3 **(e) Stack after popping the element 2**



(f) Stack after popping the element 1

Fig. 2.5 Various States of Stack after Push and Pop Operations

To implement a stack as an array in C language, the following structure named **Stack** needs to be defined as follows:

```
struct stack
{
  int item[MAX]; /*MAX is the maximum size of the
array*/
int Top;
```

Algorithm 1: Push Operation on Stack

```
push(s, element) //s is a pointer to stack
1. If s–>Top = MAX – 1 //checking for stack overflow
   Print "Overflow: Stack is full!" and go to step 5
   End If
2. Set s–>Top = s–>Top + 1 //incrementing Top by 1
3. Set s–>item[s–>Top] = element //inserting element
   in the stack
4. Print "Value is pushed onto the stack…"
5. End
```

Notes

Algorithm 2: Pop Operation on Stack

```
pop(s)
1. If s–>Top = -1 //checking for stack underflow
   Print "Underflow: Stack is empty!"
   Return 0 and go to step 5
    End If
2. Set popped = s–>item[s–>Top] //taking off the top
   element from the stack
3. Set s–>Top = s–>Top - 1 //decrementing Top by 1
4. Return popped
5. End
```

Example 1: A programme to implement a stack as an array is as follows:

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
#define True 1
#define False 0
typedef struct stack
{
    int item[MAX];
    int Top;
}stk;
/*Function prototypes*/
void createstack(stk *); /*to create an empty stack*/
void push(stk *, int); /*to push an element onto the
stack*/
int pop(stk *); /*to pop the top element from the
stack*/
int is empty(stk *); /*to check for the underflow
condition*/
int is full(stk *); /*to check for the overflow
condition*/
void main()
{
```

Notes

```
    int choice;
    int value;
    stk s;
    create stack(&s);
do{
    clrscr();
    printf("\n\tMain Menu");
    printf("\n1. Push");
    printf("\n2. Pop");
    printf("\n3. Exit\n");
    printf("\nEnter your choice:");
    scanf("%d", and choice);
    switch(choice)
    {
case 1: printf("\nEnter the value to be inserted:");
        scanf("%d", & value);
        push(and s, value);
        getch();
        break;
case 2: value = pop(and s);
        if (value= =0)
          printf("\nUnderflow: Stack is empty!");
    else
          printf ("\nPopped item is: %d", value);
        getch();
        break;
case 3: exit();
    default: printf ("\nInvalid choice!");
    }
    }while(1);
    }
void create stack(stk *s)
{
    s->Top=-1;
}
void push(stk *s, int element)
{
    if (isfull(s))
```

```
    {
         printf("\nOverflow: Stack is full!");
         return;
    }
    s->Top++;

    s->item[s->Top] = element;
    printf ("\nValue is pushed onto the stack...");
    }
    int pop(stk *s)
{
    int popped;
    if (isempty(s))
         return 0;
    popped = s->item[s->Top];
    s->Top—;
    return popped;
}
int isempty(stk *s)
{
    if (s->Top==-1)
         return True;
    else return False;
}
int isfull(stk *s)
{
    if (s->Top==MAX-1)
         return True;
    else return False;
}
The output of the programme is as follows:
 Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 23
Value is pushed onto the stack...
```

Notes

Notes

```
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 35
Value is pushed onto the stack...
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 1
Enter the value to be inserted: 40 Value is pushed
onto the stack...
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped item is: 40
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped item is: 35
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Popped item is: 23
    Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 2
Underflow: Stack is empty!
```

```
Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 3
```

Notes

2.5 Application and Implementation of Stack

Stacks are used where the last-in-first-out principle is required like reversing strings, checking whether the arithmetic expression is properly parenthesized, converting infix notation to postfix and prefix notations, evaluating postfix expressions, implementing recursion and function calls, etc. This section discusses some of these applications.

Reversing Strings

A simple application of stacks is reversing strings. To reverse a string, the characters of a string are pushed onto a stack one-by-one as the string is read from left to right. Once all the characters of the string are pushed onto the stack, they are popped one-by-one. Since the character last pushed in comes out first, subsequent POP operations result in reversal of the string.

For example, to reverse a string 'REVERSE', the string is read from left to right and its characters are pushed onto a stack, starting from the letter R, then E, V, E, and so on, as shown in **Figure 2.6**.

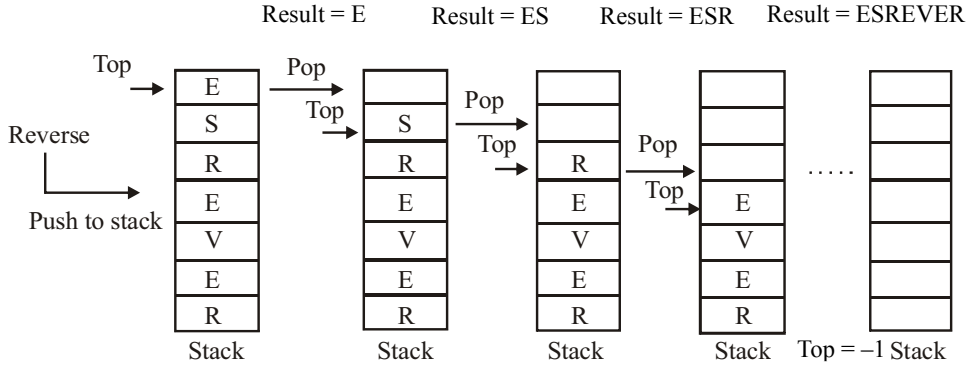


Fig. 2.6 Reversing a String using a Stack

Once all the letters are stored in a stack, they are popped one-by-one. Since the letter at the top of the stack is E, it is the first letter to be popped. The subsequent POP operations take out the letters S, R, E, and so on. Thus, the resultant string is the reverse of original one as shown in **Figure 2.6**.

Algorithm 1: String Reversal Using Stack

```
    reversal(s, str)
1.  Set i = 0
2.  While(i < length_of_str)
        Push str[i] onto the stack
        Set i = i + 1
```

Notes

```
    End While
3.  Set i = 0
4.  While(i < length_of_str)
        Pop the top element of the stack and store it
    in str[i]   Set i = i + 1
        End While
5.  Print "The reversed string is: ", str
6.  End
```

Example 1: The following is a programme to reverse a given string using stacks:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define MAX 101
typedef struct stack
Representation of Stack NOTES
{
    char item[MAX];
    int Top;
}stk;
/*Function prototypes*/
void createstack(stk *);
void reversal(stk *, char *);
void push(stk *, char);
char pop(stk *);
void main()
{
    stk s;
    char str[MAX];
    int i;
    createstack(&s);
    clrscr();
    do
    {
        printf("Enter any string (max %d
characters): ", MAX-1);
        for(i = 0;i<MAX;i++)
    {
        scanf("%c", &str[i]);
        if(str[i]=='\n')
            break;
```

```
    }
    str[i]='\0';
  }while(strlen(str)==0);
  reversal(&s, str);
  getch();
}
/*Function definitions*/
void createstack(stk *s)
{
    s->Top=-1;
}
void reversal(stk *s, char *str)
{
    int i;
    for (i=0;i<strlen(str);i++)
    push(s, str[i]);
for(i=0;i<strlen(str);i++)
    str[i]=pop(s);
printf("\nThe reversed string is: %s", str);
}
void push(stk *s, char item)
{
    s->Top++;
    s->item[s->Top]=item;
}
char pop(stk *s)
{
    char popped;
    popped=s->item[s->Top];
    s->Top--;
    return popped;
}
The output of the programme is as follows:
Enter any string (max 100 characters): Hello World
The reversed string is: dlroW olleH
```

Notes

Converting Infix Notation to Postfix and Prefix or Polish Notations

Another important application of stacks is the conversion of expressions from infix notation to postfix and prefix notations. The general way of writing arithmetic expressions is known as **infix notation** where the binary operator is placed between

Notes

two operands on which it operates. For simplicity, expressions containing unary operators have been ignored. For example, the expressions 'a + b' and '(a – c)*d', '[(a + b)*(d/f) –f]' are in infix notation. The order of evaluation of these expressions depends on the parentheses and the precedence of operators. For example, the order of evaluation of the expression '(a + b)*c' is different from that of 'a + (b*c)'. As a result, it is difficult to evaluate an expression in an infix notation. Thus, the arithmetic expressions in the infix notation are converted to another notation which can be easily evaluated by a computer system to produce a correct result.

A Polish mathematician Jan Lukasiewicz suggested two alternative notations to represent an arithmetic expression. In these notations, the operators can be written either before or after the operands on which they operate. The notation in which an operator occurs before its operands is known as the **prefix notation** (also known as **Polish notation**). For example, the expressions '+ ab' and '*– acd' are in prefix notation. On the other hand, the notation in which an operator occurs after its operands is known as the **postfix notation** (also known as **Reverse Polish** or **suffix notation**). For example, the expressions 'ab +' and 'ac – d*' are in postfix notation.

A characteristic feature of prefix and postfix notations is that the order of evaluation of expression is determined by the position of the operator and operands in the expression. That is, the operations are performed in the order in which the operators are encountered in the expression. Hence, parentheses are not required for the prefix and postfix notations. Moreover, while evaluating the expression, the precedence of operators is insignificant. As a result, they are compiled faster than the expressions in infix notation. Note that the expressions in an infix notation can be converted to both prefix and postfix notations. The subsequent sections will discuss both the types of conversions.

Conversion of Infix to Postfix Notation

To convert an arithmetic expression from an infix notation to a postfix notation, the precedence and associativity rules of operators should always kept in mind. The operators of the same precedence are evaluated from left to right. This conversion can be performed either manually (without using stacks) or by using stacks. Following are the steps for converting the expression manually:

(i) The actual order of evaluation of the expression in infix notation is determined. This is done by inserting parentheses in the expression according to the precedence and associativity of operators.

(ii) The expression in the innermost parentheses is converted into postfix notation by placing the operator after the operands on which it operates.

(iii) Step 2 is repeated until the entire expression is converted into a postfix notation.

For example, to convert the expression 'a + b*c' into an equivalent postfix notation, the steps will be as follows:

(i) Since the precedence of * is higher than +, the expression b*c has to be evaluated first. Hence, the expression is written as follows:

(a + (b*c))

(ii) The expression in the innermost parentheses, that is, b*c is converted into its postfix notation. Hence, it is written as bc*. The expression now becomes as follows:

(a + bc*)

(iii) Now the operator + has to be placed after its operands. The two operands for + operator are a and the expression bc*. The expression now becomes as follows:

```
(abc* +)
```

Hence, the equivalent postfix expression will be as follows:

```
abc*+
```

Notes

When expressions are complex, manual conversion becomes difficult. On the other hand, the conversion of an infix expression into a postfix expression is simple when it is implemented through stacks. In this method, the infix expression is read from left to right and a stack is used to temporarily store the operators and the left parenthesis. The order in which the operators are pushed on to and popped from the stack depends on the precedence of operators and the occurrence of parentheses in the infix expression. The operands in the infix expression are not pushed on to the stack, rather they are directly placed in the postfix expression. Note that the operands maintain the same order as in the original infix notation.

Algorithm 2: Infix to Postfix Conversion

```
infix to postfix(s, infix, postfix)
1. Set i = 0
2. While (i < number_of_symbols_in_infix)
    If infix[i] is a whitespace or comma
       Set i = i + 1 and go to step 2
    If infix[i] is an operand, add it to postfix
    Else If infix[i] = '(', push it onto the stack
    Else If infix[i] is an operator, follow these
   steps:
```

i. For each operator on the top of stack whose precedence is greater than or equal to the precedence of the current operator, pop the operator from stack and add it to postfix

ii. Push the current operator onto the stack

```
   Else If infix[i] = ')', follow these steps:
```

i. Pop each operator from top of the stack and add it to postfix until '(' is encountered in the stack

ii. Remove '(' from the stack and do not add it to postfix

```
       End If
       Set i = i + 1
   End While
3. End
```

For example, consider the conversion of the following infix expression to a postfix expression:

```
a -(b + c)*d/f
```

Notes

Initially, a left parenthesis '(' is pushed onto the stack and the infix expression is appended with a right parenthesis, ')'. The initial state of the stack, infix expression and postfix expression are shown in **Figure 2.7**.

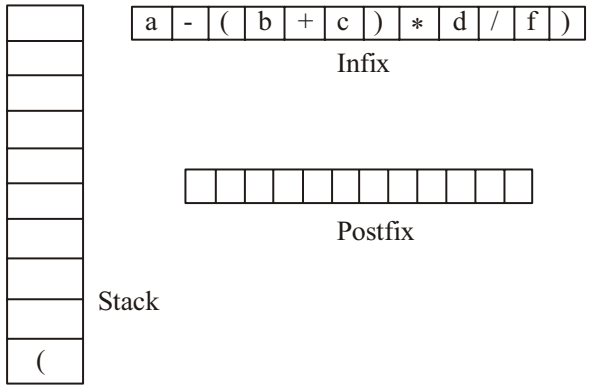
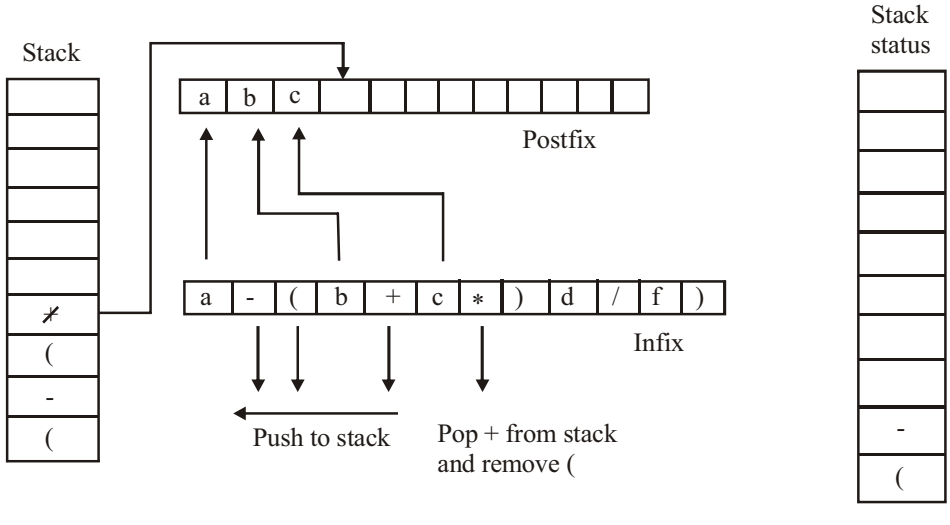


Fig. 2.7 Initial State of the Stack, Infix Expression, and Postfix Expression

infix is read from left to right and the following steps are performed:

1. The operand a is encountered, which is directly put to postfix.
2. The operator – is pushed onto the stack.
3. The left parenthesis '(' is pushed onto the stack.
4. The next element is b, which being an operand is directly put to postfix.
5. +, being an operator, is pushed onto the stack.
6. Next, c is put to postfix.
7. The next element is the right parenthesis ')' and hence, the operators at the top of the stack are popped until '(' is encountered in the stack. Till then, the only operator in the stack above the '(' is +, which is popped and put to postfix. '(' is popped and removed from the stack, as shown in **Figure 2.8(a)**. **Figure 2.8(b)** shows the current position of stack.



(a) Postfix Expression when + is Popped **(b) State of the Stack**

Fig. 2.8 Intermediate States of Postfix and Infix Expressions and the Stack

8. Then, the next element * is an operator and hence, it is pushed onto the stack.
9. Then, *d* is put to postfix.
10. The next element is/. Since the precedence of/is same as the precedence of *, the operator * is popped from the stack and/is pushed onto the stack, as shown in **Figure 2.9**.
11. The operand f is directly put to postfix after which, ')' is encountered.
12. On reaching ')', the operators in stack before the next '(' is reached and popped. Hence,/and – are popped and put to postfix as shown in **Figure 2.9**.
13. '(' is removed from the stack. Since the stack is empty, the algorithm is terminated and postfix is printed.

Notes

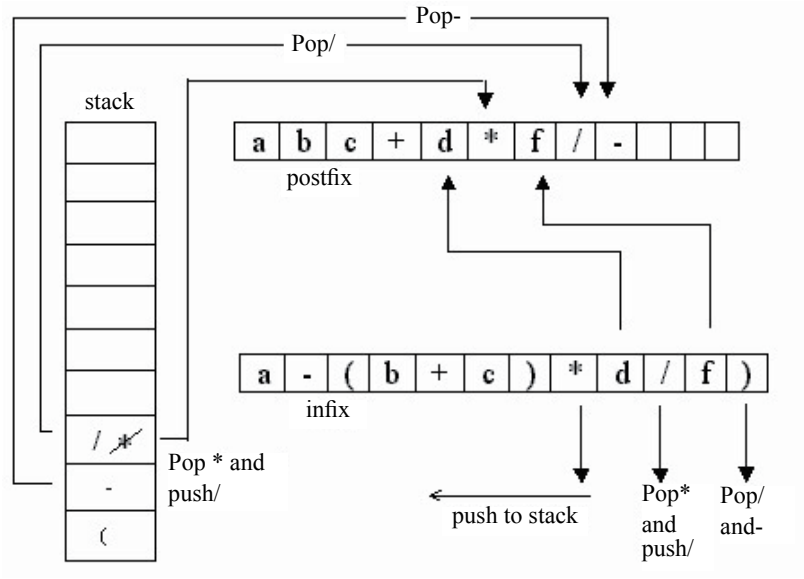


Fig. 2.9 The State when – and/are Popped

The step-wise conversion of expression a – (b + c)**d*/*f* into its equivalent postfix expression is shown in **Table 2.1**.

