PGCA/Sc IT/PGDCA (S2) 02
Exam. Code: DS

Data Structure Through C Language

SEMESTER II

MASTER OF COMPUTER APPLICATION / MASTER OF SCIENCE IN INFORMATION TECHNOLOGY / POST GRADUATE DIPLOMA IN COMPUTER APPLICATION

Block 1



KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakati Mahanta, Gauhati University
Prof. (Retd.) Pranhari Talukdar, Gauhati University
Dr. Jyotiprokash Goswami, Assam Engineering College
Prof. Sikhar Kumar Sarma, Cotton University

Course Co-ordinators

Dr. Tapashi Kashyap Das, KKHSOU
Ms. Sruti Sruba Bharali, KKHSOU

SLM Preparation Team

UNITS	CONTRIBUTORS
1 & 7	Mr. Arabinda Saikia, KKHSOU
2	Dr. Naba Jyoti Sarma, Nalbari Commerce College
3 & 5	Dr. Swapnali Gogoi, IDOL, Gauhati University
4 & 6	Dr. Tapashi Kashyap Das, KKHSOU

Editorial Team

Content : Ms. Sruti Sruba Bharali, KKHSOU
Language : Prof. (Retd.) Robin Goswami, Cotton College
Structure, Format & Graphics: Ms. Sruti Sruba Bharali, KKHSOU

April, 2019 ISBN: 978-93-89123-03-6

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State University is made available under a Creative Commons Attribution-Non Commercial-ShareAlike4.0 License (International): http.//creativecommons.org/licenses/by-nc-sa/4.0.

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University.

Headquarters : Patgaon, Rani Gate, Guwahati-781 017
City Office : Housefed Complex, Dispur, Guwahati-781 006; Web: www.kkhsou.in

The university acknowledges with thanks the financial support provided by the **Distance Education Bureau, UGC** for the preparation of this study material.

MASTER OF COMPUTER APPLICATION / MASTER OF SCIENCE IN INFORMATION TECHNOLOGY / POST GRADUATE DIPLOMA IN COMPUTER APPLICATION

Data Structure Through C Language

Block-1

DETAILED SYLLABUS

		Pages
UNIT 1:	**Introduction to Data Structure** Data and Information; Data Structure and Its Types; Data Structure Operations; Different Data Types: Arraya, Pointers, Structures; Dynamic Memory Allocation; Abstract Data Types	**7-31**
UNIT 2:	**Algorithms** Algorithm; Complexity: Space Complexity, Time Complexity; Asymptotic Notation: Theta Θ Notation, Big O (O) Notation, Big Omega (Ω) Notation	**32-49**
UNIT 3:	**Linked List** Linked List; Types of Linked List: Singly Linked List, Doubly Linked List, Circular Linked List; Traversing a Single Linked List; Searching a Single Linked List; Insertion into a Single Linked List; Deletion from a Single Linked List; Reversing a Single Linked List	**50-74**
UNIT 4:	**Stack** Stack; Operations on Stack; Implementation of Stack: Implementation of Stack using Arrays, Implementation of Stack using Linked Lists; Applications of Stack	**75-90**
UNIT 5:	**Queue** Queue; Array Implementation of Queue; Circular Queue; Linked List Implementation of Queue: Using Singly Linked List, Using Doubly Linked List, Using Circular Linked List; Priority Queues; Application of Queues	**91-135**
UNIT 6:	**Searching** Searching; Types of Searching: Linear Search, Binary Search	**136-150**
UNIT 7:	**Sorting** Sorting; Insertion Sort; Selection Sort; Bubble Sort; Quick Sort	**151-172**

COURSE INTRODUCTION

This is a course on ***Data Structure Through C Language***. A data structure is a particular way of storing and organizing data in a computer's memory or even disk storage so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. The commonly used data structures in various programming languages, like C, are arrays, linked list, stack, queues, different types of trees, graph etc. This course is designed to acquaint the learner such type of data structures.

The course is divided into two blocks:

Block 1 deals with different data structures like arrays and linked list. Other data structure's like queues, stacks and trees are implemented using array and linked list. Searching and sorting techniques are also included as topics in this block.

Block 2 concentrates on different types of trees like binary tree, binary search tree, AVL tree, red black tree etc. Concept of priority queues, partition and union are also discussed in this block. At the end different types and traversal techniques for graphs is discussed.

BLOCK INTRODUCTION

This is the *First Block* of the course ***Data Structure Through C Language***. After completing this block, learners will be able to write basic programs using different data structures.

This block comprises of the following seven units:

Unit 1: introduces you some elementary concepts like data, types of data, structure, pointer, array etc. as well as memory representation, address translation functions.

Unit 2: concentrates on algorithms, complexity of algorithm in terms of time and space and their notations.

Unit 3: deals with one of the most important linear data structure linked list. Representation of linked list, their types, operations associated with linked list like searching, insertion and deletion of element in a linked are described in this unit.

Unit 4: focuses on stack data structure. In this unit, various operations associated with stacks as well as their implementation using array and linked list are discussed.

Unit 5: concentrates on queue data structure.This unit discusses array as well as linked implementation of queue, applications of queue etc.Concept of circular queue, priority queue are also covered in this unit.

Unit 6: deals with the searching techniques. Linear and binary search techniques with their relative advantages and disadvantages are discussed in this unit.

Unit 7: discusses different sorting techniques, their implementations, complexity, advantages and disadvantages.

Each unit of these blocks includes some along-side boxes to help you know some of the difficult, unseen terms. Some “EXERCISES” have been included to help you apply your own thoughts. You may find some boxes marked with: “LET US KNOW”. These boxes will provide you with some additional interesting and relevant information. Again, you will get “CHECK YOUR PROGRESS” questions. These have been designed to self-check your progress of study. It will be helpful for you if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with “ANSWERS TO CHECK YOUR PROGRESS” given at the end of each unit.

UNIT 1: INTRODUCTION TO DATA STRUCTURE

UNIT STRUCTURE

1.1 Learning Objectives

1.2 Introduction

1.3 Data and Information

1.4 Data Structure and Its Types

1.5 Data Structure Operations

1.6 Different Data Types

1.6.1 Arraya

1.6.2 Pointers

1.6.3 Structures

1.7 Dynamic Memory Allocation

1.8 Abstract Data Types

1.9 Let Us Sum Up

1.10 Further Reading

1.11 Answers to Check Your Progress

1.12 Model Questions

1.1 LEARNING OBJECTIVES

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

- distinguish data and information
- define data structure and describe various types of data structures
- describe different data structure operations
- define abstract data types.

1.2 INTRODUCTION

Data structure in Computer Science is a way of storing and organizing data in a computer's memory or even disk storage so that it can be used efficiently. It is an organization of mathematical and logical concepts of data. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory

space, as possible. Data structures are implemented by a programming language by the data types and the references and operations provide by that particular language.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. For example, B-trees are particularly well-suited for implementation of databases. In the design of many types of computer program, the choice of data structures is a primary design consideration. Experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure.

In this unit, we will introduce you to the fundamental concepts of data structure. We shall also discuss in the unit arrays, pointers and structures besides different data structure operations and abstract data types. In the next unit we will discuss about the time complexity and space complexity of algorithms.

1.3 DATA AND INFORMATION

The term **data** comes from its singular form **datum**, which means 'a fact'. The data is a fact about people, places or some entities. In computers, data is simply the value assigned to a variable. The term *variable* refers to the name of a memory location that can contain only one data at any point of time. For example, consider the following statements:

Vijay is 16 years old.