Table 2.1 Conversion of Infix Expression into Postfix

Element	Action Performed	Stack Status	Postfix Expression
A	Put to postfix	(	A
–	Push	(-	a
(	Push	(-(	a
b	Put to postfix	(-(	ab
+	Push	(-(+	ab
c	Put to postfix	(-(+	abc
)	Pop +, put to postfix, pop (	(- abc+	
*	Push	(-*	abc+
d	Put to postfix	(-*	abc+d
/	Pop *, put to postfix, push/	(-/	abc+d*
f	Put to postfix	(-/	abc+d*f
)	Pop/and -	Empty	abc+d*f/-

Notes

Conversion of Infix to Prefix Notation

The conversion of an infix expression to a prefix expression is similar to the conversion of infix to postfix expression. The only difference is that the expression in an infix notation is scanned in reverse order, that is, from right to left. Therefore, the stack in this case stores the operators and the closing (right) parenthesis.

Algorithm 3: Infix to Prefix Conversion

```
infix to prefix(s, infix, prefix)
1.  Set i = 0
2.  While (i < number_of_symbols_in_infix)
    If infix[i] is a white space or comma
    Set i = i + 1 go to step 2
    If infix[i] is an operand, add it to prefix
    Else If infix[i] = ')', push it onto the stack
    Else If infix[i] is an operator, follow these steps:
    i.  For each operator on the top of stack whose
        precedence is greater than or equal to the
        precedence of the current operator, pop the
        operator from stack and add it to prefix
    ii. Push the current operator onto the stack
    Else If infix[i] = '(', follow these steps:
    i.  Pop each operator from top of the stack and
        add it to prefix until ')' is encountered in
        the stack
    ii. Remove ')' from the stack and do not add it
    to prefix
    End If
    Set i = i + 1
    End While
3.  Reverse the prefix expression
4.  End
```

For example, consider the conversion of the following infix expression to a prefix expression:

$a - (b + c)*d/f$

The step-wise conversion of the expression $a - (b + c)*d/f$ into its equivalent prefix expression is shown in **Table 2.2**. Note that initially ')' is pushed onto the stack, and '(' is inserted in the beginning of the infix expression. Since the infix expression is scanned from right to left, but elements are inserted in the resultant expression from left to right, the prefix expression needs to be reversed.

Table 2.2 Conversion of Infix Expression into Prefix Expression

Element	Action Performed	Stack Status	Prefix Expression
f	Put to expression	)	f
/	Push	)/	f
d	Put to expression	)/	fd

*	Push	)/*	fd
)	Push	)/*)	fd
c	Put to expression	)/*)	fdc
+	Push	)/*)+	fdc
b	Put to expression	)/*)+	fdcb
(	Pop and + and put to expression, pop)	)/*	fdcb +
-	Pop *,/and push -	)-	fdcb + */
a	Put to expression	)/*-	fdcb + a
(	Pop - and put to expression, pop (Reverse the resultant expression	Empty	fdcb + */a- -a/* + bcdf

The equivalent prefix expression is –a/* + bcdf.

Evaluation of Postfix Expression

In a computer system, when an arithmetic expression in an infix notation needs to be evaluated, it is first converted into its postfix notation. The equivalent postfix expression is then evaluated. Evaluation of postfix expressions is also implemented through stacks. Since the postfix expression is evaluated in the order of appearance of operators, parentheses are not required in the postfix expression. During evaluation, a stack is used to store the intermediate results of evaluation.

Since an operator appears after its operands in a postfix expression, the expression is evaluated from left to right. Each element in the expression is checked to find out whether it is an operator or an operand. If the element is an operand, it is pushed onto the stack. On the other hand, if the element is an operator, the first two operands are popped from the stack and an operation is performed on them. The result of this operation is then pushed back to the stack. This process is repeated until the entire expression is evaluated.

Algorithm 4: Evaluation of a Postfix Expression

```
evaluation of postfix(s, postfix)
1.  Set i = 0, RES = 0.0
2.  While (i < number_of_characters_in_postfix)
    If postfix[i] is a white space or comma
        Set i = i + 1 and continue
    If postfix[i] is an operand, push it onto the stack
    If postfix[i] is an operator, follow these steps:
    i.   Pop the top element from stack and store it
         in operand2
    ii.  Pop the next top element from stack and store
         it in operand1
    iii. Evaluate operand2 op operand1, and store the
         result in  RES (op is the current operator)
    iv.  Push RES back to stack
```

Notes

```
    End If
    Set i = i + 1
    End While
3.  Pop the top element and store it in RES
4.  Return RES
5.  End
```

Notes

For example, consider the evaluation of the following postfix expression using stacks:

```
abc + d*f/-
where,
a = 6
b = 3
c = 6
d = 5
f = 9
```

After substituting the values of a, b, c, d and f, the postfix expression becomes as follows:

```
636 + 5*9/-
```

The following are the steps performed to evaluate an expression:

1. The expression to be evaluated is read from left to right and each element is checked to find out if it is an operand or an operator.
2. First element is 6, which being an operand is pushed onto the stack.
3. Similarly, the operands 3 and 6 are pushed onto the stack. 4. Next element is +, which is an operator. Hence, the element at the top of stack 6 and the next top element 3 are popped from the stack, as shown in **Figure 2.10**.

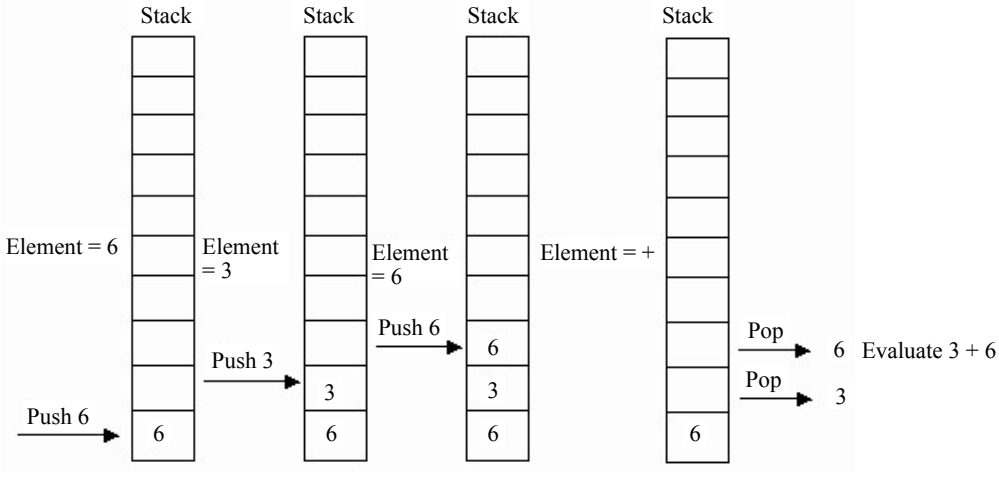


Fig. 2.10 Evaluation of the Expression using Stacks

5. Expression 3 + 6 is evaluated and the result, that is 9, is pushed back to stack, as shown in **Figure 2.11**.
6. Next element in the expression, that is 5, is pushed to the stack.

7. Next element is *, which is a binary operator. Hence, the stack is popped twice and the elements 5 and 9 are taken off from the stack, as shown in **Figure 2.11.**

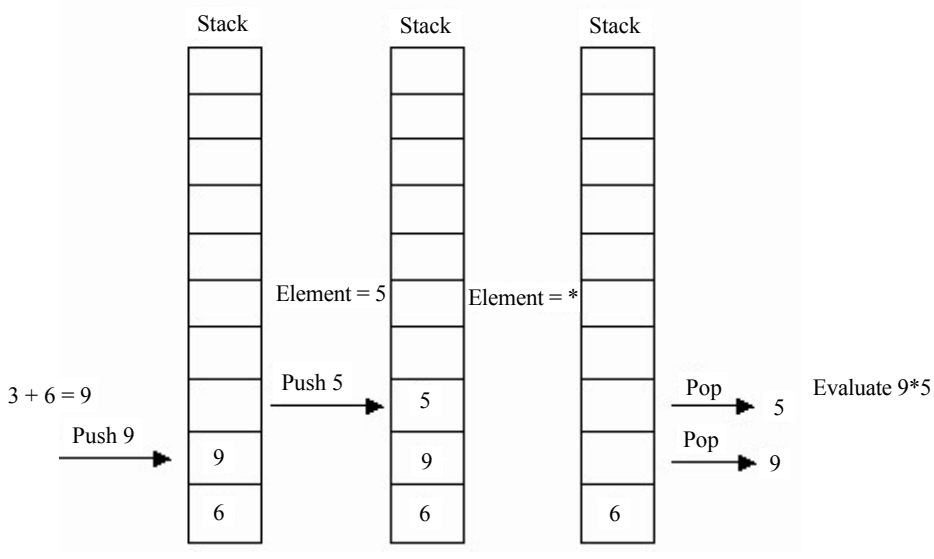


Fig. 2.11 Popping 9 and 5 from the Stack

8. Expression 9*5 is evaluated and the result, that is 45, is pushed to the back of the stack.
9. Next element in the postfix expression is 9, which is pushed onto the stack.
10. Next element is the operator /. Therefore, the two operands from the top of the stack, that is 9 and 45, are popped from the stack and the operation 45/9 is performed. Result 5 is again pushed to the stack.
11. ext element in the expression is –. Hence, 5 and 6 are popped from the stack and the operation 6 – 5 is performed. The resulting value, that is 1, is pushed to the stack (**Figure 2.12**).

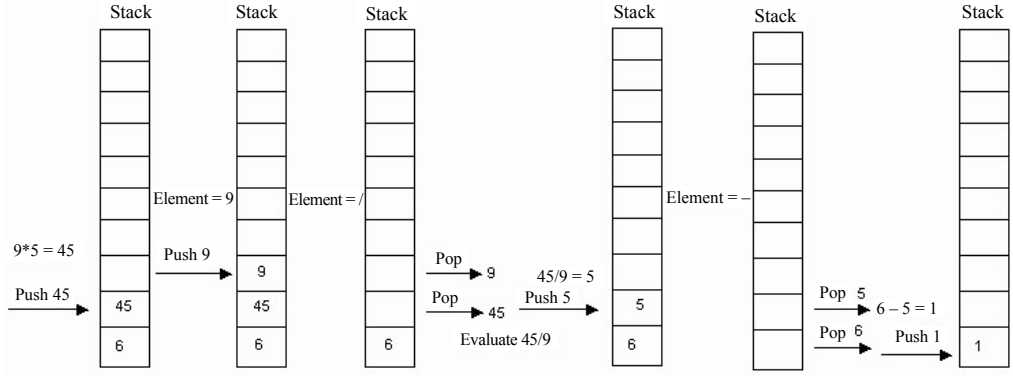


Fig. 2.12 Final State of Stack with the Result

12. There are no more elements to be processed in the expression. Element on top of the stack is popped, which is the result of the evaluation of the postfix expression. Thus, the result of the expression is 1.

Notes

Notes

The step-wise evaluation of the expression 636 + 5*9/- is shown in **Table 2.3.**

Table 2.3 Evaluation of the Postfix Expression

Element	Action Performed	Stack Status
6	Push to stack	6
3	Push to stack	6 3
6	Push to stack	6 3 6
+	Pop 6	6 3
	Pop 3	6
	Evaluate 3 + 6 = 9	6
	Push 9 to stack	6 9
5	Push to stack	6 9 5
*	Pop 5	6 9
	Pop 9	6
	Evaluate 9*5 = 45	6
	Push 45 to stack	6 45
9	Push to stack	6 45 9
/	Pop 9	6 45
	Pop 45	6
	Evaluate 45/9 = 5	6
	Push 5 to stack	6 5
-	Pop 5	6
	Pop 6	EMPTY
	Evaluate 6 - 5 = 1 EMPTY	
	Push 1 to stack	1
	Pop VALUE = 1	EMPTY

Multi-Stacks

So far, programmes containing a single stack have been discussed in the unit. If two or more stacks are needed in a programme, then it can be accomplished in two ways. One way is to have a separate array for each stack in the programme. This approach has a disadvantage—if one stack needs to store larger number of elements than the specified size and the other stack has lesser number of elements— it is not possible to store the elements of first stack in the second stack, in spite of the vacant space. This problem can be solved by another way, that is, by having a single array of sufficient size to hold two or more stacks. The two stacks can be represented efficiently in the same array, provided, one stack grows from left to right and other grows from right to left. Also, the memory is utilized more efficiently in this case.

For example, consider an array stack[MAX] to hold two stacks, stack1 and stack2. Two top variables Top1 and Top2 are required to represent the top of the two stacks. Initially, to represent the empty stacks, Top1 is set as '–1' and Top2 is set as MAX. In this case, the condition of overflow occurs when the combined size of

both the stacks exceed MAX. For this, variable count is used that keeps track of the number of elements stored in the array. Initially, count is set to '0'. Overflow occurs when the value of count exceeds the value of MAX and an attempt is made to insert a new element.

To PUSH an element in stack1, Top1 is incremented by 1 and the element is inserted in that position. On the other hand, to PUSH an element in stack2, Top2 is decremented by 1 and the element is inserted in that position. To POP an element from stack1, the element at the position indicated by Top1 is assigned to a local variable and then Top1 is decremented by 1. On the other hand, to POP an element from stack2, the element at the position indicated by Top2 is assigned to a local variable and then Top2 is incremented by 1. **Figure 2.13** shows an array of size MAX to hold two stacks stack1 and stack2, where stack1 grows from left to right and stack2 grows from right to left.

Notes

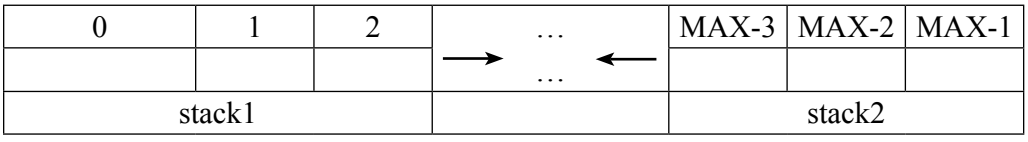


Fig. 2.13 Representing Two Stacks by an Array of Size MAX

To represent two stacks in the same array, the following structure called multi-stack needs to be defined in C language:

```
struct multistack
{
int item[MAX];
int Top1, Top2;
int count;
int sno; /*sno indicates the stack number (1 or
2)*/
};
```

Algorithm 5: PUSH Operation on Multi-Stack Stack

```
push(s, element) //s is a pointer to multi-stack
1.  If (s->count == MAX)
       Print "Overflow: Stack is full!" and go to step 4
    End If
2.  If(s->sno == 1) //if element is to be inserted in
    stack1
      Set s->Top1 = s->Top1 + 1
       Set s->item[s->Top1] = element
       Print "Value is pushed onto stack1..."
       Set s->count = s->count + 1
    End If
3.  If(s->sno == 2)
       Set s->Top2 = s->Top2 - 1
```

```
            Set s->item[s->Top2] = element
            Print "Value is pushed onto stack2..."
            Set s->count = s->count + 1
      End If
   4. End
```

Notes

Algorithm 6: POP Operation on Multi-Stack

```
    Algorithm 3.8 POP Operation on Multi-Stack
    pop(s)      //s is a pointer to multi-stack
   1. If(s->sno == 1)
         If (s->Top1 == -1)
               Print "Underflow! Stack1 is empty"
               Return 0 and go to step 4
     Else
         Set popped = s->item[s->Top1]
         Set s->Top1 = s->Top1 - 1
         Set s->count = s->count - 1
        End If
    End If
   2. If(s->sno == 2)
         If (s->Top2 == MAX)
               Print "Underflow! Stack2 is empty"
               Return 0 and go to step 4
     Else
               Set popped = s->item[s->Top2]
               Set s->Top2 = s->Top2 + 1
               Set s->count = s->count - 1
         End If
    End If
   3. Return popped
   4. End
```

Example 2: A programme to implement multi-stacks using a single array is as follows:

```
    #include<stdio.h>
#include<conio.h>
#define MAX 10
#define True 1
#define False 0
```

```
typedef struct multistack
{
    int item[MAX];
    int Top1, Top2;
    int count;
int sno;   /*sno is the stack number (1 or 2)*/
}mstk;
/*Function prototypes*/
void createstack (mstk *);
void push (mstk *, int);
int pop (mstk *);
int isempty (mstk *);
int isfull (mstk *);
void main ()
{
    int choice;
    int value;
    mstk s;
    createstack (&s);
    do{
        clrscr ();
        printf ("\n\tMain Menu");
        printf ("\n1. Push");
        printf ("\n2. Pop");
        printf ("\n3. Exit\n");
        printf ("\nEnter your choice: ");
        scanf ("%d", &choice);
        switch (choice)
        {
           case 1: printf ("\nEnter the stack number
           (1 or 2):");
              scanf ("%d", &s.sno);
              printf ("\nEnter the value to be
              inserted:");
              scanf ("%d", &value);
              push (&s, value);
              getch ();
              break;
```

Notes

Notes

```
        case 2: printf ("\nEnter the stack number
                (1 or 2):");
           scanf("%d", &s.sno);
           value = pop(&s);
           if (value == 0)
           {
              if (s.sno == 1)
                   printf  ("\nUnderflow:  Stack1
                   is empty!");
         else if (s.sno == 2)
            printf ("\nUnderflow: Stack2 is empty!");
            }
            else
              printf ("\nPopped element is: %d", value);
            getch ();
            break;
            case 3: exit ();
            default: printf ("\nInvalid choice!");
         }
     }while (1);
}
/*Function definitions*/
void createstack (mstk *s)
{
     s->Top1= -1;
     s->Top2 = MAX;
     s->count = 0;
}
void push (mstk *s, int item)
{
     if (isfull (s))
     {
         printf ("\nOverflow: Stack is full!");
         return;
}
if (s->sno == 1)
{
```

```
    s->Top1++;
    s->item [s->Top1] = item;
    printf ("\nValue is pushed onto stack1...");
        s->count++;
}
if (s->sno == 2)
{
        s->Top2-;
        s->item[s->Top2]=item;
        printf("\nValue is pushed onto stack2...");
        s->count++;
    }
}
int pop (mstk *s)
{
    int popped;
    if (is empty(s))
    return 0;
    if (s->sno == 1)
    {
        popped = s->item [s->Top1]; s->Top1-;
        s->count-;
    }
    if (s->sno == 2)
    {
        popped = s->item [s->Top2];
        s->Top2++;
        s->count-;
    }
    return popped;
}
int is empty (mstk *s)
{
    int r;
    if (s->sno == 1)
    {
        if (s->Top1 == -1)
            r = True;
        else
```

Notes

Notes

```
            r = False;
        }
        if (s->sno == 2)
        {
            if (s->Top2 == MAX)
                r = True;
            else
                r = False;
        }
        return r;
    }
    int is full (mstk *s)
    {
        if (s->count == MAX)
        return True;
        else return False;
    }
```

The Output of the Programme is as follows:

Main Menu

1. Push
2. Pop
3. Exit

Enter your choice: 1

Enter the stack number (1 or 2): 1

Enter the value to be inserted: 34

Value is pushed onto stack1...

Main Menu

1. Push
2. Pop
3. Exit

Enter your choice: 1

Enter the stack number (1 or 2): 2

Enter the value to be inserted: 45

Value is pushed onto stack2...

Main Menu

1. Push
2. Pop
3. Exit

Enter your choice: 1

Enter the stack number (1 or 2):

1 Enter the value to be inserted: 23

Value is pushed onto stack1...

Main Menu

1. Push
2. Pop
3. Exit

Enter your choice: 2

Enter the stack number (1 or 2): 1

Popped element is: 23

Main Menu

1. Push
2. Pop
3. Exit

Enter your choice: 2

Enter the stack number (1 or 2): 2

Popped element is: 45

Main Menu

1. Push
2. Pop
3. Exit

Enter your choice: 2

Enter the stack number (1 or 2): 1

Popped element is: 34

Main Menu

1. Push
2. Pop
3. Exit

Enter your choice: 2

Enter the stack number (1 or 2): 1

Underflow: Stack1 is empty!

Main Menu

1. Push
2. Pop
3. Exit

Enter your choice: 2

Notes

```
Enter the stack number (1 or 2): 2
Underflow: Stack2 is empty!
        Main Menu
1. Push
2. Pop
3. Exit
Enter your choice: 3
```

Notes

2.6 Queues

A **queue** is a linear data structure in which a new element is inserted at one end and an element is deleted from the other end. The end of the queue from which the element is deleted is known as the **front** and the end at which a new element is added is known as the rear. **Figure 2.14** shows a queue.

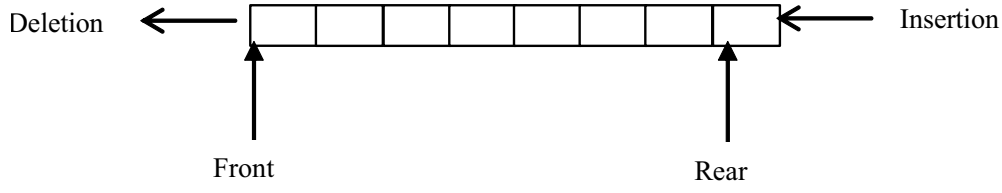


Fig. 2.14 Queue

The following are the basic operations that can be performed on queues:

- **Insert Operation:** To insert an element at the rear of the queue
- **Delete Operation:** To delete an element from the front of the queue

Before inserting a new element in the queue, it is necessary to check whether there is space for the new element. If no space is available, the queue is said to be in the condition of overflow. Similarly, before deleting an element from the queue, it is necessary to check whether there is an element in the queue. If there is no element in the queue, the queue is said to be in the condition of underflow.

2.7 Representation of Queues

Like stacks, queues can be represented in the memory by using an array or a singly linked list. In this section, we will discuss how a queue can be implemented using an array.

Array Implementation of a Queue

When a queue is implemented as an array, all the characteristics of an array are applicable to the queue. Since an array is a static data structure, the array representation of a queue requires the maximum size of the queue to be predetermined and fixed. As we know that a queue keeps on changing as elements are inserted or deleted, the maximum size should be large enough for a queue to expand or shrink.

The representation of a queue as an array needs an array to hold the elements of the queue and two variables rear and front to keep track of the rear and the front ends of the queue, respectively. Initially, the value of rear and front is set to –1 to indicate an empty queue. Before we insert a new element in the queue, it is necessary to test the condition of overflow. A queue is in a condition of overflow (full) when rear is equal to MAX–1, where MAX is the maximum size of the array. If the queue is not full, the insert operation can be performed. To insert an element in the queue, rear is incremented by one and the element is inserted at that position.

Notes

Similarly, before we delete an element from a queue, it is necessary to test the condition of underflow. A queue is in the condition of underflow (empty) when the value of front is –1. If a queue is not empty, the delete operation can be performed. To delete an element from a queue, the element referred by front is assigned to a local variable and then front is incremented by one.

The total number of elements in a queue at a given point of time can be calculated from the values of rear and front given as follows:

Number of elements = rear – front + 1

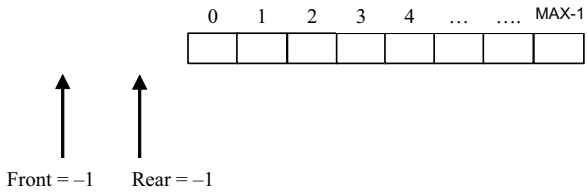
To understand the implementation of a queue as an array in detail, consider a queue stored in the memory as an array named queue that has MAX as its maximum number of elements. rear and front store the indices of the rear and front elements of queue. Initially, rear and front are set to –1 to indicate an empty queue (refer **Figure 2.15(a)**).

Whenever a new element has to be inserted in a queue, rear is incremented by one and the element is stored at queue[rear]. Suppose an element 9 is to be inserted in the queue. In this case, the rear is incremented from –1 to 0 and the element is stored at queue[0]. Since it is the first element to be inserted, front is also incremented by one to make it to refer to the first element of the queue (**Figure 2.15(b)**). For subsequent insertions, the value of rear is incremented by one and the element is stored at queue[rear]. However, front remains unchanged (**Figure 2.15(c)**). Observe that the front and rear elements of the queue are the first and last elements of the list, respectively.

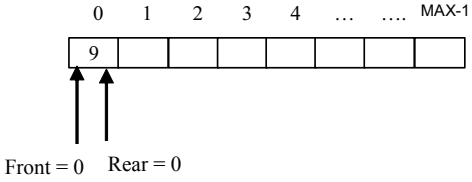
Whenever, an element is to be deleted from a queue, front is incremented by one. Suppose that an element is to be deleted from queue. Then, here it must be 9. It is because the deletion is always made at the front end of a queue. Deletion of the first element results in the queue as shown in **Figure 2.15(d).** Similarly, deletion of the second element results in the queue as shown in Figure **2.15(e).** Observe that after deleting the second element from the queue, the values of rear and front are equal. Here, it is apparent that when values of front and rear are equal other than –1, there is only one element in the queue. When this only element of the queue is deleted, both rear and front are again made equal to –1 to indicate an empty queue.

Further, suppose that some more elements are inserted and rear reaches the maximum size of the array (**Figure 2.15(f)**). This means that the queue is full and no more elements can be inserted in it even though the space is vacant on the left of the front.

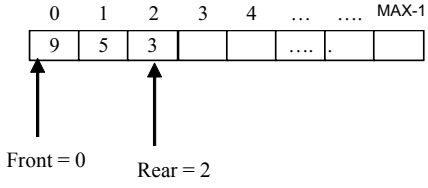
Notes



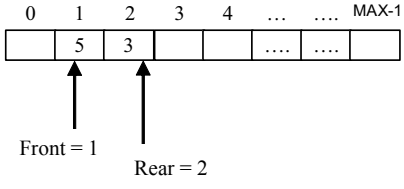
(*a*) *An Empty Queue*



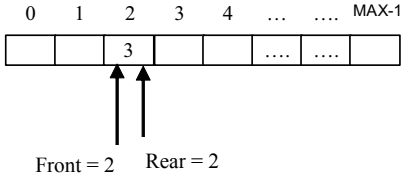
(*b*) *Queue after Inserting the First Element*



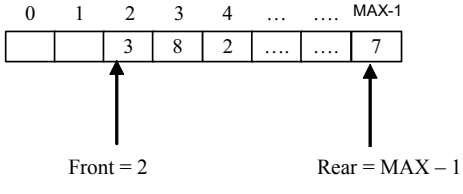
(*c*) *Queue after Inserting a few Elements*



(*d*) *Queue after Deleting the First Element*



(*e*) *Queue after Deleting the Second Element*



(*f*) *Queue having Vacant Space though Rear = MAX – 1*

Fig. 2.15 Various States of a Queue after the Insert and Delete Operations

To implement a queue as an array in the C language, the following structure named queue is used:

```
struct queue
{
    int item[MAX];
    int front;
    int rear;
};
```

Notes

Algorithm 1: Insert Operation on a Queue

```
qinsert (q, val))     //q is a pointer to structure
                      type   queue   and   val   is
                      the value to be   //inserted
1.  If q->rear = MAX-1 //check if queue is full
      Print "Overflow: queue is full!" and go to step 5
    End If
2.  If q->front = -1   //check if queue is empty
      Set q->front = 0 //  make  front  to  refer  to
first element   End If
3.  Set q->rear = q->rear + 1//increment rear by one
4.  Set q->item [q->rear] = val   //insert val
5.  End
```

Algorithm 2: Delete Operation on a Queue

```
qdelete (q)
1.  If q->front = -1   //check if queue is empty
      Print "Underflow: queue is empty!"
      Return 0 and go to step 5
    End If
2.  Set del_val = q->item [q->front]   //del_val   is
    the value to be deleted
3.  If q->front = q->rear        //check if there is
    only one element
      Set q->front = q->rear = -1
    Else
      Set q->front = q->front + 1     //increment
front by one
    End If
4.  Return del_val
5.  End
```

Linked Implementation of a Queue

A queue implemented as a linked list is known as a **linked queue.** A linked queue is represented using two pointer variables front and rear that point to the first and the last node of the queue, respectively. Initially, rear and front are set to NULL to indicate an empty queue.

To understand the implementation of a linked queue, consider a linked queue, say queue. The info and next fields of each node represent the element of the queue and a pointer to the next element in the queue, respectively. Whenever a new element

is to be inserted in the queue, a new node nptr is created and the element is inserted into the node. If it is the first element being inserted in the queue, both front and rear are modified to point to this new node. On the other hand, in subsequent insertions, only rear is modified to point to the new node; front remains unchanged.

Notes

Whenever an element is deleted from the queue, a temporary pointer is created, which is made to point to the node pointed to by front. Then front is modified to point to the next node in the queue, and the temporary node is deleted from the memory. **Figure 2.16** shows the various states of a queue after the insert and delete operations.

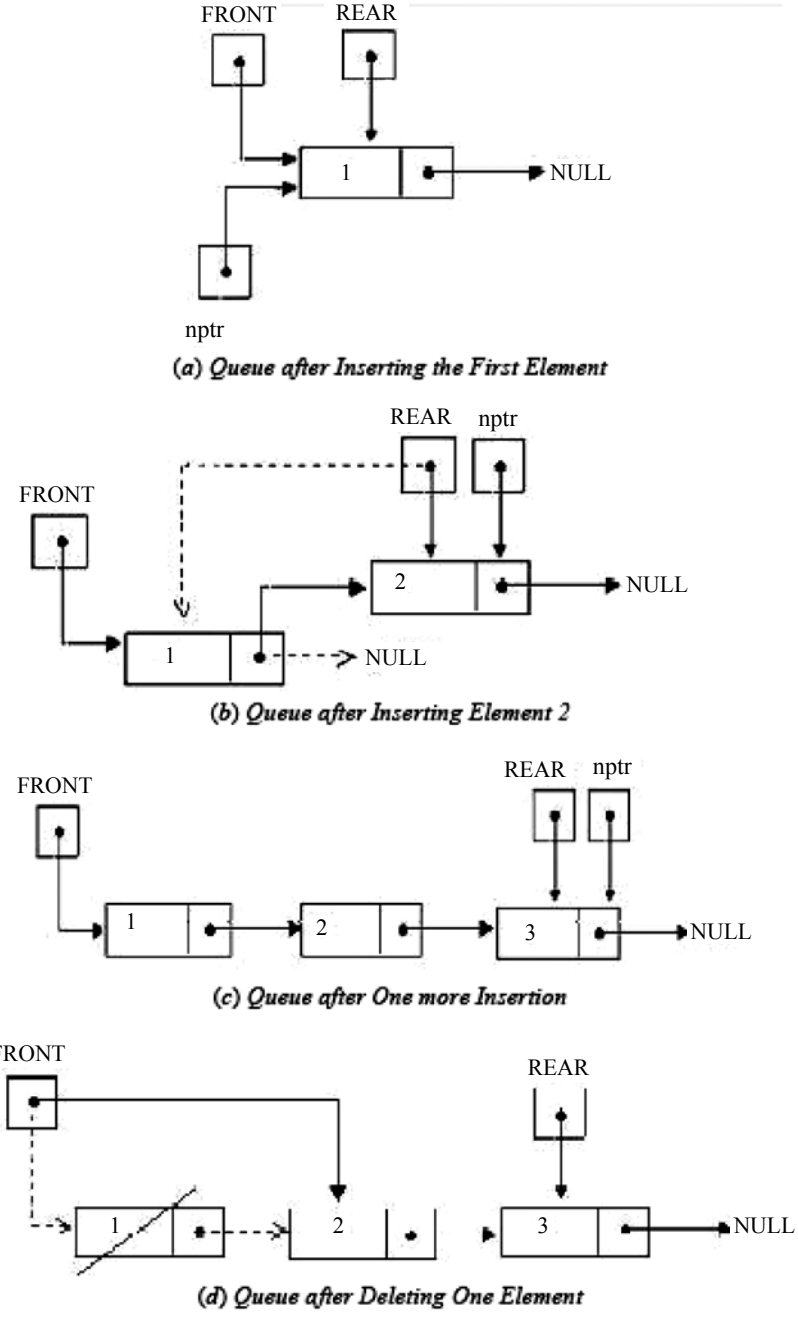


Fig. 2.16 Various states of a Linked Queue after the Insert and Delete Operations

Note: Since the memory is allocated dynamically, a linked queue reaches the overflow condition when no more free memory space is available to be dynamically allocated.

Algorithm 3: Insert Operation on a Linked Queue

```
qinsert(q, val)        //val is the value to be inserted
1.  Allocate memory for nptr //nptr is a pointer to
    the new node to be inserted
2.  If nptr = NULL     // checking for queue overflow
            Print "Overflow: Memory not allocated!"
            and go to step 6
      End If
3.  Set nptr->info = val
4.  Set nptr->next = NULL
5.  If front = NULL    //check if queue is empty
  Set q->rear = q->front = nptr  //rear and front
  are made to point to new
                            //node
     Else
           Set q->rear->next = nptr
          Set q->rear = nptr     //rear is made to
          point to new node
     End If
6.  End
```

Algorithm 4: Delete Operation on a Linked Queue

```
qdelete(q)
1.  If front = NULL
      Print "Underflow: queue is empty!"
      Return 0 and go to step 7
      End if
2.  Set del_val = q->front->info //del_val  is  the
    element pointed by the front
3.  Set temp = q->front//temp is the temporary pointer
    to front
4.  If q->front = q->rear   //checking  if  there  is
    one element in the queue
      Set q->front = q->rear = NULL
       Else
       Set q->front = q->front->next   //making front
       point to next node
      End If
5.  De-allocate temp //de-allocating memory
6.  Return del_val
7.  End
```

2.8 Circular Queue and Deque

As discussed earlier, in the case of a queue represented as an array, once the value of the rear reaches the maximum size of the queue, no more elements can

Notes

Notes

be inserted. However, there may be the possibility that the space on the left of the front index is vacant. Hence, in spite of space on the left of front being empty, the queue is considered full. This wastage of space in the array implementation of a queue can be avoided by shifting the elements to the beginning of the array if space is available. In order to do this, the values of the rear and front indices have to be changed accordingly. However, this is a complex process and difficult to implement. An alternative solution to this problem is to implement a queue as a circular queue.