Vijay is in the 12th standard.

Vijay got 80% marks in Mathematics.

Let 'name', 'age', 'class', 'marks' and 'subject' be some defined variables. Now, let us assign a value to each of these variables from the above statements.

Name = Vijay

Class = 12

Age = 16

Marks = 80

Subject = Mathematics

In the above example the values assigned to the five different variables are called data.

We will now see what is **information** with respect to computers. Information is defined as a set of processed data that convey the relationship between data considered. *Processing* means to do some operations or computations on the data of different variables to relate them so that these data, when related, convey some meaning. Thus, *information* is as group of related data conveying some meaning. In the examples above, when the data of the variables 'name' and 'age' are related, we get the following information:

Vijay is 16 years old.

In the same example, when the data of the variables 'name' and 'class' are related we get another information as:

Vijay is in the 12th standard.

Further, when we relate the data of the variables, 'name', 'marks', and 'subject', we get more information that Vijay got 80% marks in Mathematics. The following figure shows how information can be drawn from data.

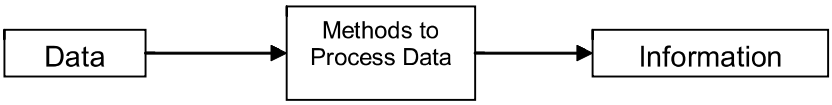


Figure 1.1: Relation between data and information

1.4 DATA STRUCTURE AND ITS TYPES

The way in which the various data elements are organized in memory with respect to each other is called a *data structure*. Data structures are the most convenient way to handle data of different types including *abstract data type* for a known problem. Again problem solving is an essential part of every scientific discipline. To solve a given problem by using a computer, you need to write a program for it. A program consists of two components: *algorithm* and d*ata structure*.

Many different algorithms can be used to solve the same problem. Similarly, various types of data structures can be used to represent a problem in a computer.

To solve the problem in an efficient manner, you need to select a combination of algorithms and data structures that provide maximum efficiency. Here, efficiency means that the algorithm should work in minimal time and use minimal memory. In addition to improving the efficiency of an algorithm, the use of appropriate data structures also allows you to overcome some other programming challenges, such as:

– simplifying complex problems

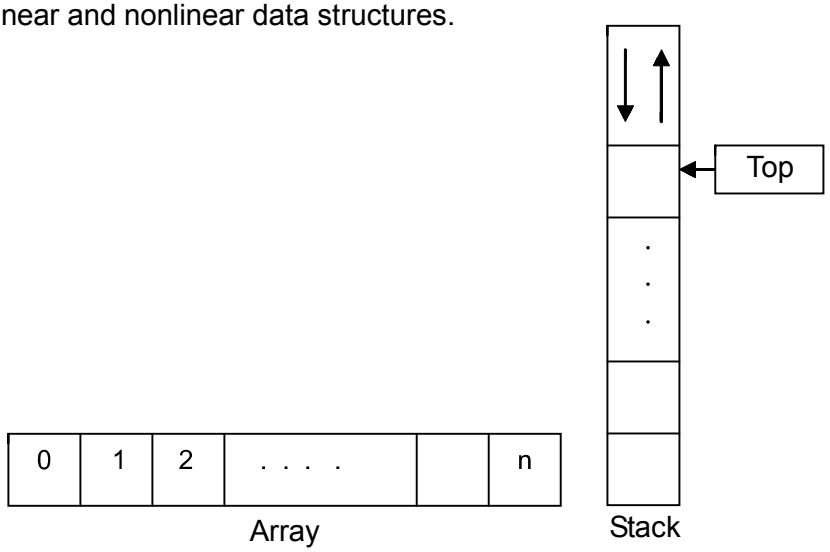
– creating standard, reusable code components

– creating programs that are easy to understand and maintain

Data can be organized in many different ways; therefore, you can create as many data structures as you want. However, data structures have been classified in several ways. Basically, data structures are of two types: **linear data structure** and **non linear data structure**.

Linear data structure: A data structure is said to be linear if the elements form a sequence i.e., while traversing sequentially, we can reach only one element directly from another. For example: Array, Linked list, Queue etc.

Non linear data structure: Elements in a nonlinear data structure do not form a sequence i.e. each item or element may be connected with two or more other items or elements in a non-linear arrangement. Moreover removing one of the links could divide the data structure into two disjoint pieces. For example: Trees and Graphs etc. The following figures shows the linear and nonlinear data structures.



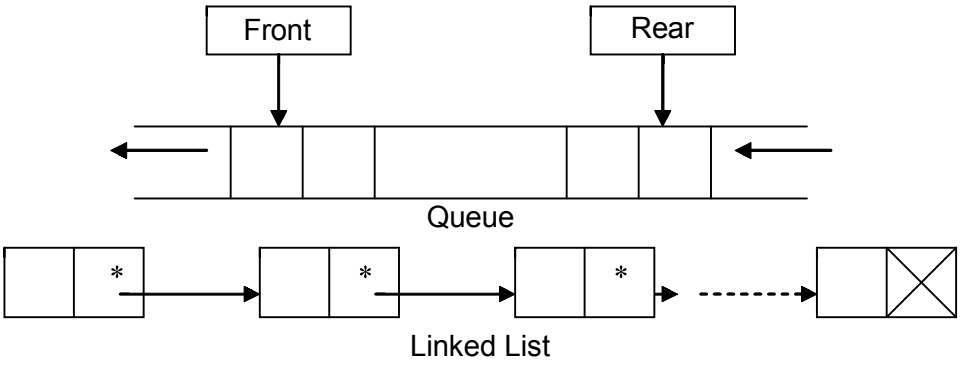


Figure 1.2: Linear data structure

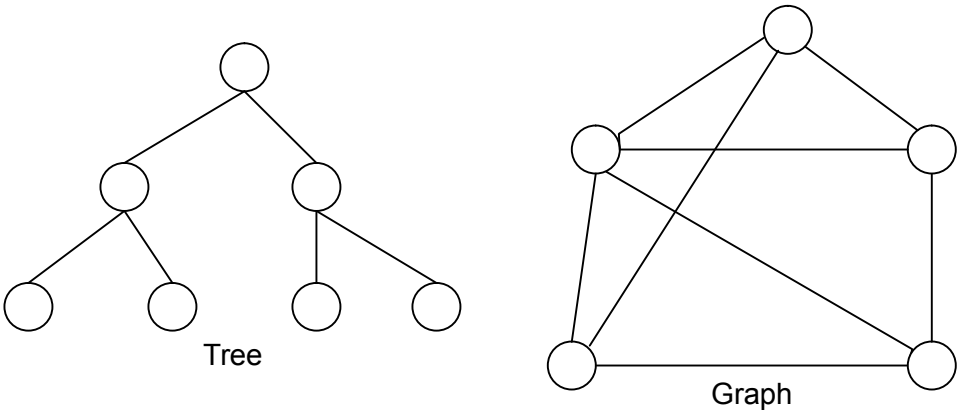


Figure 1.3: Non linear data structure

All these data structures are designed to hold a collection of data items.

1.5 DATA STRUCTURE OPERATIONS

We come to know that data structure is used for the storage of data in computer so that data can be used efficiently. The data manipulation within the data structures are performed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. The following four operations play a major role on data structures.

a) *Traversing:* accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called “visiting” the record.)

b) *Searching:* finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions.

c) *Inserting:* adding a new record to the structure.

d) *Deleting:* removing a record from the structure.