The array implementation of a circular queue is similar to the array implementation of the queue. The only difference is that as soon as the rear index of the queue reaches the maximum size of the array, rear is reset to the beginning of the queue, provided it is free. The circular queue is full only when all the locations in the array are occupied. A circular queue is shown in **Figure 2.17.**

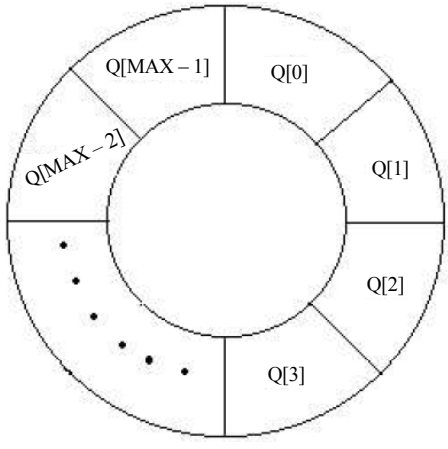


Fig. 2.17 Circular Queue

Note: A circular queue is generally implemented as an array though it can also be implemented as a circular linked list.

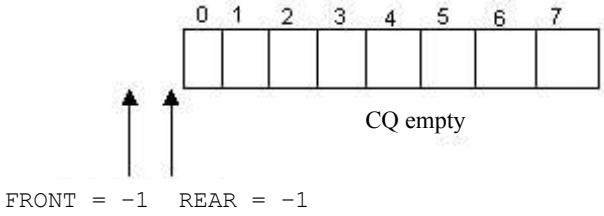
To understand the operations on a circular queue, consider a circular queue represented in the memory by the array Cqueue [MAX]. Rear and front are used to store the indices of the rear and front elements of Cqueue, respectively. Initially, both rear and front are set to NULL to indicate an empty queue.

Whenever an element is to be inserted in a circular queue, rear is incremented by one. However, if the value of the rear index is MAX-1, instead of incrementing rear, it is reset to the first index of the array if space is available in the beginning. Hence, if any location to the left of the front index is empty, the elements can be added to the queue at an index starting from 0. A queue is considered full in the following cases:

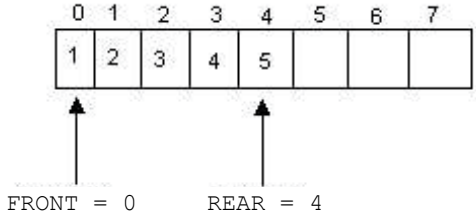
- When the value of rear equals the maximum size of the array and front is at the beginning of the array
- When the value of front is one more than the value of rear

Whenever an element is to be deleted from the queue, front is incremented by one. However, if the value of front is MAX-1, it is reset to the 0th position in the array. When the value of front equals the value of rear (other than –1), it indicates that there is only one element in the queue. On deleting the last element, both rear

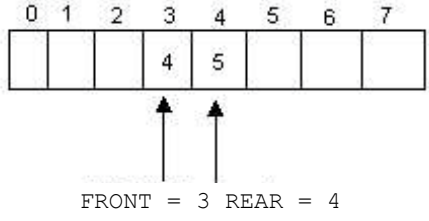
and front are reset to NULL to indicate an empty queue. **Figure 2.18** shows the various states of a queue after some insert and delete operations.



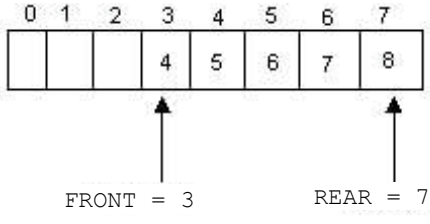
(*a*) *An Empty Queue*



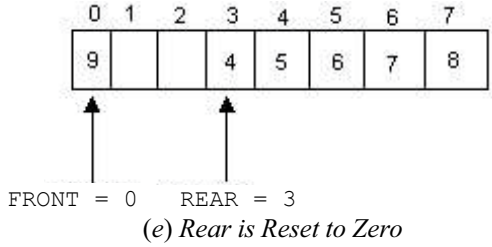
(*b*) *Queue after Inserting a few Elements*



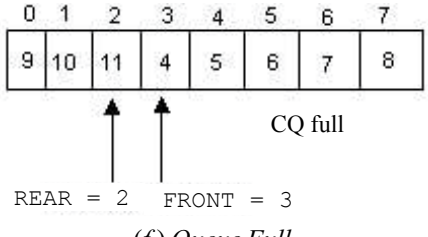
(*c*) *Queue after Deleting a few Elements*



(*d*) *Queue when* `Rear = MAX-1`



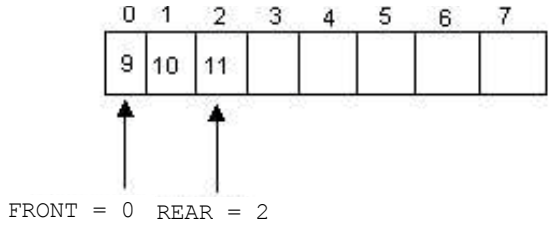
(*e*) *Rear is Reset to Zero*



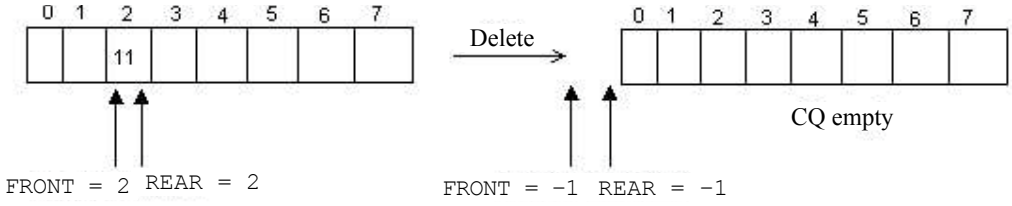
(*f*) *Queue Full*

Notes

Notes



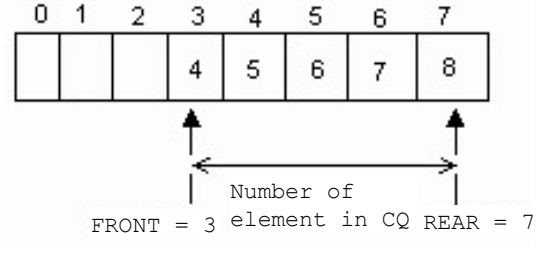
(*g*) *Front is Reset to Zero*



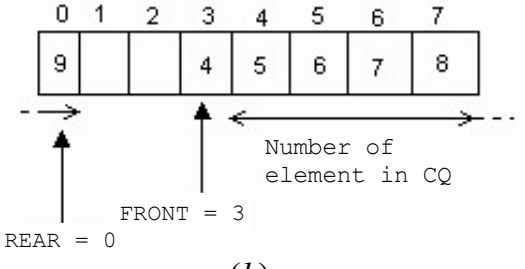
(*h*) *Queue after deletion of only element*

Fig. 2.18 Various States of a Circular queue after the Insert and Delete Operations

The total number of elements in a circular queue at any point of time can be calculated from the current values of the rear and front indices of the queue. In case, front < rear, the total number of elements = rear – front + 1. For instance, in **Figure 2.19**(a), front = 3 and rear = 7. Hence, the total number of elements in C queue at this point of time is 7 – 3 + 1 = 5. In case, front > rear, the total number of elements = Max + (rear – front) + 1. For instance, in **Figure 2.19** (b), front = 3 and rear = 0. Hence, the total number of elements in C queue is 8 + (0 – 3) + 1.



(*a*)



(*b*)

Fig. 2.19 Number of Elements in a Circular Queue

Algorithm 5: Insert Operation on a Circular Queue

```
qinsert(q, val)
1.  If ((q->rear = MAX-1 AND q->front = 0) OR (q->rear
    + 1 = q->front))
          Print "Overflow: queue is full!" and go to
          step 5
     End If //check if circular queue is full
2.  If q->rear = MAX-1 // check if rear is MAX-1
          Set q->rear = 0
      Else
          Set q->rear = q->rear + 1  //increment
          rear by one
      End If
3.  Set q->C queue[q->rear] = val //val is the value
    to be inserted in the queue
4.  If q->front = -1 //check if queue is empty
          Set q->front = 0
      End If
5.  End
```

Notes

Algorithm 6: Delete Operation on a Circular Queue

```
qdelete(q)
1.  If q->front = -1
          Print "Underflow: Queue is empty!"
          Return 0 and go to step 5
      End If
2.  Set del_val = q->C queue[q->front] //del_val  is
    the value to be deleted
3.  If q->front = q->rear   // check if there is one
    element in the queue
         Set q->front = q->rear = -1
      Else
          If q->front = MAX-1
            Set q->front = 0
          Else
            Set q->front = q->front +1
          End If
      End If
4.  Return del_val
5.  End
```

2.9 Deque (Double-Ended Queue)

A deque (short form of double-ended queue) is a linear list in which elements can be inserted or deleted at either end but not in the middle. That is, elements can be inserted/deleted to/from the rear end or the front end. **Figure 2.20** shows the representation of a deque.

Notes

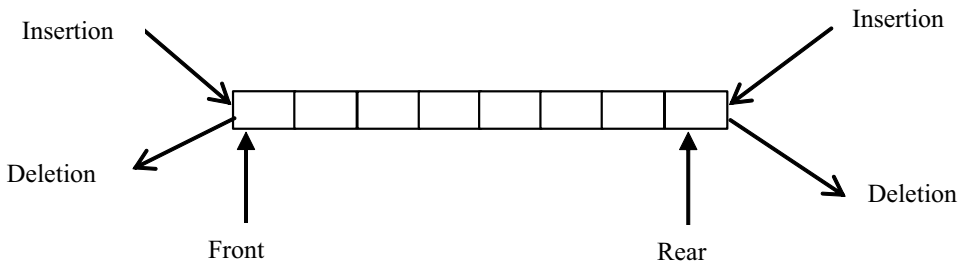


Fig. 2.20 A Deque

Like a queue, a deque can be represented as an array or a singly linked list. Here, we will discuss the array implementation of a deque.

Algorithm 1: Insert Operation in the Beginning of a Deque

```
qinsert_beg(q, val)
1.  If (q->rear = MAX-1 And q->front = 0)
            Print "Overflow: queue is full!" and go to
            step 4
      End If
2.  If q->front = -1
           Set q->front = q->rear = 0
           Set q->De queue[q->front] = val
      End If
3.  If q->rear != MAX -1 //check if last position is
    occupied
          Set num_item = q->rear - q->front + 1//total
          number of elements
          Set i = q->rear +1
          Set j = 1
          While j <= num_item
           Set q->De queue[i] = q->D queue[i - 1]
                 //shift elements one space  to the
                 //right
           Set i = i - 1
           Set j = j + 1
      End While
      Set q->De queue[i] = val
      Set q->front = i
      Set q->rear = q->rear + 1
  Else
      Set q->front = q->front - 1
      Set q->De queue[q->front] = val
  End If
4.  End
```

Algorithm 2: Insert Operation at the End of a Deque

```
qinsert_end (q, val)
1.  If (q->rear = MAX-1 And q->front = 0)   //check
    if queue is full
      Print "Overflow: queue is full!" and go to step 4
    End If
2.  If q->front = -1   //check if queue is empty
      Set q->front = q->rear = 0
      Set q->De queue[q->front] = val and go to step 4
    End If
3.  If q->rear = MAX-1 // check if last position is
    occupied
      Set i = q->front -1
      While i < q->rear //shift elements one place to
the left of queue
          Set q->De queue[i] = q->De queue[i+1]
           Set i = i +1
      End While
      Set q->De queue[q->rear] = val
      Set q->front = q->front - 1
    Else
      Set q->rear = q->rear +1
      Set q->De queue[q->rear] = val
    End If
4.  End
```

Algorithm 3: Delete Operation in the Beginning of a Deque

```
qdelete_beg (q)
1.  If q->front = -1
      Print "Underflow: queue is empty!"
      Return 0 and go to step 5
    End If
2.  Set del_val = q->De queue[q->front]
3.  If q->front = q->rear
      Set q->front = q->rear = -1
    Else
      Set q->front = q->front + 1
    End If
4.  Return del_val
5.  End
```

Algorithm 4: Delete Operation at the End of a Deque

```
qdelete_end(q)
1.  If q->front = -1
```

Notes

Notes

```
        Print "Underflow: De queue is empty!"
        Return 0 and go to step 5
    End If
2.  Set del_val = q->De queue[q->rear]
3.  If q->front = q->rear
        Set q->front = q->rear = -1
    Else
        Set q->rear = q->rear - 1
        If q->rear = -1
            Set q->front = -1
        End If
    End If
4.  Return del_val
5.  End
```

The two variations of a deque are as follows:

- **Input Restricted Deque:** It allows insertion of elements only at one end but deletion can be done at both ends **(Figure 2.21)**.

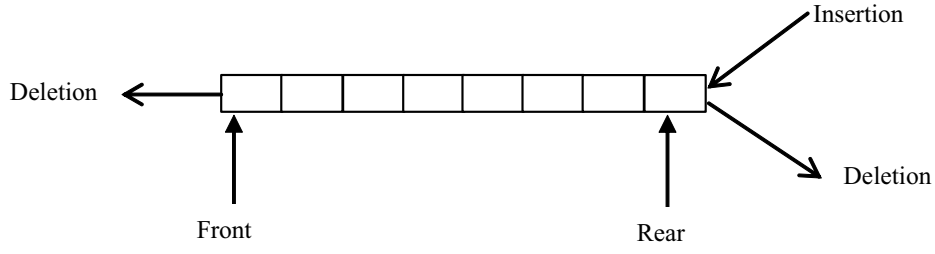


Fig. 2.21 Input Restricted Deque

- **Output Restricted Deque:** It allows deletion of elements only at one end but insertion can be done at both ends **(Figure 2.22)**.

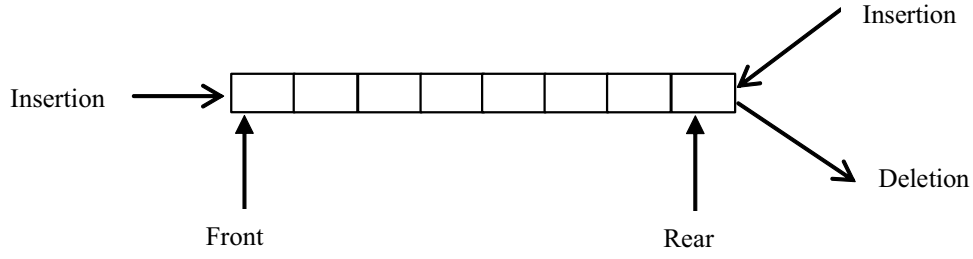


Fig. 2.22 An Output Restricted Deque

The implementation of both these queues is similar to the implementation of a deque. The only difference is that in an input restricted queue, the function for insertion in the beginning is not needed, whereas in an output restricted queue, the function for deletion in the beginning is not needed.

2.10 Priority Queue

A priority queue is a type of queue in which each element is assigned a priority and the elements are added or removed according to that priority. While implementing a priority queue, the following two rules are applied:

- The element with higher priority is processed before any element of lower priority.
- The elements with the same priority are processed according to the order in which they were added to the queue.

A priority queue can be represented in many ways. Here, we will discuss the implementation of a priority queue using multiple queues.

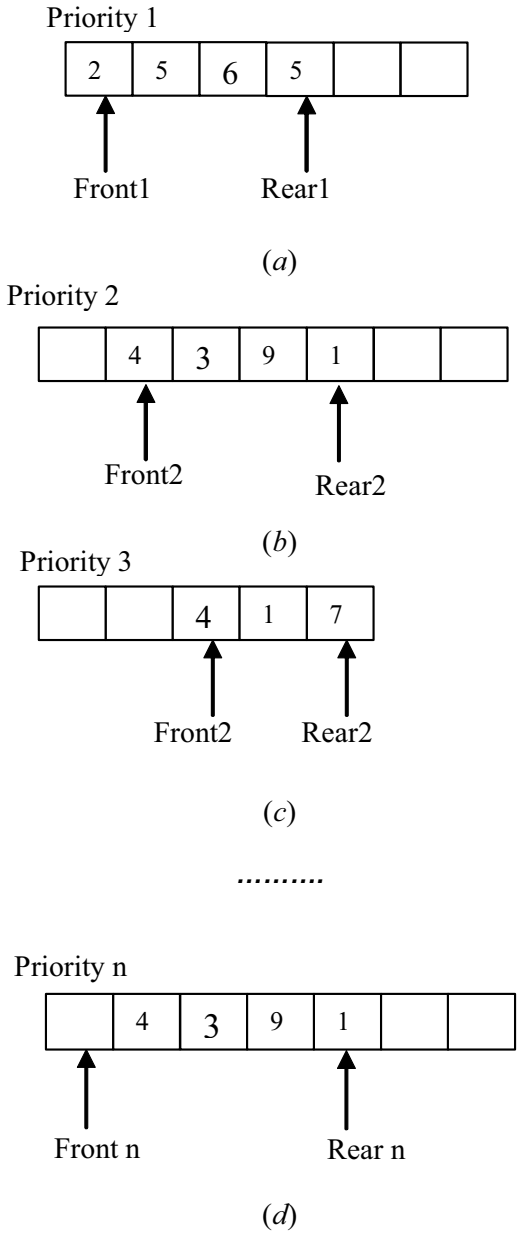


Fig. 2.23 Multiple Queues according to the Priority

Notes

In the multiple queue representation of a priority queue, a separate queue for each priority is maintained. Each queue is implemented as a circular array and has its own two variables, front and rear (**Figure 2.23**). The element with the given priority number is inserted in the corresponding queue. Similarly, whenever an element is to be deleted from the queue, it must be the element from the highest priority queue. Note that lower priority number indicates higher priority.

If the size of each queue is the same, then instead of multiple one-dimensional arrays, a single two-dimensional array can be used where the row number shows the priority and the column number shows the position of the element within the queue. In addition, two arrays to keep track of the front and rear positions of each queue corresponding to each row are maintained (**Figure 2.24**).

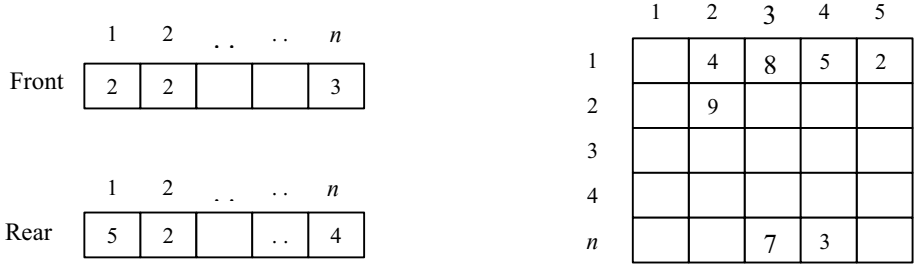


Fig. 2.24 Priority Queue as a Two-Dimensional Array

Algorithm 1: Insert Operation in a Priority Queue

```
qinsert(q, val, prno) //prno is the priority of val
1.  If (q->rear [prno] = MAX-1 And q->front [prno] =
    0) OR (q->rear [prno]+1 =  q->front [prno])
    Print "Overflow: queue full!" and go to step 5
 End If
2.  If q->rear [prno-1] = MAX-1
       Set q->rear [prno-1] = 0
  Else
       Set q->rear [prno-1] = q->rear [prno-1] + 1
  End If
3.  Set q->C queue [prno-1] [q->rear [prno-1]] = val
 4. If q->front [prno-1] = -1
       Set q->front [prno-1] = 0
  End If
5.  End
```

Algorithm 2: Delete Operation in a Priority Queue

```
qdelete (q)
1.  Set flag = 0, i = 0
2.  While i <= MAX-1
```

```
            If NOT (q->front [prno]) = -1 //check if
            not empty
              Set flag = 1
             Set del_val = q->Cqueue [i][q->front[i]]
             If q->front[i] = q->rear [i]
                  Set q->front [i] = q->rear [i] = -1
                   Else If q->front [i] = MAX-1
                        Set q->front [i] = 0
                        Else
                            Set   q->front   [i]   =
                            q->front [i] + 1
                   End If
              End If
             Break//jump out of the while loop
        End If
        Set i = i +1
    End While
3.  If flag = 0
        Return 0 and go to step 4
    Else
        Return del_val
    End If
4.  End
```

2.11 Applications of Queues

There are numerous applications of queues in computer science. Various real-life applications such as railway ticket reservation and the banking system are implemented using queues. One of the most useful applications of a queue is in simulation. Another application of a queue is in the operating system, to implement various functions like CPU scheduling in a multiprogramming environment, device management (printer or disk), etc. Besides, there are several algorithms like level order traversal of binary tree, etc., that use queues to solve problems efficiently. This section discusses some of the applications of queues.

Simulation

Simulation is the process of modelling a real-life situation through a computer programme. Its main use is to study a real-life situation without actually making it occur. It is mainly used in areas like military operations, scientific research, etc., where it is expensive or dangerous to experiment with the real system. In simulation, corresponding to each object and action, there is a counterpart in the programme. The objects that are studied are represented as data and the actions are represented as operations on the data. By supplying different data, we can observe the result of the programme. If the simulation is accurate, the result of the programme represents the behaviour of the actual system accurately.

Notes

Notes

Consider a ticket reservation system having four counters. If a customer arrives at time t_a and a counter is free, then the customer will get the ticket immediately. However, it is not always possible that a counter is free. In that case, a new customer goes to the queue having fewer customers. Assume that the time required to issue the ticket is t. Then the total time spent by the customer is equal to the time t (time required to issue the ticket) plus the time spent waiting in line. The average time spent in the line by the customer can be computed by a programme simulating the customer action. This programme can be implemented using a queue, since while one customer is being serviced, the others are kept waiting.

Central Processing Unit (CPU) Scheduling in a Multiprogramming Environment

As we know, in a multiprogramming environment, multiple processes run concurrently to increase CPU utilization. All the processes that are residing in the memory and are ready to execute are kept in a list referred to as a ready queue. It is the job of the scheduling algorithm to select a process from the processes and allocate the CPU to it.

Let us consider a multiprogramming environment where the processes are classified into three different groups, namely system processes, interactive processes and batch processes. Some priority is associated with each group of processes. The system processes have the highest priority, whereas the batch processes have the least priority. To implement a multiprogramming environment, a multi-level queue scheduling algorithm is used. In this algorithm, the ready queue is partitioned into multiple queues (**Figure 2.25**). The processes are assigned to the respective queues. The higher priority processes are executed before the lower priority processes. For example, no batch process can run unless all the system processes and interactive processes are executed. If a batch process is running and a system process enters the queue, then batch process would be pre empted to execute this system process.

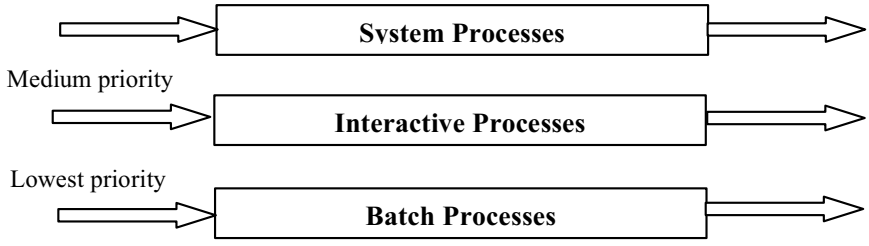


Fig. 2.25 Multi-Level queue Scheduling

In this algorithm, the processes of a lower priority may starve if the number of processes in a higher-priority queue is high. Starvation can be prevented by two ways. One way is to time-slice between the queues, that is, each queue gets a certain interval of time. Another way is using a multi-level feedback queue algorithm. In this algorithm, processes are not assigned permanently to a queue; instead, they are allowed to move between the queues. If a process uses too much CPU time, it is moved to lower priority. Similarly, a process that has been waiting for too long in a lower-priority queue is moved to the higher-priority queue. To implement multiple programming environments, a priority queue using multiple queues can be used.

Notes

Round Robin Algorithm

The Round Robin algorithm is one of the CPU scheduling algorithms designed for time-sharing systems. In this algorithm, the CPU is allocated to a process for a small time interval called **time quantum** (generally from 10 to 100 milliseconds). Whenever a new process enters, it is inserted at the end of the ready queue. The CPU scheduler picks the first process from the ready queue and processes it until the time quantum elapses. Then, the CPU switches to the next process in the queue and the first process is inserted at the end of the queue if it has not been finished. If the process is finished before the time quantum, the process itself releases the CPU voluntarily and the process gets deleted from the ready queue. This process continues until all the processes are finished. When a process is finished, it is deleted from the queue. To implement the Round Robin algorithm, a circular queue can be used.

Suppose there are *n* processes, such as $P_1, P_2, \ldots, P_n$ served by the CPU. Different processes require different execution time. Suppose, sequence of processes arrivals is arranged according to their subscripts, *i.e.*, P_1 comes first, then P_2. Therefore, P*i* comes after P_{i-1} where $1< i \leq n$. Round Robin algorithm first decides a small unit of time called **time quantum** or **time slice** represented by τ. A time quantum generally starts from 10 to 100 milliseconds. CPU starts services from P_1. Then, P_1 gets CPU for τ instant of time; afterwards CPU switches to process P_2 and so on. Now, during time-sharing, if a process finishes its execution before the finding of its time quantum, the process then simply releases the CPU and the next process waiting will get the CPU immediately. When CPU reaches the end of time quantum of P_n it returns to P_1 and the same process will be repeated. For an illustration, consider **Table 2.4** for the set of processes.

Table 2.4 Table for Process and Burst Time

Process	Burst Time
P_1	7
P_2	18
P_3	5

The total required CPU time keeps 30 units for burst time as summarized in **Table 2.4** and depicted in **Figure 2.26**.

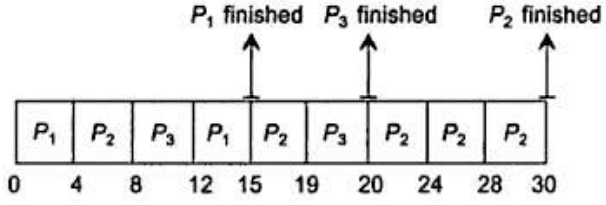


Fig. 2.26 Round Robin Scheduling

The advantage of Round Robin algorithm is in reducing the average turn-around time. The turn-around time of a process is the time of its completion, i.e., time of its arrival. Thus, Round Robin algorithm uses first come first served or FCFS strategy.

Notes

2.12 Linked List

A dynamic data structure is one in which the memory for elements is allocated dynamically during run-time. The successive elements of a dynamic data structure need not be stored in contiguous memory locations but they are still linked together by means of some linkages or references. Whenever a new element is inserted, the memory for the same is allocated dynamically and is linked to the data structure. The elements can be inserted as the long as memory is available. Thus, there is no upper limit on the number of elements in the data structure. Similarly, whenever an element is deleted from the data structure, memory is de-allocated so that it can be reused in the future. Linked list is an example of a dynamic data structure. It has been explained in this section.

Linked List

A linked list is a linear collection of homogeneous elements called **nodes**. Successive nodes of a linked list need not occupy adjacent memory locations. The linear order between nodes is maintained by means of pointers. In linked lists, insertion or deletion of nodes do not require shifting of existing nodes as in the case of arrays; they can be inserted or deleted merely by adjusting the pointers or links.

Depending on the number of pointers in a node or the purpose for which the pointers are maintained, a linked list can be classified into various types such as singly-linked list, circular-linked list and doubly-linked list. The unit will discuss these types in detail in the subsequent sections.

2.13 Singly-Linked Lists

A singly-linked list is also known as a **linear linked list.** In it, each node consists of two fields, viz. 'info' and 'next', as shown in **Figure 2.27**. The 'info' field contains the data and the 'next' field contains the address of memory location where the subsequent node is stored. The last node of the singly-linked list contains NULL in its 'next'field which indicates the end of the list.

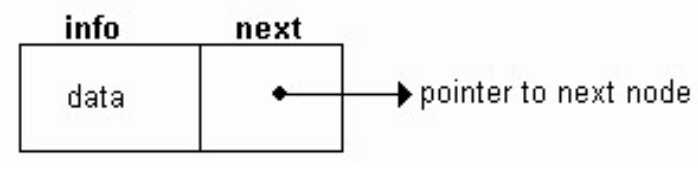


Fig. 2.27 Node of a Linked List

Note: The data stored in the 'info' field may be a single data item of any data type or a complete record representing a student, or an employee, or any other entity. In this unit, however, it is assumed that the 'info' field contains an integer data.

A linked list contains a list pointer variable 'Start' that stores the address of the first node of the list. In case, the 'Start' node contains NULL, the list is called **an empty list** or a **null list**. Since each node of the list contains only a single pointer pointing to the next node, not to the previous node—allowing traversing in only one direction—hence, it is also referred to as a one-way list. **Figure 2.27** shows a singly-linked list with four nodes.

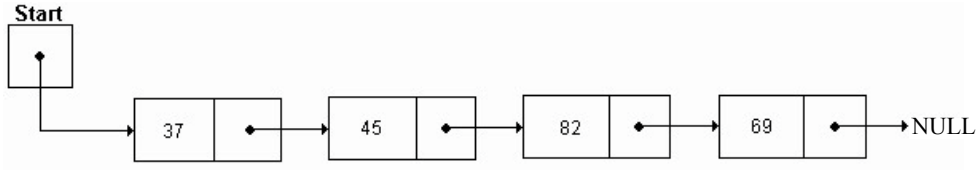


Fig. 2.28 Singly-Linked List with Four Nodes

Notes

Operations

A number of operations can be performed on singly-linked lists. These operations include traversing, searching, inserting and deleting nodes, reversing, sorting, and merging linked lists. Before implementing these operations, it is important to understand how the node of a linked list is created.

Creating a node means defining its structure, allocating memory to it, and its initialization. As discussed earlier, the node of a linked list consists of data and a pointer to the next node. To define a node containing an integer data and a pointer to next node in C language, a self-referential structure can be used whose definition is as follows:

```
typedef struct node
{
int info; /*to store integer type data*/
struct node *next; /*to store a pointer to next node*/
}Node;
Node *nptr; /*nptr is a pointer to node*/
```

After declaring a pointer nptr to new node, the memory needs to be allocated dynamically to it. If the memory is allocated successfully (means no overflow), the node is initialized. The info field is initialized with a valid value and the next field is initialized with NULL.

Algorithm 1: Creation of a Node

```
create_node()
1. Allocate memory for nptr //nptr is a pointer to
   new node
2. If nptr = NULL
   Print "Overflow: Memory not allocated!" and go to
   step 7  End If
3. Read item //item is the value to be inserted in
   the new node
4. Set nptr->info = item
5. Set nptr->next = NULL
6. Return nptr //returning pointer nptr
7. End
```

Now, the linked list can be formed by creating several nodes of type Node and inserting them either in the beginning or at the end or at a specified position in the list.

Notes

2.14 Circular Linked Lists

A linear linked list, in which the next field of the last node points back to the first node instead of containing NULL, is termed as a **circular linked list**. The main advantage of a circular linked list over a linear linked list is that by starting with any node in the list, its predecessor nodes can be reached. This is because when a circular linked list is traversed, starting with a particular node, the same node is reached at the end. **Figure 2.29** shows an example of a circular linked list.

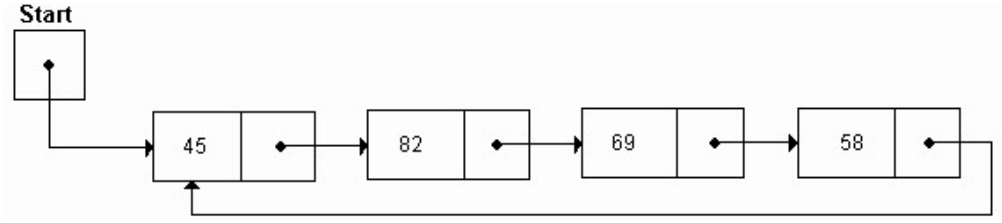


Fig. 2.29 Circular Linked List

All the operations that can be performed on linear linked lists can be easily performed on circular linked lists but with some modifications. Some of these operations have been discussed in this section.

Note: The process of creating a node of a circular linked list is same as that of linear linked list.

Traversing

A circular linked list can be traversed in the same way as a linear linked list, except the condition for checking the end of list. A circular linked list is traversed until a node in the list is reached at, which contains address of the first node in its next field rather than NULL as in case of a linear linked list.

Algorithm 2: Traversing a Circular Linked List

```
    display (Start)
1.  If Start = NULL
    Print "List is empty!!" and go to step 4
    End If
2.  Set temp = Start //initialising temp with start
3.  Do
    Print temp->info //displaying value of each node
    Set temp = temp->next
    While temp != Start
 4  End
```

2.15 Doubly-Linked Lists

In a singly-linked list, each node contains a pointer to the next node and it has no information about its previous node. Thus, one can traverse only in one direction, i.e., from beginning to end. However, sometimes it is required to traverse

in the backward direction, *i.e.*, from end to beginning. This can be implemented by maintaining an additional pointer in each node of the list that points to the previous node. Such type of a linked list is called **doubly-linked list**.