Sometimes two or more of these operations may be used in a given situation. For example, if we want to delete a record with a given key value, at first we willl have need to search for the location of the record and then delete that record.

The following two operations are also used in some special situations:

i) *Sorting:* operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabatically, with character data.

ii) *Merging:* combining the records in two different sorted files into a single sorted file.

1.6 CONCEPT OF DATA TYPES

A *data type* is nothing but a term that refers to the type of data values that may be used for processing and computing. The fundamental data types in C are *char, int, float* and *double*. These data types are called built-in data types. There are three categories of data types in C, they are:

a) Built-in types, includes *char, int, float* and *double*

b) Derived data types, includes *array* and *pointers*

c) User defined types, includes *structure, union and enumeration.*

In this section, we will briefly review the data types *array*, *pointers* and *structures* that we have already learnt in the first semester course Programming in C Language.

1.6.1 Array

An array is a collection of two or more adjacent memory locations containing same types of data. These data are the array elements. The individual data items can be characters, integers, floating-point numbers, etc. However, they must all be of the same type and the same storage class.

Each array element (i.e., each individual data item) is referred to by specifying the array name followed by one or more **subscripts**,

with each subscript enclosed in square brackets. The syntax for declaration of an array is given below.

Storage Class datatype arrayname [expression]

Here, *storage class* may be *auto, static or extern*, *Storage class* refers to the permanence of a variable, and its scope within the program, i.e., the portion of the program over which the variable is recognized. If the storage class is not given then the compiler assumes it is an *auto* storage class.

The array can be declared as:

int x[15]; **x** is an 15 element integer array

char name[25]; **name** is a 25 element character array

In the array x, the array elements are x[0], x[1], ..., x[14] as illustrated in the figure 1.4.

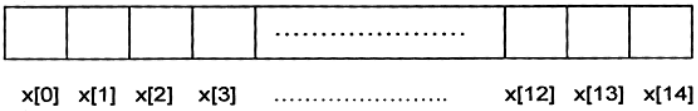


Figure 1.4: An array data structure

Array can be initialize at the time of the declaration of the array. For example,

int marks [5] ={ 85, 79, 60, 87, 70 };

Then, the **marks** array can be represented as follows:

85	79	60	87	70

marks [0] marks [1] marks [2] marks [3] marks [4]

Figure 1.5: the marks array after initialization

In the case of a character array *"name"* it is represented as:

char name [20] = { "krishna" }

K	r	i	s	h	n	a	\0

name [0] name [1] name [2] name [3] name [4] name[5] name[6] name[7]

Figure 1.6: "Name" array after initialization

We know that every character string terminated by a null character (\0). Some more declarations of arrays with initial values are given below:

char **vowels** [ ] = { 'A', 'E', 'I', 'O', 'U' };

int **age** [ ] = { 16, 21, 19, 5, 25 }

In the above case, compiler assumes that the array size is equal to the number of elements enclosed in the curly braces. Thus, in the above statements, size of array would automatically assumed to be 5. If the number of elements in the initializer list is less than the size of the array, the rest of the elements of the array are initialized to zero. The number of the subscripts determines the dimensionality of the array. For example,

marks [i],

refers to an element in the one dimensional array. Similarly, **matrix [i] [j]** refers to an element in the two dimensional array.

Two dimensional arrays are declared the same way that one dimensional arrays. For example,

int matrix [3] [5]

is a two dimensional array consisting of 3 rows and 5 column for a total of 20 elements. Two dimensional array can be initialized in a manner analogous to the one dimensional array:

```
int matrix [ 3 ] [ 5 ] =   {
                               { 10, 5,  –3, 9, 2 },
                               {  1 , 0, 14, 5, 6 },
                               { –1, 7,  4,  9,  2 }
                           };
```

The matrix array can be represented as follows:

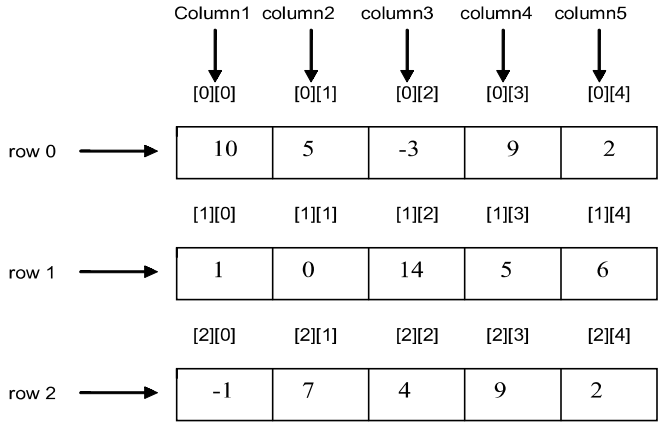


Figure 1.7: Matrix array after initialization

The above statement can be written as follows:

int matrix [3] [5] = {10, 5, –3, 9, 2, 1, 0, 14, 5, 6, –1, 7, 4, 9, 2}

A statement such as:

```
int matrix [ 3 ] [ 5 ] =  {
                              {10, 5, –3},
                              {1, 0, 14},
                              {–1, 7, 4}
                          };
```

only initializes the first three elements of each row of the two dimensional array *matrix*. The remaining values are set to 0.

Program 1.1: A Simple Program Using One-dimensional Array

A one dimensional array is used when it is necessary to keep a large number of items in memory and reference all the items in a uniform manner. Let us try to write a program to find average marks obtained by a class of 30 students in a test.

```
#include<stddio.h>
#include<conio.h>
void main( )
   {
      int avg, sum = 0;
      int i; int marks[30]; /* array declaration */
      clrscr( );
      for ( i = 0 ; i<=29 ; i++ )
      {
         printf ( ''\nEnter marks'' ) ;
         scanf ( ''%d", &marks[i] ) ; /* store
         data in array */
      }
   for ( i = 0 ; i <= 29 ; i++ )
      sum = sum + marks[i] ; /* read data
      from an array */
      avg = sum / 30 ;
```

```
      printf ( "\nAverage marks = %d", avg) ;
      getch( );
}
```

Program 1.2: A Simple Program Demonstrating the use of Two-Dimensional Array

A transpose of a matrix is obtained by interchanging its rows and columns. If A is a matrix of order m x n then its transpose A^T will be of order n x m. Let us implement this program using two dimensional array as follows:

```
#include<stdio.h>
#include<conio.h>
void main()
   {
      int matrix1 [20][20], matrix2 [20][20], i,
      j, m, n;
      clrscr();
      printf("Enter the Rows and Column of
      matrix\n");
      scanf("%d %d", &m, &n);
      printf("Enter the elements of the Matrix\n");
      for(i=1; i<m+1; i++)
      for(j=1; j<n+1; j++)
      scanf("%d", &matrix1 [i][j]);
      for(i=1; i<n+1; i++)
      for(j=1; j<m+1; j++) /*stores the elements
in matrix2*/
      matrix2 [i][j] = matrix1 [j][i];
      printf("\n\t Transpose of Matrix\n");
      for(i=1; i<n+1; i++)
   {
      for(j=1; j<m+1; j++)
      printf( "%3d", matrix2 [i][j]);
```

```
        printf("\n");
    }
    getch();
}
```

1.6.2 Pointers

You have already introduced to the concept of pointer. We will recall some of its properties and applications. A **pointer** is a variable that represents the **location** (rather than the **value**) of a data item, such as a variable or an array element.