Each node of a doubly-linked list consists of three fields—prev, info, and next (**Figure 2.30**). The info field contains the data, the prev field contains the address of the previous node, and the next field contains the address of the next node.

Notes

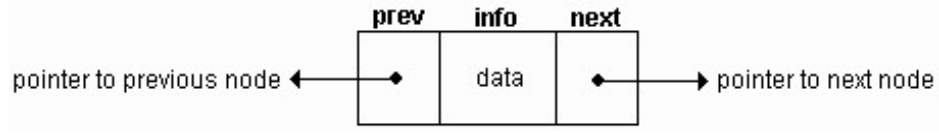


Fig. 2.30 Node of a Doubly-Linked List

Since a doubly-linked list allows traversing in both forward and backward directions, it is also referred to as a two-way list. **Figure 2.31** shows an example of a doubly-linked list having four nodes. It must be noted that the prev field of the first node and next field of the last node in a doubly-linked list points to NULL.

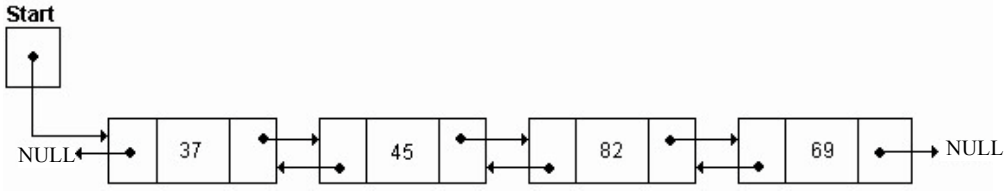


Fig. 2.31 Doubly-Linked List with Four Nodes

To define the node of a doubly-linked list in C language, the structure used to represent the node of a singly-linked list is extended to have an extra pointer which points to previous node. The structure of a node of a doubly-linked list is as follows:

```
typedef struct node
{
int info; /*to store integer type data*/ struct node
*next; /*to store a pointer to next node*/
struct node *prev; /*to store a pointer to previous
node*/
}Node;
Node *nptr; /*nptr is a pointer to node*/
```

When memory is allocated successfully to a node, *i.e.*, when there is no condition of overflow, the node is initialized. The info field is initialized with a valid value and the prev and next fields are initialized with NULL.

Algorithm 3: Creating a Node of a Doubly-Linked List

```
create_node ()
1. Allocate memory for nptr //nptr is a pointer to
   new node
2. If nptr = NULL
```

Notes

```
        Print "Overflow: Memory not allocated!" and go to
        step 8
    3.  Read item //item is the value stored in the node
    4.  Set nptr->info = item
    5.  Set nptr->next = NULL
    6.  Set nptr->prev = NULL
    7.  Return nptr
    8.  End
```

It must be brought to notice that all the operations that are performed on singly linked lists can also be performed on doubly-linked lists. In the subsequent sections, only insertion and deletion operations on doubly-linked lists have been discussed.

Insertion

Insertion in the Beginning

To insert a new node in the beginning of a doubly-linked list, a pointer, for example nptr to new node is created. The next field of the new node is made to point to the existing first node and prev field of the existing first node (that has become the second node now) is made to point to the new node. After that, Start is modified to point to the new node. **Figure 2.32** shows the insertion of node in the beginning of a doubly-linked list.

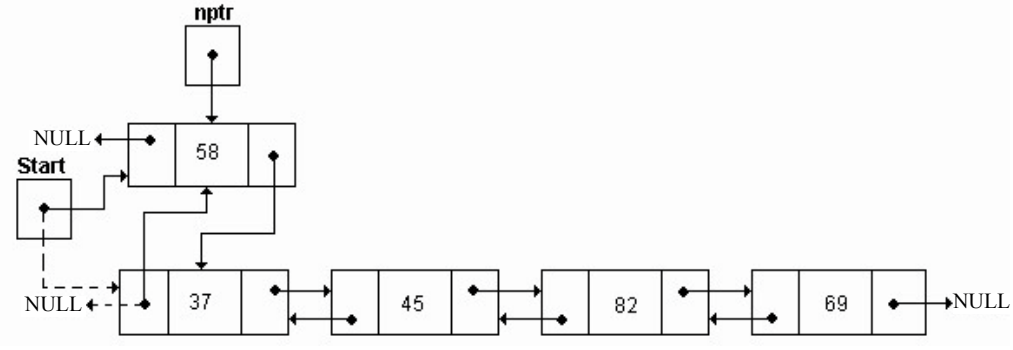


Fig. 2.32 Insertion in the Beginning

Algorithm 4: Insertion in the Beginning

```
    insert_beg(Start)
1.  Call create_node() //creating a new node pointed
    to by nptr
2.  If Start != NULL
    Set nptr->next = Start //inserting node in the
    beginning   Set Start->prev = nptr
    End If
3.  Set Start = nptr //making Start to point to new
    node
4.  End
```

2.16 Merging Lists and Header Linked List

Notes

Merging Lists

Merge lists or algorithms are a family of algorithms that take multiple sorted lists as a medium of input and in turn produce a single list as an output. This output contains all the elements of the inputs lists in a neatly sorted out order. These algorithms are then used as subroutines in various sorting algorithms, which most famously merge sort.

Header Linked List

A header linked list is a linked list that contains a special note at the front of the list. This special node is called a **headed node** and it does not contain any actual data item that is included in the list but generally contains some useful information about the entire linked list.

2.17 Insertion and Deletion Operations in Linked List

To insert a node in the beginning of a list, the next field of the new node (pointed to by nptr) is made to point to the existing first node and the Start pointer is modified to point to the new node as shown in **Figure 2.33**.

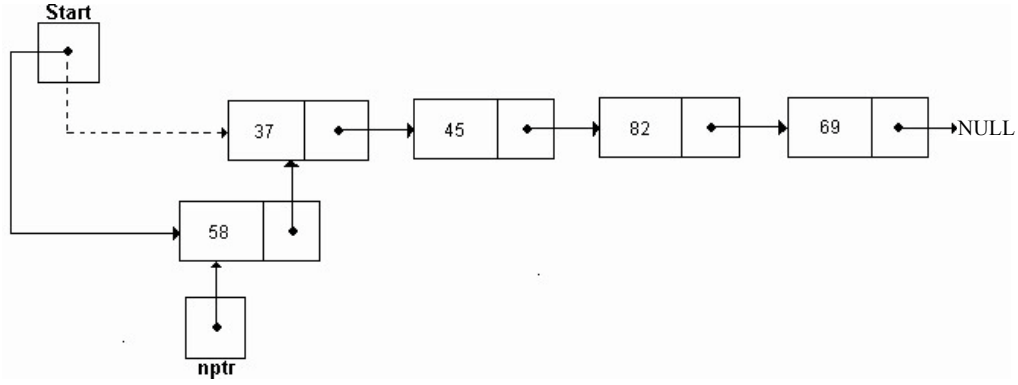


Fig. 2.33 Insertion in the Beginning of a Linked List

Algorithm 1: Insertion in Beginning

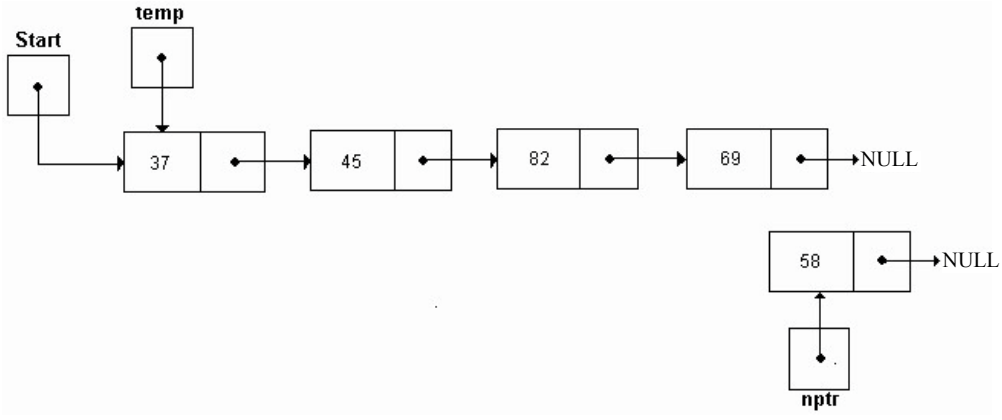
```
insert_beg (Start)
1. Call create_node() //creating a new node pointed
   to nptr
2. Set nptr->next = Start
3. Set start = nptr //Start pointing to new node
4. End
```

Insertion at End

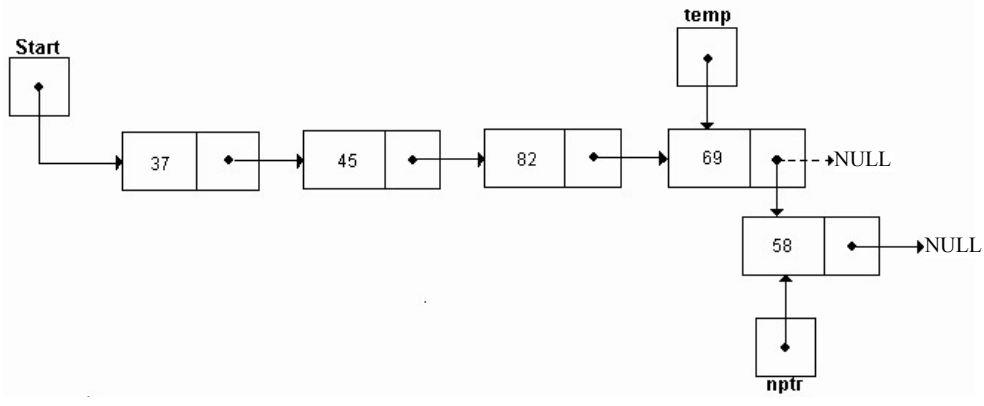
To insert a node at the end of a linked list, the list is traversed up to the last node and the next field of this node is modified to point to the new node. However, if the linked list is initially empty then the new node becomes the first node and start points to it. **Figure 2.34(a)** shows a linked list with a pointer variable temp pointing

to its first node and Figure **2.34(b)** shows temp pointing to the last node and the next field of last node pointing to the new node.

Notes



(a)



(b)

Figs. 2.34 (a) and (b) Insertion at the End of a Linked List

Algorithm 2: Insertion at the End

```
insert_end (Start)
1. Call create_node() //creating a new node pointed
   to nptr
2. If start = NULL //checking for empty list
   Set start = nptr //inserting new node as the first
   node
   Else
      Set temp = Start
   While temp->next ! = NULL //traversing up to the
   last node
       Set temp = temp->next
   End While
   Set temp->next = nptr //appending new node at the
   end
   End if
3. End
```

Insertion at a Specified Position

To insert a node at a position pos as specified by the user, the list is traversed up to pos-1 position. Then the next field of the new node is made to point to the node that is already at the pos position and the next field of the node at pos 1 position is made to point to the new node. **Figure 2.35** shows the insertion of the new node pointed to by nptr at the third position.

Notes

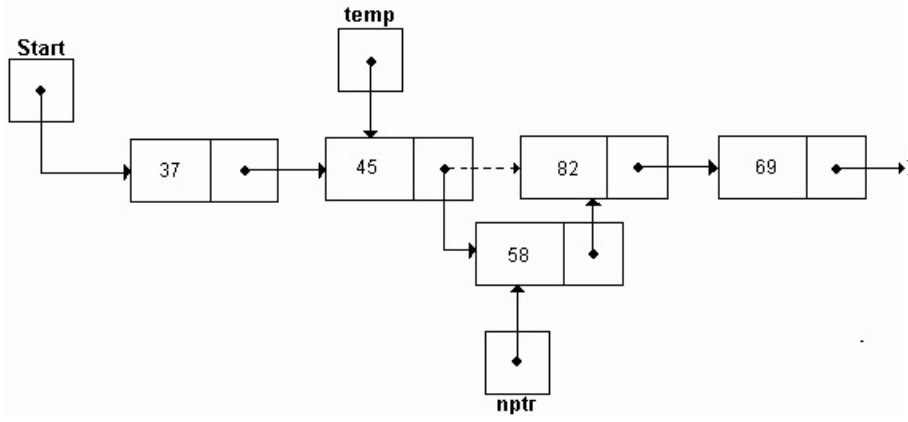


Fig. 2.35 Insertion at a Specified Position in a Linked List

Algorithm 3: Insertion at a Specified Position

```
insert_pos (Start)
1. Call create_node () //creating a new node pointed
   to by step nptr
2. Set temp = Start
3. Read pos //position at which the new node is to
   be  inserted
4. Call count_node(temp) //counting total number of
   nodes in count  variable
5. If (pos > count + 1 OR pos = 0)
   Print "Invalid position!" and go to step 7
   End If
6. If pos = 1
   Set nptr->next = Start
   Set Start = nptr //inserting new node as the first
   node  Else
   Set i = 1
   While i < pos - 1 //traversing up to the node at
   pos-1  position
   Set temp = temp-> next
   Set i = i + 1
   End While
   Set nptr->next = temp->next //inserting new node
   at pos position
   Set temp->next = nptr
   End If
7. End
```

Notes

Deletion

Like insertion, nodes can be deleted from the linked list at any point of time and from any position. Whenever a node is deleted, the memory occupied by the node is de-allocated. It must be noted that while performing deletions, the immediate predecessor of the node to be deleted must be keep track of. Thus, two temporary pointer variables are used (except in case of deletion from beginning), while traversing the list.

Note: A situation where the user tries to delete a node from an empty linked list is termed as underflow.

Deletion from Beginning

To delete a node from the beginning of a linked list, the address of the first node is stored in a temporary pointer variable temp and start is modified to point to the second node in the linked list. After this, the memory occupied by the node pointed to by step temp is de-allocated. **Figure 2.36** shows the deletion of a node from the beginning of a linked list.

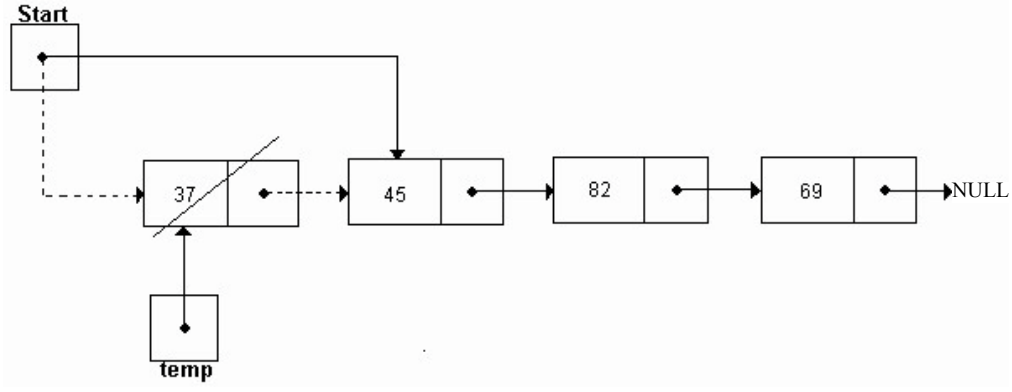


Fig. 2.36 Deletion from the Beginning of a Linked List

Algorithm 4: Deletion from the Beginning

```
delete_beg (Start)
1. If start = NULL //checking for underflow
   Print "Underflow: List is empty!" and go to step 5
   End if
2. Set temp = Start //temp pointing to the first node
3. Set start = Start-> next //moving start to point
   to the second node
4. Deallocate temp //deallocating memory
5. End
```

Deletion from End

To delete a node from the end of a linked list, the list is traversed up to the last node. Two pointer variables save and temp are used to traverse the list where save points to the node as previously pointed to by temp. At the end of traversing, temp points to the last node and save points to by the second last node. Then the next field

of the node pointed to by save is made to point to NULL and the memory occupied by the node pointed to by temp is de-allocated. **Figure 2.37** shows the deletion of a node from the end of a linked list.

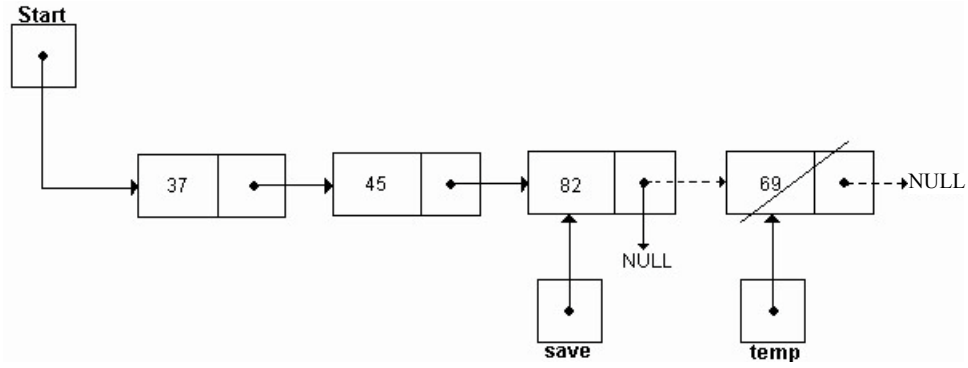


Fig. 2.37 Deletion from the End of a Linked List

Algorithm 5: Deletion from the End

```
delete_end (Start)
1. If start = NULL //checking for underflow   Print
   "Underflow: List is empty!" and go to step 6
   End if
2. Set temp = Start //temp pointing to the first node
3. If temp-> next = NULL //deleting the only node of
   the list
   Set start = NULL
   Else
   While (temp-> next) != NULL //traversing up to by
   the last node
   Set save = temp //save pointing to node previously
   //pointed to step temp
   Set temp = temp->next //moving temp to point to
   next node
   End while
   End if
4. Set save-> next = NULL //making new last node to
   point to NULL
5. Deallocate temp //deallocating memory
6. End
```

Deletion from a Specified Position

To delete a node from a position pos as specified by the user, the list is traversed up to pos position using pointer variables temp and save. At the end of traversing, temp points to the node at pos position and save points to the node at pos-1 position. Then the next field of the node pointed to by save is made to point to the node at pos+1 position and the memory occupied by the node as pointed to by temp is de-allocated. **Figure 2.38** shows the deletion of a node at the third position.

Notes

Notes

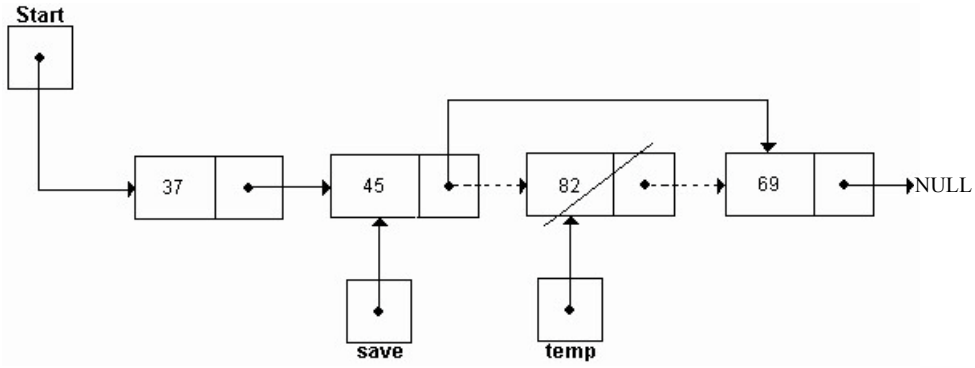


Fig. 2.38 Deletion from a Specified Position in a Linked List

Algorithm 6: Deletion from a Specified Position

```
delete_pos (Start)
1.  If start = NULL //checking for underflow
    Print "Underflow: List is empty!" and go to step 8
    End if
2.  Set temp = start
3.  Read pos //position of the node to be deleted
4.  Call count_node (Start) //counting total number
    of nodes in count variable
5.  If pos > count OR pos = 0
    Print "Invalid position!" and go to step 8
    End if
6.  If pos = 1
    Set start = temp->next //deleting the first node
    Else
      Set i = 1
     While i < pos //traversing up to the node at
    position pos   Set save = temp
      Set temp = temp->next
      Set i = i + 1
    End while
      Set save-> next = temp->next //deleting the node
    at position pos
    End if
7.  Deallocate temp //deallocating memory
8.  End
```

Searching

Searching a value for example item in a linked list means finding the position of a node, which stores item as its value. If item is found in the list, the search is successful and the position of that node is displayed. However, if item is not found till the end of list, then search is unsuccessful and an appropriate message is displayed. It must be noted that the linked list may be in a sorted or an unsorted

order. Therefore, two search algorithms are discussed, one for sorted and another for an unsorted linked list.

Note: Only linear search can be performed on linked lists.

Notes

Searching in an Unsorted List

If the data in a linked list is not arranged in a specific order, the list is traversed completely, starting from the first node to the last node and the value of each node (node->info) is compared with the value to be searched.

Algorithm 7: Searching in an Unsorted List

```
search_unsort (Start)
1. If start = NULL
   Print "List is empty!!" and go to step 7
   End if
2. Set ptr = Start //ptr pointing to the first node
3. Set pos = 1
4. Read item //item is the value to be searched  5.
   While ptr != NULL //traversing up to the last node
   If item = ptr->info
   Print "Value found at position", pos and go to
   step 7   Else
   Set ptr = ptr->next //moving ptr to point to next
   node   Set pos = pos + 1
   End if
   End while
6. Print "Value not found" //search unsuccessful
7. End
```

Searching in a Sorted List

The process of searching an item in a sorted (ascending order) linked list is similar to that of an unsorted linked list. However, while comparing, once the value of any node exceeds item (the value to be searched), the search is stopped immediately. In such a case, the list is not required to be traversed completely.

Algorithm 8: Searching in a Sorted List

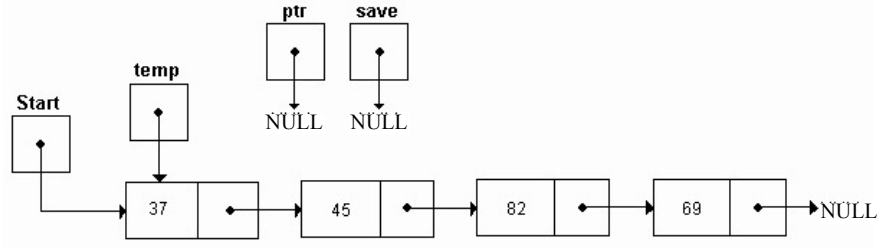
```
search_sort (Start)
1. If start = NULL
   Print "List is empty!!" and go to step 7
   End if
2. Set ptr = Start //ptr pointing to the first node
3. Set pos = 1
4. Read item
5. While ptr-> next != NULL //traversing up to the
   last node   If item < ptr->info //comparing item
   with the value of current node
   Print "Value not found" and go to step 7
   Else If item = ptr->info
   Print "Value found at position", pos and go to
   step 7 Else
```

```
        Set ptr = ptr->next //moving ptr to point to next
        node   Set pos = pos + 1
        End if
        End while
    6.  Print "Value not found" //search unsuccessful
    7.  End
```

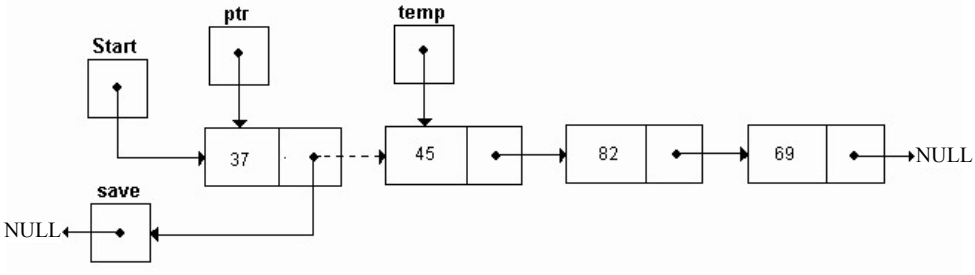
Notes

Reversing

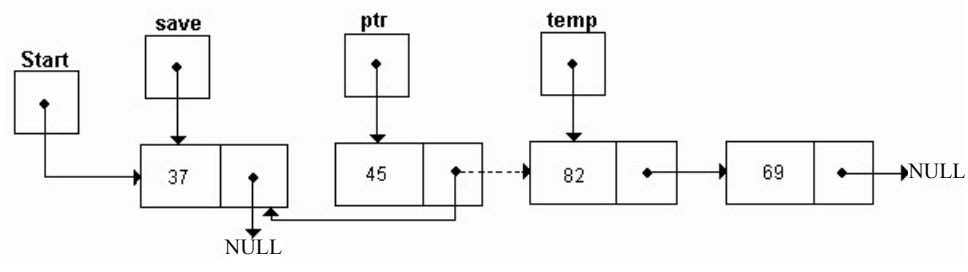
To reverse a singly-linked list, the list is traversed up to the last node and links of the nodes are reversed such that the first node of the list becomes the last node and the last node becomes the first. For this, three pointer variables like save, ptr and temp are used. Initially, temp points to start and both ptr and save point to NULL. While traversing the list, temp points to the current node, ptr points to the node previously pointed to by temp and save points to the node previously pointed to by ptr. The links between the nodes are reversed by making the next field of the node pointed to by ptr to point to the node pointed to by save. At the end of traversing, temp points to NULL, ptr points to last node, and save points to the second last node of the list. Then start is made to point to the node pointed to by ptr in order to make the last node as the first node of the list. **Figure 2.39(a to f)** shows the process of reversing a linked list.



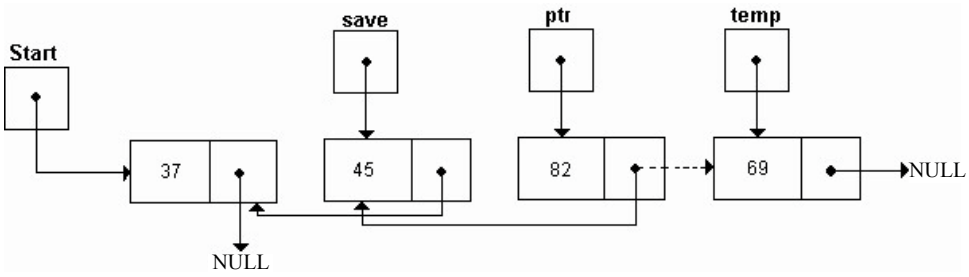
(a)



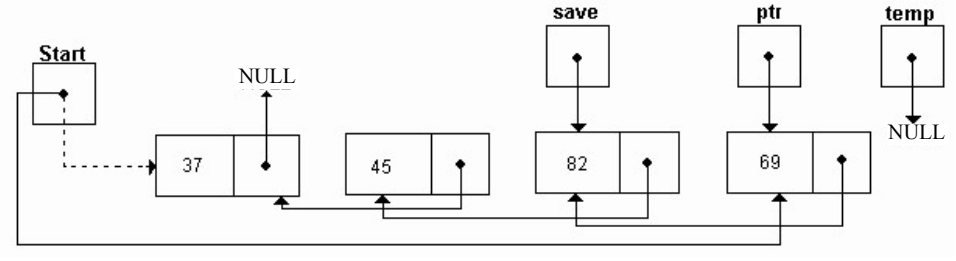
(b)



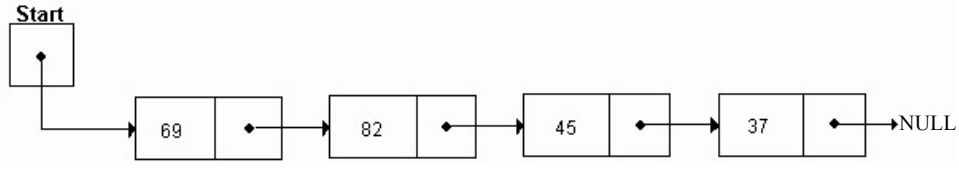
(c)



(d)



(e)



(f)

Fig. 2.39(a to f): Reversal of a Linked List

Algorithm 9: Reversing a Singly-Linked List

```
reverse (Start)
1.  Set temp = Start
2.  Set ptr = NULL
3.  Set save = NULL
4.  While temp != NULL //traversing up to the last
    node   Set save = ptr
    Set ptr = temp
    Set temp = temp-> next
    Set ptr-> next = save
    End while
5.  Set start = ptr
6.  End
```

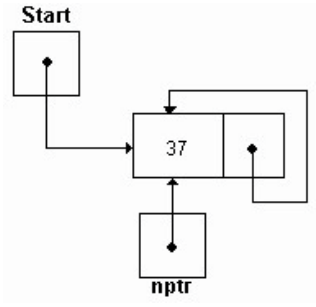
The following programme shows the searching and reversing operations on a singly–linked list. It must be noted that to simplify a programme the linked list is built by creating nodes and inserting them at the end of a list.

Notes

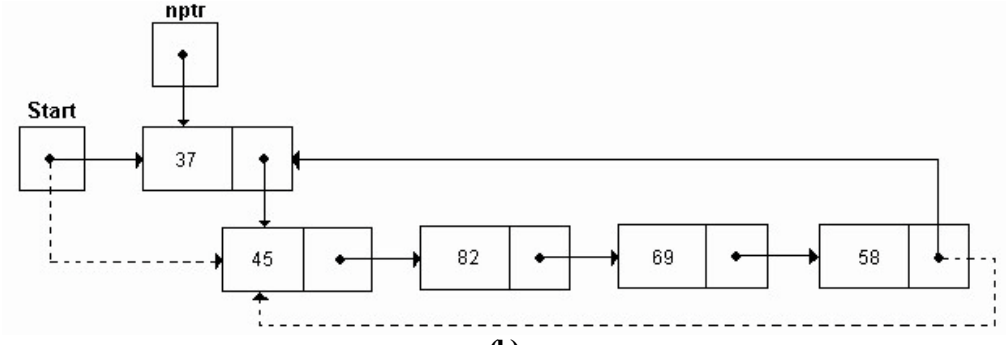
Notes

2.18 Insertion and Deletion in Circular Linked List

Like linear linked lists, nodes can be inserted at any position in a circular linked list, at the beginning, end or at a specified position. Insertion of a new node at various positions has been discussed in this section. To insert a new node (pointed to by nptr) at the beginning of a circular linked list [**Figure 2.40**(b)], the next field of the new node is made to point to the existing first node and the start pointer is modified to point to the new node. Since the first node of the list is changed, the next field of the last node also needs to be modified to point to a new node. However, if initially the list is empty, a new node is inserted as the first node and its next field is made to point to itself as shown in **Figure 2.40**(a).



(a)



(b)

Fig. 2.40(a and b): Insertion in the Beginning

Algorithm 1: Insertion in the Beginning

```
insert_beg (Start)
1.  Call create_node () //creating a new node pointed
    to by nptr
2.  If start = NULL //checking for empty list
    Set start = nptr //inserting new node as the first node
    Set start->next = Start
    Else
    Set temp = Start
    While temp->next != Start //traversing up to the
    last node
    Set temp = temp-> next
    End while
    Set nptr->next = Start //inserting new node in the
    beginning
```

```
    Set start = nptr //Start pointing to new node
    Set temp-> next = Start //next field of last node
    pointing to new node
    End if
3.  End
```

Insertion at the End

Notes

While inserting a new node (pointed to by nptr), at the end of a circular linked list, the list is traversed up to the last node. The next field of the last node is made to point to a new node and the next field of the new node is made to point to start. However, if the circular linked list is empty, a new node becomes the first node and start points to it. In addition, the next field of the new node points to itself as it is the only node in the list. **Figure 2.41** shows the insertion of a new node pointed to by nptr at the end of a circular linked list.

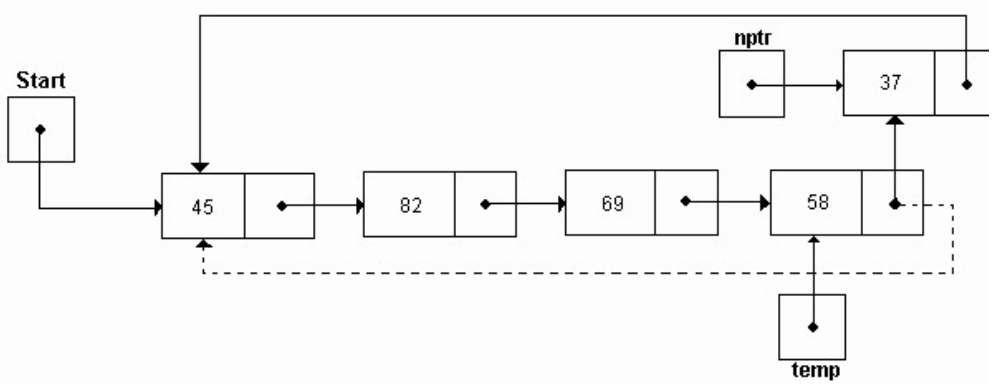


Fig. 2.41 Insertion at the End

Algorithm 2: Insertion at the End

```
    insert_end (Start)
1.  Call create_node() //creating a new node pointed
    to by nptr
2.  If start = NULL //checking for empty list   Set
    start = nptr //inserting new node in the empty
    linked list
    Set start-> next = Start //next field of first node
    pointing to itself
    Else
    Set temp = Start
    While temp-> next != Start //traversing up to the
    last node
    Set temp = temp-> next
    End while
    Set temp-> next = nptr //next field of last node
    pointing to new node
    Set nptr-> next = Start //next field of new node
    pointing to start
    End if
3.  End
```

Notes

Deletion

To delete a node from the beginning of a circular linked list, start is modified to point to the second node and field next of the last node is made to point to the new first node. For this, two pointer variables temp and ptr are used. Pointer temp stores the address of the node to be deleted (address of the first node) and start is modified to point to the second node. Pointer ptr is used for traversing the list and at the end of traversing, it stores the address of the last node. Then field next of the last node is made to point to the new first node. Also, memory occupied by the node pointed to by temp is de-allocated. **Figure 2.42** shows the deletion of a node from the beginning of a circular linked list.