Suppose we define a variable called **sum** as follows:

int sum = 25;

Let us define an another variable, called **pt_sum** like the following way,

int *pt_sum;

It means that **pt_sum** is a pointer variable pointing to an integer, where * is a unary operator, called the **indirection operator**, that operates only on a pointer variable.

We have already used the '**&**' *unary operator* as a part of a *scanf* statement in our C programs. This operator is known as the **address operator**, that evaluates the address of its operand.

Now, let us assign the address of sum to the variable **pt_sum** such as:

pt_sum = ∑

Now the variable **pt_sum** is called a **pointer** to **sum**, since it "points" to the location or address where **sum** is stored in memory. Remember, that **pt_sum** represents **sum's** address, not its value. Thus, **pt_sum** referred to as a **pointer variable**.

The relationship between pt_sum and sum is illustrated in Figure 1.8.

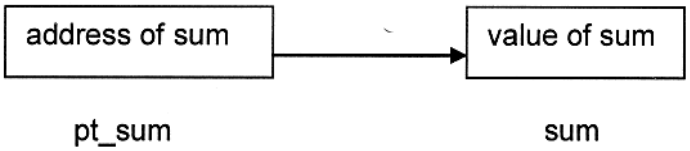


Figure 1.8: Relationship between pt_sum and sum

The data item represented by **sum** (i.e., the data item stored in sum's memory cells) can be accessed by the expression ***pt_sum**.

Therefore, ***pt_sum** and **sum** both represent the same data item i.e., 25.

Several typical pointer declarations in C program are shown below:

```
int *alpha;
char *ch;
float *s;
```

Here, alpha, ch and s are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers, therefore pointers always contain whole numbers.

The declaration **float *s** does not mean that **s** is going to contain a floating-point value. What it means is, **s** is going to contain the address of a floating-point value. Similarly, **char *ch** means that **ch** is going to contain the address of a char value.

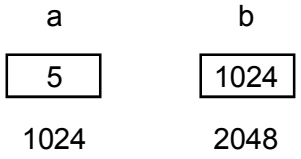
Let us try to write a program that demonstrate the use of a pointer:

Program 1.3: Program to demostrate the use of a pointer

```
#include<stdio.h>
#include<conio.h>
void main( )
   {
      int a = 5;
      int *b;
      b = &a;
```

```
    clrscr();
    printf ("value of a = %d\n", a);
    printf ("value of a = %d\n", *(&a));
    printf ("value of a = %d\n", *b);
    printf ("address of a = %u\n", &a);
    printf ("address of a = %d\n", b);
    printf ("address of b = %u\n", &b);
    printf ("value of b = address of a = %u", b);
    getch();
}
```

[Suppose address of the variable a = 1024, b = 2048]



OUTPUT: value of a = 5
value of a = 5
value of a = 5
address of a = 1024
address of a = 1024
value of b = address of a = 1024

Pointer to Pointer: A pointer to a pointer is a techniques used frequently in more complex programs. To declare a pointer to a pointer, place the variable name after two successive asterisks (*). In this case, one pointer variable holds the address of the another pointer variable. In the following shows a declaration of pointer to pointer:

int **x;

Following program shows the use of pointer to pointer techniques:

Program 1.4: Program showing use of a pointer

```
#include<stdio.h>
#include<conio.h>
```

```
void main( )
{
   int a = 5;
   int *b;
   int **c;
   b = &a;
   c = &b;
   printf ("value of a = %d\n", a);
   printf ("value of a = %d\n",*(&a));
   printf ("value of a = %d\n",*b);
   printf ("value of a = %d\n",**c);
   printf ("value of b = address of a = %u\n", b);
   printf ("value of c = address of b = %u\n", c);
   printf ("address of a = %u\n", &a);
   printf ("address of a = %u\n", b);
   printf ("address of a = %u\n", *c);
   printf ("address of b = %u\n", &b);
   printf ("address of b = %u\n", c);
   printf ("address of c = %u\n", &c);
   getch();
}
```

[Suppose address of the variable a = 1024, b = 2048]

a	b	c
5	1024	2048
1024	2048	4096

OUTPUT: value of a = 5
value of a = 5
value of a = 5
value of a = 5
value of b = address of a = 1024
value of c = address of b = 2048

address of a = 1024

address of a = 1024

address of a = 1024

address of b = 2048

address of b = 2048

address of c = 4096



CHECK YOUR PROGRSS

Q.1: Describe the array that is defined in each of the following statements. Indicate what values are assigned to the individual array elements.

a) char flag[5]= {'T', 'R', 'U', 'E'};

c) int p[2][4] = {
{1, 3, 5, 7},
{2, 4, 6, 8}
};

Q.2: Define a one-dimensional, six-element floating-point array called **consts.** Assign the following values to the array elements: 0.005, –0.032, 1e–6, 0.167, –0.3e8, 0.015

Q.3: Explain the meaning of each of the following declarations

a) float a, b;
float *pa, *pb;

b) float a = –0.167;
float *pa = &a;

c) char *d[4] = {"north'', ''south", "east", "west"};

1.6.3 Structure

Structure is the most important user defined data type in C. A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. But in an array, all the data items are of same type. The individual variables ina structure are called

member variables. A structure is a convenient way of grouping several pieces of related information together.

Here is an example of a structure definition.

```
struct student
  {
     char name[25];
     char course[20];
     int age;
     int year;
  };
```

Declaration of a structure always begins with the key word **struct** followed by a user given name, here the **student**. After the opening brace there will be the member of the structure containing different names of the variables and their data types followed by the closing brace and the semicolon.

Graphical representation of the structure student is shown below:

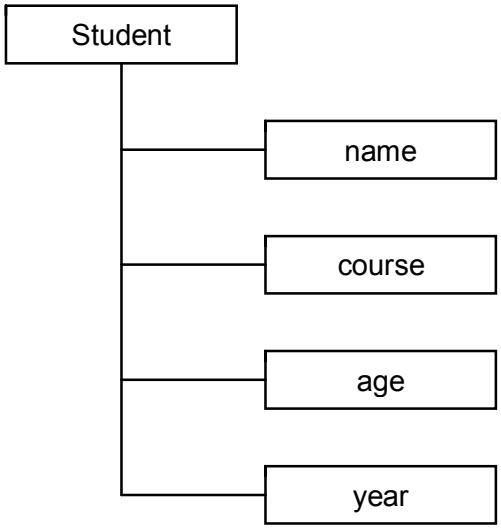


Figure 1.9: A structure named Student

Now let us see how to declare a structure variable. In the following we have declare a variable **st_rec** of type student:

```
student st_rec[100];
```

In this declaration **st_rec** is a 100 element array of structures.

Hence each element of **st_rec** is a separate structure of type student (i.e. each element of **st_rec** represents an individual student record.)

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed. In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. As an example if we want to access the name of the 10th student (i.e., **st_rec[9]** since the subscript begins with 0) from the above structure then we will have to write,

st_rec[9].name

Similarly, course and age of the 10th student can be accessed by writing,

st_rec[9].course and **st_rec[9].age**

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. Example of assigning the values for the 10th student record is shown in the following:

struct student **st_rec[9]** = {“Arup Deka”, “BCA”, 21, 2008};

Program 1.5: Program showing use of structure data type

```
#include<stdio.h>
#include<conio.h>
void main( )
   {
   struct book
      {
         char name;
         float price;
         int pages;
      };
   struct book b1, b2, b3;
```

```
clrscr();
printf (''\nEnter names, prices & no. of pages
of 3 books\n'')
scanf (''%c %f %d'', &b1.name, &b1.price,
&b1.pages);
scanf (''%c %f %d'', &b2.name, &b2.price,
&b2.pages);
scanf (''%c %f %d'', &b3.name, &b3.price,
&b3.pages);
printf (''\nAnd this is what you entered'');
printf (''\n%c %f %d'', b1.name, b1.price,
b1.pages);
printf (''\n%c %f %d'', b2.name, b2.price,
b2.pages);
printf (''\n%c %f %d'', b3.name, b3.price,
b3.pages);
getch();
}
```

Self-referencial Structure: When a member of a structure is declared as a pointer that points to the same structure (parent structure), then it is called a *self-referential structure*. It is expressed as shown below:

```
struct node
    {
        int data;
        struct node *next;
    };
```

where '**node**' is a structure that consists of two members one is the data item and other is a pointer variable holding the memory address of the other node. The pointer variable **next** contains an address of either the location in memory of the successor node or the special value **NULL**.

Self-referential structures are very useful in applications that involves linked data structures such as linked list and trees.

The basic idea of a linked data structure is that each component wiithin the structure includes a pointer indicating where the next component can be found. Therefore, the relative order of the components can easily be changed simply by altering the pointers. In addition, individual components can easily be added or deleted again by altering the pointers.

The diagramatic representation of a node is shown below:

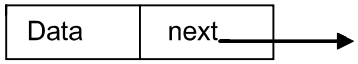


Figure 1.10: A node structure

1.7 DYNAMIC MEMORY ALLOCATION

The memory allocation process may be classified as static allocation and dynamic allocation. In static allocation, a fixed size of memory are reserved before loading and execution of a program. If that reserved memory is not sufficient or too large in amount then it may cause failure of the program or wastage of memory space. Therefore, C language provides a technique, in which a program can specify an array size at run time. The process of allocating memory at run time is known as *dynamic memory allocation*.

There are three dynamic momory allocation functions and one memory deallocation (releasing the memory) function. These are **malloc( ), calloc( ), realloc( )** and **free( )**.

malloc( ): The function *malloc( )* allocates a block of memory. The *malloc( )* function reserves a block of memory of specified size and returns a pointer of type void. The reserved block is not initialize to zero. The syntax for usinig *malloc( )* function is:

ptr = (cast-type *) malloc(byte-size);

where *ptr* is a pointer of type cast-type. The *malloc( )* returns a pointer (of cast-type) to an area of memory with size byte-size.

Suppose x is a one dimensional integer array having 15 elements.

It is possible to define *x* as a pointer variable rather than an array. Thus, we write,

int *x;

instead of int x[15] or #define size 15
int x[size];

When *x* is declared as an array, a memory block having the capacity to store 15 elements will be reserved in advance. But in this case, when *x* is declared as a pointer variable, it will not assigned a memory block automatically.

To assign sufficient memory for *x*, we can make use of the library function *malloc*, as follows:

x = (int *)malloc(15 * sizeof(int));

This function reserves a block of memory whose size (in bytes) is equivalent to 15 times the size of an integer. The address of the first byte of the reserved memory block is assigned to the pointer *x* of type **int**. Diagammatic representation is shown below in figure 1.11.

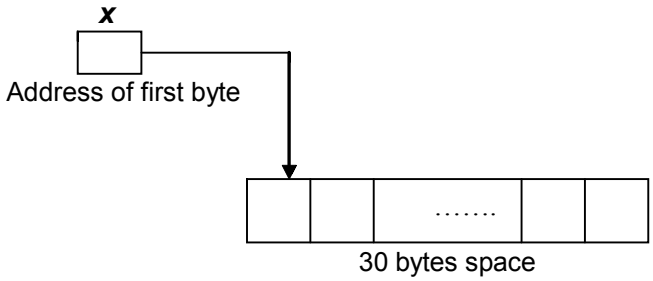


Figure 1.11: Representation of dynamic memory allocation

calloc( ): *calloc* is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. The main difference between the *calloc* and *malloc* function is that *malloc* function allocates a single block of storage space while the *calloc* function allocates a multiple blocks of storage space having the same size and intialize the allocated bytes to zero. The syntax for usinig *calloc( )* function is:

ptr = (cast-type *) calloc (n, element-size)

Where, n is the number of contiguous blocks to be allocate to each of having the size *element-size*.

realloc( ): *realloc( )* function is used to change the size of the previouslly allocated memory blocks. If the previously allocated memory is not sufficient or much larger and we need more space for more elements or we need reduced space for less elements then by the using the realloc function block size can be maximize or minimize. The syntax for usinig *realloc( )* function is:

ptr = realloc(ptr, newsize)

where, *newsize* is the size of the memory space to be allocate.

free( ): It is necessary to free the memory allocated so that the memory can be reused. The free( ) function frees up (deallocates) memory that was previously allocated with malloc( ), calloc( ) or realloc( ). The syntax for usinig *free( )* function is:

free(ptr)

where, *ptr* is apointer to a memory block, which has already been created by *malloc* or *calloc*.

1.8 ABSTRACT DATA TYPES

By now, you are well acquainted with data types, like integers, arrays, and so on. To access the data, you have used operations defined in the programming language for the data type. For example, array elements are accessed by using the square bracket notation, or scalar values are accessed simply by using the name of the corresponding variables.

This approach doesn't always work on large and complex programs in the real world. A modification to a program commonly requires a change in one or more of its data structures. It is the programmers responsibility to create special kind of data types. The programmer needs to define everything related to the new data type such as:

- how the data values are stored,
- the possible operations that can be carried out with the custom data type and
- new data type should be free from any confusion and must behave like a built-in type

Such custom data types are called **Abstract Data Types**.

Thus, an abstract data type is a formal specification of the logical properties of a data type such as its values, operations that are to be defined for the data type etc. It hides the detailed implementation of the data type and provides an interface to manipulate them.

Examples of abstract data types are– stacks, queues etc. We will discuss on these abstract data types in the next units.



CHECK YOUR PROGRESS

Q.4: Define a structure named complex having two floating point members *real* and *imaginary*. Also declare a variable x of type complex and assign the initial vaules 3.25 and –2.25.

Q.5: Declare a one dimensional, 75 element array called *sum* whose elements are stucture of type complex (declared in the above question).

Q.6: Define a self referential structure named *team* with the following three members :

a) a character array of 30 elements called name

b) an integer called age

c) a pointer to another strucutre of this same type, called next.

Q.7: a) The free( ) function is used to:

i) release the memory ii) destroy the memory

iii) create a link iv) none of the above

b) The data type defined by the user is known as

i) abstract data type ii) classic data type

iii) built-in data type iv) all of the above



1.9 LET US SUM UP

- The *data* is a fact about people, places or some entities. In computers, data is simply the value assigned to a variable.
- *Information* is a group of related data conveying some meaning.
- Data structures are of two types: linera data structure and non linear data structure. For example, Array, Linked list, Queue etc. are linear datastructure and Trees and Graphs etc are non-linear data structure.
- The possible data structure operations are– traversing, searching, inserting, deleting, sorting and merging.
- An array is a collection of two or more adjacent memory locations containing same types of data.
- A pointer is a memory variable that stores a memory address of another variable. It can have any name that is valid for other variable and it is declared in the same way as any other variable. It is always denoted by '*'.
- A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure.
- A strcture which contains a member field that point to the same structure type are called a self-referential structure.
- There is a technique in C language, in which a program can specify an array size at run time. The process of allocating memory at run time is known as dynamic memory allocation. There are three dynamic momory allocation functions: **malloc( ), calloc( ), realloc( )** and one memory deallocation function which is **free( )**.
- an abstract data type is a formal specification of the logical properties of a data type such as its values, operations that are to be defined for the data type etc.