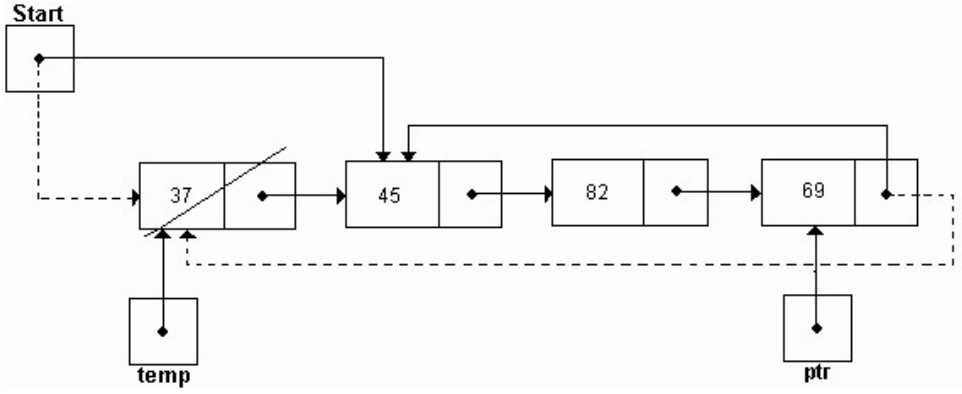


Fig. 2.42 Deletion from the Beginning

Algorithm 3: Deletion from the Beginning

```
delete_beg (Start)
1.  If start = NULL
    Print "Underflow: List is empty!" and go to step 8
    End if
2.  Set temp = Start
3.  Set ptr = temp
4.  While ptr-> next != Start //traversing up to the
    last node   Set ptr = ptr->next
    End while
5.  Set start = Start-> next //Start pointing to the
    next node
6.  Set ptr-> next = Start //last node pointing to new
    first node
7.  Deallocate temp //deallocating memory
8.  End
```

Deletion from the End

To delete a node from the end of a circular linked list, two pointer variables save and temp are used. Pointer variable temp is used to traverse the list and save points to the node previously pointed to by temp. At the end of traversing, temp points to the last node and save points to the second last node. Then the next field of save is made to point to Start and the memory occupied by the last node, *i.e.*, pointed

to by temp is de-allocated. **Figure 2.43** shows the deletion of a node from the end of a circular linked list.

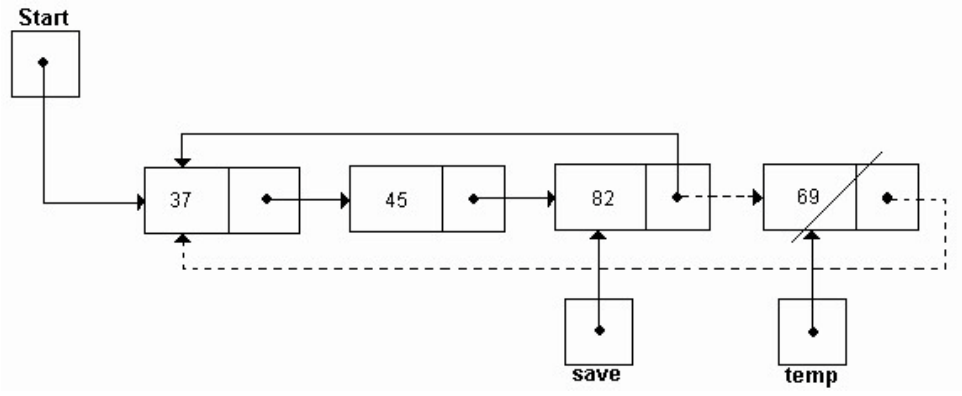


Fig. 2.43 Deletion from the End

Algorithm 4: Deletion from the End

```
delete_end (Start)
1.  If start = NULL //checking for underflow
    Print "Underflow: List is empty!" and go to step 5
    End if
2.  Set temp = Start
3.  If temp-> next = Start //deleting the only node
    of the list
    Set start = NULL
    Else
    While temp-> next != Start //traversing up to the
    last node
    Set save = temp
    Set temp = temp-> next
    End while
    Set save->next = Start //second last node becomes
    the last node
    End if
4.  Deallocate temp //deallocating memory
5.  End
```

Note: The process of deleting a node from a specified position in a circular linked list is same as that of a singly-linked list.

2.19 Insertion and Deletion in Doubly-Linked Lists

To insert a new node in the beginning of a doubly-linked list, a pointer, for example nptr to new node is created. The next field of the new node is made to point to the existing first node and prev field of the existing first node (that has become the second node now) is made to point to the new node. After that, start is modified to point to the new node. **Figure 2.44** shows the insertion of node in the beginning of a doubly-linked list.

Notes

Notes

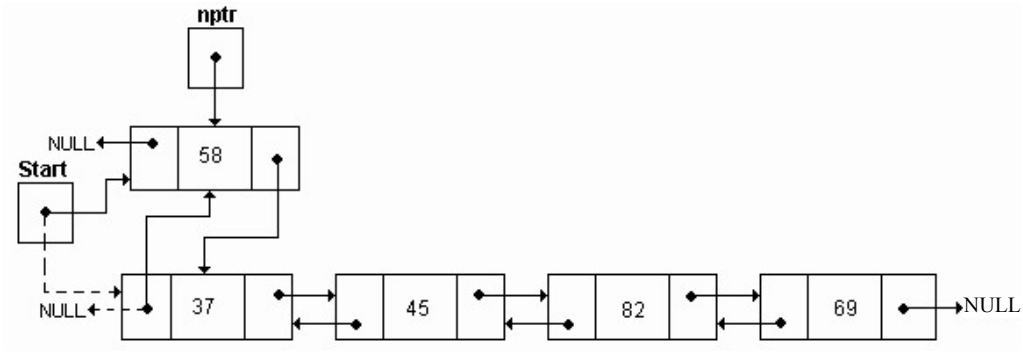


Fig. 2.44 Insertion in the Beginning

Algorithm 1: Insertion in the Beginning

```
insert_beg (Start)
1. Call create_node() //creating a new node pointed
   to by nptr
2. If start != NULL
   Set nptr->next = Start //inserting node in the
   beginning
   Set start->prev = nptr
   End if
3. Set start = nptr //making Start to point to new
   node
4. End
```

Insertion at the End

To insert a new node at the end of a doubly-linked list, the list is traversed up to the last node using some pointer variable, for example, the temp. At the end of traversing, temp points to the last node. Then, field next of the last node (pointed to by temp), is made to point to the new node and the field prev of the new node is made to point to the node pointed to by temp. However, if a list is empty, the new node is inserted as the first node in the list. **Figure 2.45** shows the insertion of a new node at the end of a doubly-linked list.

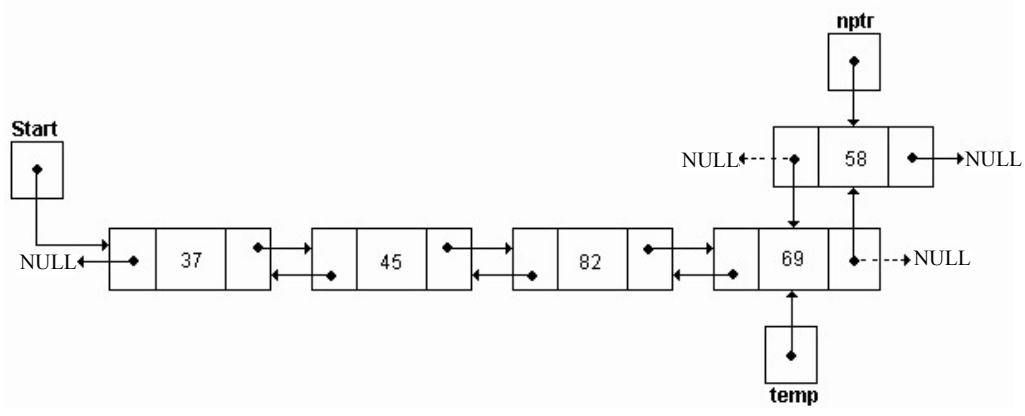


Fig. 2.45 Insertion at the End

Algorithm 2: Insertion at the End

```
insert_end (Start)
1. Call create_node () //creating a new node pointed
   to by nptr
2. If start = NULL
   Set start = nptr //inserting new node as the first
   node
   Else
   Set temp = Start //pointer temp used for traversing
   While temp-> next != NULL
   Set temp = temp-> next
   End while
   Set temp-> next = nptr
   Set nptr-> prev = temp
   End if
3. End
```

Notes

Insertion at a Specified Position

To insert a new node (pointed to by nptr) at a specified position, for example, pos in a doubly-linked list, the list is traversed up to pos-1 position. At the end of traversing, temp points to the node at pos-1 position. For simplicity, another pointer variable, ptr is used to point to the node that is already at position pos. Then, field prev of the node, pointed to by ptr is made to point to the new node and field next of the new node is made to point to the node pointed to by ptr. Also, field prev of the new node is made to point to the node pointed to by temp and field next of the node pointed to by temp is made to point to the new node. **Figure 2.46** shows the insertion of a new node at the third position in a doubly-linked list.

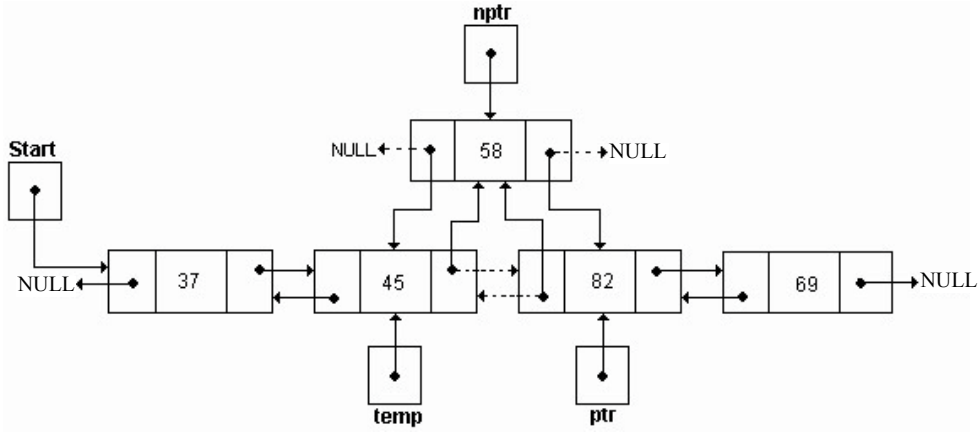


Fig. 2.46 Insertion at a Specified Position

Algorithm 3: Insertion at a Specified Position

```
insert_pos (Start)
1. Call create_node () //creating a new node pointed
   to by nptr
```

Notes

```
2.  Set temp = Start
3.  Read pos
4.  Call count_node(temp) //counting number of nodes
    in count
5.  If pos = 0 OR pos > count + 1
    Print "Invalid position!" and go to step 7
    End if
6.  If pos = 1
    Set nptr-> next = Start //inserting node at the
    beginning
    Set Start = nptr //Start pointing to new node  Else
    Set i = 1
    While i < pos-1 //traversing up to the node at
    pos-1 position
    Set temp = temp-> next
    Set i = i + 1
    End while
    Set ptr = temp-> next
    Set ptr-> prev = nptr
    Set nptr-> next = ptr
    Set nptr-> prev = temp
    Set temp-> next = nptr
      End if
7. End
```

Deletion

To delete a node from the beginning of a doubly-linked list, a pointer variable, for example, temp is used to point to the first node. Then start is modified to point to the next node and the prev field of this node is made to point to NULL. After that, the memory occupied by the node pointed to by temp is de-allocated. **Figure 2.47** shows the deletion of a node from the beginning of a doubly-linked list.

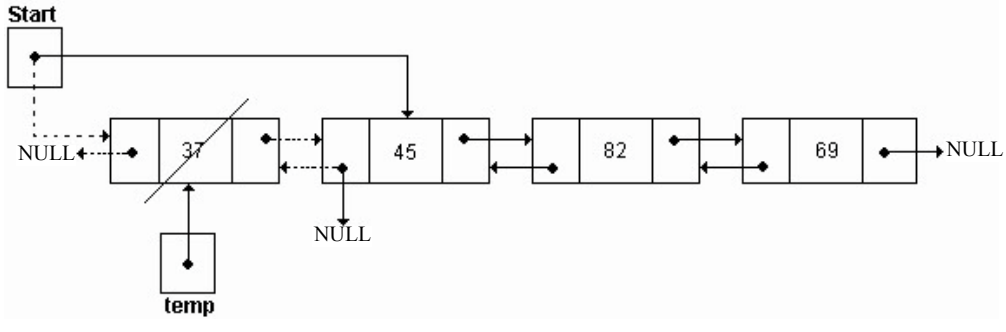


Fig. 2.47 Deletion from the Beginning

Algorithm 4: Deletion from the Beginning

```
    delete_beg (Start)
1.  If start = NULL
    Print "Underflow: List is empty!" and go to step 6
    End if
```

```
2. Set temp = Start //temp points to the node to be
   deleted
3. Set Start = Start-> next //making start to point
   to next node
4. Set start-> prev = NULL
5. Deallocate temp //de-allocating memory
6. End
```

Notes

Note: The process of deleting node from the end of a doubly-linked list is same as that of singly-linked list.

Deletion from a Specified Position

To delete a node from a position, for example, pos, as specified by the user, the list is traversed up to the position pos, using pointer variables temp and save. At the end of traversing, temp points to the node at pos position and save points to the node at pos-1 position. For simplicity, another pointer variable ptr is used to point to the node at pos +1 position. Then the next field of the node at pos –1 position (pointed to by save) is made to point to the node at pos +1 position (pointed to by ptr). In addition, the field prev of the node at pos +1, position (pointed to by ptr) is made to point to the node at pos –1 position (pointed to by save). After that, the memory occupied by the node pointed to by temp is de-allocated. **Figure 2.48** shows the deletion of a node at the third position from a doubly-linked list.

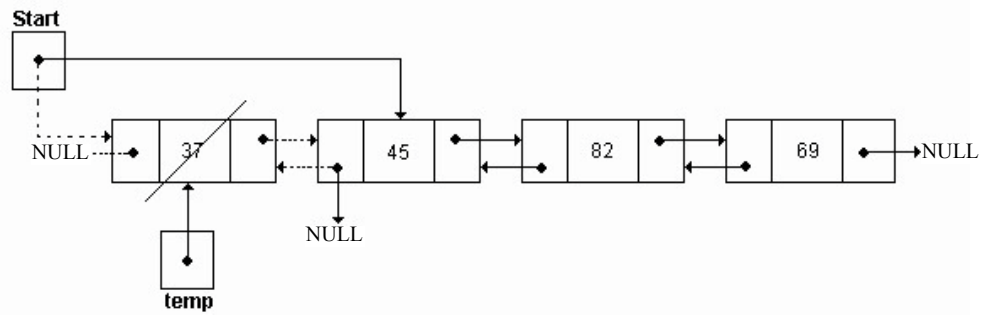


Fig. 2.48 Deletion from a Specified Position

Algorithm 5: Deletion from a Specified Position

```
delete_pos (Start)
1. If start = NULL
   Print "Underflow: List is empty!" and go to step 8
   End if
2. Set temp = Start
3. Read pos
4. Call count_node (temp) //counting total number of
   nodes in count variable
5. If pos > count OR pos = 0
   Print "Invalid position!" and go to step 6
   End if
6. If pos = 1
   Set Start = Start-> next //deleting the first node
   Start-> prev = NULL
```

Notes

```
Else
     Set i = 1
While i < pos //traversing up to the node at pos position
Set save = temp //save pointing to the node at pos-1 position
Set temp = temp->next //making temp to point to next node   Set i = i + 1
End while
Set ptr = temp-> next
Set save-> next = ptr
Set ptr-> prev = save
End if
7. Deallocate temp //de-allocating memory
8. End
```

Note: A doubly-linked list, in which the next field of the last node points to the first node instead of 'NULL', is termed as a doubly-circular linked list.

2.20 Traversing Linked Lists

Traversing

Traversing a list means accessing the elements of a linked list, one-by-one, to process all or some of the elements. For example, to display values of the nodes, the number of nodes counted, or a particular item in the list is searched, then traversing is required. A list can be traversed by using a temporary pointer variable temp, which will point to the node that is currently being processed. Initially, temp points to the first node, processes that element, moves temp point to the next node using the statement temp = temp-> next, processes that element, and moves on as long as the last node is not reached, that is, until temp becomes NULL.

Algorithm 1: Traversing a List

```
display (Start)
1. If start = NULL //Start points to the first node of list
   Print "List is empty!!" and go to step 4
   End if
2. Set temp = Start //initialising temp with start
3. While temp != NULL
   Print temp-> info //displaying value of each node
   Set temp = temp-> next //moving temp to point to next node
   End while
4. End
```

Another example of traversing a linked list is counting the number of nodes in the linked list, which is described in the algorithm as illustrated here.