1.10 FURTHER READING

1) Kamthane, Ashok N. *Introduction to Data Structures in C*. Perason Education.
2) Langsam, Yedidyah; Augenstein, Moshe J. & Tenenbaum, Aaron M. *Data Structures Using C and C++*. Prentice-Hall India.
3) Lipschutz, Seymour. *Data Structures*. Schaum's Outline Series in Computers. Tata McGraw Hill.



1.11 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) flag[0] = 'T', flag[1] = 'R', flag[2] = 'U', flag[3] = 'E' and flag[4] is assigned to zero.

b) p[0][0] = 1, p[0][1] = 3, p[0][2] = 5, p[0][3] = 7, p[1][0] = 2, p[1][1] = 4, p[1][2] = 6, p[1][3] = 8

Ans. to Q. No. 2: float consts[6] = {0.005, –0.032, 1e-6, 0.167, –0.3e8, 0.015}

Ans. to Q. No. 3: a) a and b are floating point variables, pa and pb are pointers to floating point quantities (though not necessarily to a and b)

b) a is a floatinig point variable whose initial value is –0.167; pa is a pointer to a floating point quantity, the address of a is assigned to pa as an intial value.

c) d is a one dimensional array of pointers to the string 'north', 'south', 'east' and 'west'.

Ans. to Q. No. 4:

```
struct complex
    {
        float  real;
        float imaginary;
    };
struct complex x = { 3.25, –2.25 }
```

Ans. to Q. No. 5: struct complex sum[75];

Ans. to Q. No. 6: struct team

```
{
    char name[30];
    int age;
    struct team *next;
};
```

Ans. to Q. No. 7: a) i.; b) i.



1.12 MODEL QUESTIONS

Q.1: What is information? Explain with few examples.

Q.2: What is data? Explain with few examples.

Q.3: Name and describe the four basic data types in C.

Q.4: What is a data structure? Why is an array called a data structure?

Q.5: How does a structure differ from an array? How is a structure member accessed?

Q.6: What is a pointer? How is a pointer initialized?

Q.7: What is dynamic memory allocation? How is it useful?

Q.8: What is the principal difference between the functions *malloc()* and *calloc().*

Q.9: Define and explain abstract data type?

Q.10: What is a self-referential structure? For what kind of applications are self referential structures useful?

Q.11: Write short notes on:

i) dynamic memory allocation ii) abstract data type

iii) pointers iv) structures

v) self-referential structure

*** ***** ***

UNIT 2: ALGORITHM

UNIT STRUCTURE

2.1 Learning Objectives

2.2 Introduction

2.3 Algorithm

2.4 Complexity

- 2.4.1 Space Complexity
- 2.4.2 Time Complexity

2.5 Asymptotic Notation

- 2.5.1 Theta Θ Notation
- 2.5.2 Big O (O) Notation
- 2.5.3 Big Omega (Ω) Notation

2.6 Let Us Sum Up

2.7 Further Reading

2.8 Answers to Check Your Progress

2.9 Model Questions

2.1 LEARNING OBJECTIVES

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

- define algorithm
- describe the concept of time complexity and space complexity
- describe the notations for defining the complexity of algorithm
- calculate time complexity of algorithm.

2.2 INTRODUCTION

In the previous unit, we have learnt about the different types of data structures like array, pointers and structures. These data structures are used to solve various problems. Algorithm exists for many common problems, but designing an efficient algorithm is a challenge and it plays a crucial role in large scale computer system. The concept of an algorithm is the basic need of any programming development in computer science. In this unit we

will discuss about algorithms and its space and time complexity. We will also discuss the asymptotic notation of algorithms. In the next unit, we will explore the concept of linked list and also discuss different types of linked lists.

2.3 ALGORITHM

An algorithm is a well-defined computational method which takes some value(s) as input and produces some value(s) as output. In other words, an algorithm is a sequence of computational steps that transforms input(s) into output(s).

Each algorithm must have:

- **Specification:** Description of the computational procedure
- **Pre-conditions:** The condition(s) on input
- **Body of the Algorithm:** A sequence of clear and unambiguous instructions
- **Post-conditions**: The condition(s) on output

Consider a simple algorithm for finding the factorial of **n**.

Algorithm 2.1: Factorial (n)

```
Step 1:  FACT = 1
Step 2:  for i = 1 to n do
Step 3:  FACT = FACT*i
Step 4:  print FACT
```

In the above algorithm we have:

Specification: Computes *n*!

Pre-condition: *n*> = 0

Post-condition: FACT = *n*!

Now, let us take one more example.

Problem Definition: Sort given *n* numbers by non-descending order by using insertion sort.

Input: A sequence of n numbers $<a_1, a_2, a_3, \ldots, \ldots, a_n>$.

Output: A permutation (reordering) $<a'_1, a'_2, a'_3, \ldots, \ldots, a'_n>$ of input sequences such that $a'_1 = a'_2 = a'_3 = \ldots = a'_n$.

Insertion sort is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing

cards. We start with an empty left hand and the cards are face down on the table. Then we draw one card at a time from the table and place into correct position in the left hand.

Consider the following example of five integers:

79 43 39 58 13

Here we assume that the array has only one element that is 79 and it is sorted. So the array is:

79 43 39 58 13

Next we will take 43, since 43 is less than 79 so it will be placed before 79. After placing 43 into its place the array will be:

43 79 39 58 13

Next we will take 39, since 39 is less than 43 and 79 so it will be placed before 43. After placing 39 into its place the array will be:

39 43 79 58 13

Next we will take 58, since 58 is less than 79 but grater then 39 and 49 so it will be placed before 79 but after 43. After placing 58 into its place the array will be:

39 43 58 79 13

Finally 13 will be considered, since 13 is smaller than all other elements so it will be placed before 39. After placing 13 the sorted array will be:

13 39 43 58 79

Here in insertion sort, we consider that first (i–1) numbers are sorted then we try to insert the i^{th} number into its correct position. This can be done by shifting numbers right one number at a time until the position for i^{th} number is found.

The algorithmic description of insertion sort is given below.

Algorithm 2.2: Insertion_Sort (a[n])

```
Step 1:  for i = 2 to n do
Step 2:  current_num = a[i]
Step 3:  j = i
Step 4:  while ((j>1) and (a[j–1]>current_num)) do
Step 5:  a[j] = a[j–1]
Step 6:  j = j–1
Step 7:  a[j] = current_num
```

2.4 COMPLEXITY

Once we develop an algorithm, it is always better to check whether the algorithm is efficient or not. The efficiency of an algorithm depends on the following factors:

- Accuracy of the output
- Robustness of the algorithm
- User friendliness of the algorithm
- Time required to run the algorithm
- Space required to run the algorithm
- Reliability of the algorithm
- Extensibility of the algorithm

In case of complexity analysis, we mainly concentrate on the time and space required by a program to execute. So complexity analysis is broadly categorized in two classes

- Space complexity
- Time complexity

2.4.1 Space Complexity

Though computer memory is less costly nowadays, it is very much important to analyze the amount of memory used by a program. This is because if the algorithm takes memory beyond the capacity of the machine, then the algorithm will not be able, to successfully execute. So, it is very much important to analyze the space complexity before we execute it on the computer.

The space complexity of an algorithm is the amount of main memory needed to run the program till completion.

To measure the space complexity in absolute memory unit there are the following problems:

1) The space required for an algorithm depends on space required by the machine during execution, they are
   i) Programme space
   ii) Data space

2) The programme space is fixed and it is used to store the temporary data, object code, etc.
3) The data space is used to store the different variables, data structures defined in the program.

In case of analysis we consider only the data space, since programme space is fixed and depends on the machine where it is executed.

Consider the following algorithms for exchange of two numbers:

Algorithm 2.3: Exchange_1(a, b)

Step 1: temp = a;
Step 2: a = b;
Step 3: b = temp;

Algorithm 2.4: Exchange_2(a, b)

Step 1: a = a + b;
Step 2: b = a – b;
Step 3: a = a – b;

The first algorithm uses three variables a, b and temp and the second algorithm takes only two variables. So, if we look from the space complexity perspective the second algorithm is better than the first one.

2.4.2 Time Complexity

The time complexity of an algorithm is the amount of computer time it needs to run the program till completion. To measure the time complexity in absolute time unit there are the following problems:

1) The time required for an algorithm depends on number of instructions executed by the algorithm.
2) The execution time of an instruction depends on the computer's power. Since, different computers take different amount of time for the same instruction.

3) Different types of instructions take different amount of time on the same computer.

For time complexity analysis we design a machine by removing all the machine dependent factors called Random Access Machine (RAM). The random access machine model of computation was devised by John von Neumann to study algorithms. The design of RAM is as follows:

1) Each "simple" operation (+, –, =, if, call) takes exactly 1 step.
2) Loops and subroutine calls are not simple operations. They depend upon the size of the data and the contents of a subroutine.
3) Each memory access takes exactly 1 step.

Consider the following algorithm for add two number:

Algorithm 2.5: Add_Odd(a, b)

```
Step 1:  C = a + b;
Step 2:  return C;
```

Here this algorithm has only two simple statements so the complexity of this algorithm is 2.

Consider another algorithm for add n even number:

Algorithm 2.6: Add_Even(n)

```
Step 1:  i = 2;
Step 2:  sum = 0;
Step 3:  while i <= 2*n
Step 4:  sum = sum + i
Step 5:  i = i + 2;
Step 6:  end while;
Step 7:  return sum;
```

Here, in algorithm 2.6, Step 1, Step 2 and Step 7 are simple statement and they will execute only once. Step 3 is a loop statement and it will execute as many times as the loop condition is true and one more time for check if the condition is false. Step 5 and Step 6 are inside the loop so it will run as much as the loop condition is true. Step 6 just indicates the end of while and no cost is associated with it.

Statement	Cost	Frequency	Total cost
Step 1: i = 2;	1	1	1
Step 2: sum = 0;	1	1	1
Step 3: while i <= 2*n	1	n + 1	n + 1
Step 4: sum = sum + i	1	n	n
Step 5: i = i + 2;	1	n	n
Step 6: end while;	0	1	0
Step 7: return sum;	1	1	1
Total cost			**3n + 4**



CHECK YOUR PROGRESS

Q.1: State True or False:

a) Time complexity is the time taken to design an algorithm.

b) Space complexity is the amount of space required by a program during execution.

c) An algorithm may not produce any output.

d) Algorithms are computer programs which can be directly run into the computer.

e) If an algorithm is designed for a problem then it will work all the valid inputs for the problem.

2.5 ASYMPTOTIC NOTATION

When we calculate the complexity of an algorithm we often get a complex polynomial. To simplify this complex polynomial we use some notation to represent the complexity of an algorithm called asymptotic notation.

2.5.1 Theta (Θ) Notation

For a given function g(n), Θ(g(n)) is defined as:

$$\Theta(g(n)) = f(n): \begin{cases} \text{there exist constants } c_1 > 0,\ c_2 > 0 \text{ and } n_0 \in N \\ \text{such that } 0 \le c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words, a function f(n) is said to belong to Θ(g(n)), if there exists positive constants c_1 and c_2 such that $0 \le c_1\, g(n) \le f(n) \le c_2 g(n)$ for sufficiently large value of n. Figure 2.1 gives a intuitive picture of functions f(n) and g(n), where f(n) = Θ(g(n)). For all the values of n at and to right of n_0, the values of f(n) lies at or above $c_1 g(n)$and at or below $c_2 g(n)$. In other words, for all $n \ge n_0$, the function f(n) is equal to g(n) to within a constant factor. So, g(n) is said to be **asymptotically tight bound** for f(n).

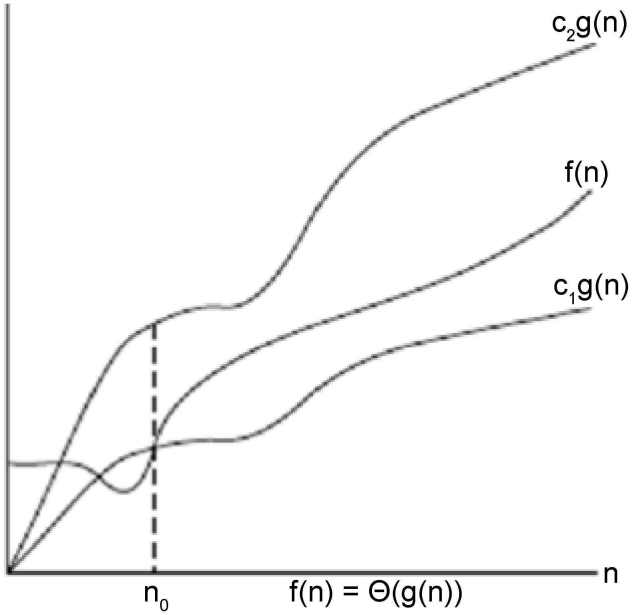


Figure 2.1: Graphic example of Θ notation

For example, $f(n) = ½n^2 – 3n$

Let $g(n) = n^2$

to proof f(n) = Θ(g(n)) we must determine the positive constants c_1, c_2 and n_0 such that,

$c_1 n^2 \le ½n^2 – 3n \le c_2 n^2$ for all $n = n_0$

Dividing the whole equation by n^2, we get,

$c_1 \le ½ – 3/n \le c_2$

We can make the right hand inequality hold for any value of $n \ge 1$ by choosing $c_2 \ge ½$. Similarly we can make the left hand inequality hold for any value of n = 7 by choosing $c_1 \le 1/14$. Thus, by choosing $c_1 = 1/14$, $c_2 = ½$. And $n_0 = 7$, we can have, f(n) = Θ(g(n)).

That is $½n^2 – 3n = Θ(n^2)$.

2.5.2 Big O (O) Notation

For a given function g(n), O(g(n)) is defined as:

$$O(g(n)) = f(n): \begin{cases} \text{there exist constants } c > 0, \text{ and } n_0 \in N \\ \text{such that } 0 \le f(n) \le c\, g(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words a function f(n) is said to belong to O(g(n)), if there exists positive constant c such that $0 \le f(n) \le c\, g(n)$ for sufficiently large value of n. Figure 2.2 gives a intuitive picture of functions f(n) and g(n), where f(n) = O (g(n)). For all the values of n at and to right of n_0, the values of f(n) lies at or below cg(n). So, g(n) is said to be an **asymptotically upper bound** for f(n).

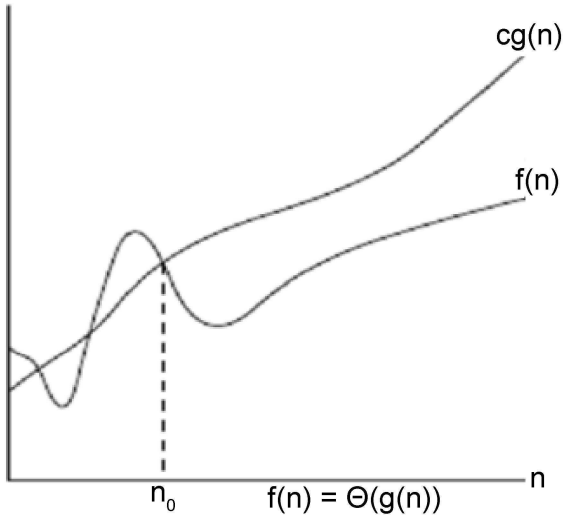


Figure 2.2: Graphic example of O notation

2.5.3 Big Omega (Ω) Notation

For a given function g(n), Ω(g(n)) is defined as:

$$\Omega(g(n)) = f(n): \begin{cases} \text{there exist constants } c > 0, \text{ and } n_0 \in N \\ \text{such that } 0 \le c\, g(n) \le f(n) \text{ for all } n \ge n_0 \end{cases}$$

In other words, a function f(n) is said to belong to Ω(g(n)), if there exists positive constant c such that $0 \le c\, g(n) \le f(n)$ for sufficiently large value of n. Figure 2.3 gives a intuitive picture of functions f(n) and g(n), where f(n) = Ω(g(n)). For all the values of n at and to right of n_0, the values of f(n) lies at or above cg(n). So, g(n) is said an asymptotically lower bound for f(n).

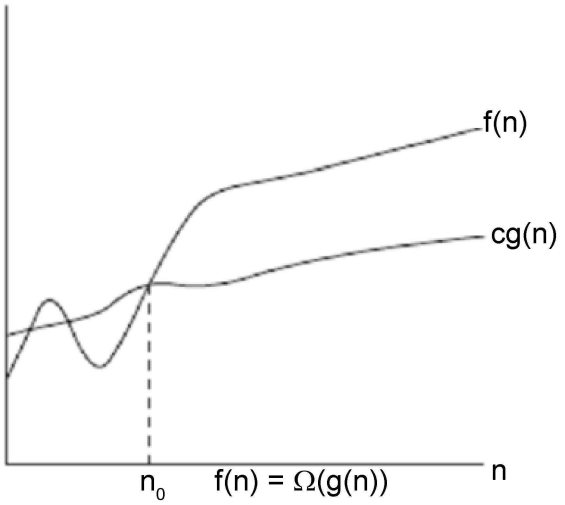


Figure 2.3: Graphic example of Ω notation

The growth patterns of order notations have been listed below:

$O(1) < O(\log(n)) < O(n) < O(n \log(n)) < O(n^2) < O(n^3) \ldots < O(2^n)$.

The common name of few order notations is listed below:

Notation	Name
O(1)	Constant
O(log(n))	Logarithmic
O(n)	Linear
O(nlog(n))	Linearithmic
$O(n^2)$	Quadratic
$O(c^n)$	Exponential
O(n!)	Factorial

A comparison of typical running time of different order notations for different input size is listed below:

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1014	32768	4294967296

Now let us take few examples of the above asymptotic notations.

Example 2.1: Prove that $3n^3 + 2n^2 + 4n + 3 = O(n^3)$

Here, $f(n) = 3n^3 + 2n^2 + 4n + 3$

$g(n) = O(n^3)$

to proof $f(n) = O(g(n))$ we must determine the positive constants c and n_0 such that, $3n^3 + 2n^2 + 4n + 3 \leq cn^3$ for all $n \geq n_0$

dividing the whole equation by n^3, we get,

$3 + 2/n + 4/n^2 + 3/n^3 \leq c$

We can make the inequality hold for any value of n = 1 by choosing $c \geq 12$. Thus, by choosing $c \geq 12$ and $n_0 = 1$, we can have,

$f(n) = O(g(n))$.

Thus, $3n^3 + 2n^2 + 4n + 3 = O(n^3)$.

Example 2.2: Prove that, $3n^3 + 2n^2 + 4n + 3 = \Omega(n^3)$

Here, $f(n) = 3n^3 + 2n^2 + 4n + 3$

$g(n) = O(n^3)$

to proof $f(n) = \Omega(g(n))$ we must determine the positive constants c and n_0 such that,

$c\, n^3 \leq 3n^3 + 2n^2 + 4n + 3$ for all $n \geq n_0$

dividing the whole equation by n^3, we get,

$c \leq 3 + 2/n + 4/n^2 + 3/n^3$

We can make the inequality hold for any value of $n \geq 1$ by choosing $c \leq 3$. Thus, by choosing c = 3 and $n_0 = 1$ we can have,

$f(n) = \Omega(g(n))$.

Thus, $3n^3 + 2n^2 + 4n + 3 = \Omega(n^3)$.

Example 2.3: Prove that $7n^3 + 7 = \Theta(n^3)$

Here, $f(n) = 7n^3 + 7$

$g(n) = O(n^3)$

to proof $f(n) = \Theta(g(n))$ we must determine the positive constants c_1, c_2 and n_0 such that,

$c_1 n^3 \leq 7n^3 + 7 \leq c_2\, n^3$ for all $n \geq n_0$

dividing the whole equation by n^3, we get,

$c_1 \leq 7 + 7/n^3 \leq c_2$

We can make the right hand inequality hold for any value of $n \geq 1$ by choosing $c_2 \geq 14$. Similarly we can make the left hand inequality hold

for any value of $n \geq 1$ by choosing $c_1 \leq 7$. Thus, by choosing $c_1 = 7$, $c_2 = 14$. And $n_0 = 1$ we have $f(n) = \Theta(g(n))$.

Thus, $7n^3 + 7 = \Theta(n^3)$.

Now let us take few examples of algorithms and represent their complexity in asymptotic notations.

Example 2.4: Consider the following algorithm to find out the sum of all the elements in an array:

Statement	**Cost**	**Frequency**	**Total cost**
Sum_Array(arr[], n)			
Step 1: i = 0;	1	1	1
Step 2: s = 0;	1	1	1
Step 3: while i < n	1	n + 1	n + 1
Step 4: s = s + arr [i]	1	n	n
Step 5: i = i + 1;	1	n	n
Step 6: end while;	0	1	0
Step 7: return s;	1	1	1
Total Cost:			**3n + 4**

Here, $f(n) = 3n + 4$

Let, $g(n) = n$

If we want to represent it in O notation then we have to show that for some positive constant c and n_0

$0 \leq f(n) \leq cg(n)$

$\Rightarrow 0 \leq 3n + 4 \leq cn$

Now if we take n = 1 and c = 7

$\Rightarrow 0 \leq 3 \times 1 + 4 \leq 7 \times 1$

Which is true, so we can say that for $n_0 = 1$ and c = 7

$f(n) = O(g(n))$ that is $3n + 4 = O(n)$.

Example 2.5: Consider the following algorithm to add two square matrixes.

Statement	Cost	Frequency	Total cost
Mat_add(a[], n, b[])			
Step 1: i = 0	1	1	1
Step 2: j = 0;	1	1	1
Step 3: while i < n	1	n + 1	n + 1
Step 4: while j < n	1	n(n + 1)	n(n + 1)
Step 5: c[i][j] = a[i][j] + b[i][j]	1	n*n	n*n
Step 6: j = j + 1	1	n*n	n*n
Step 7: end inner while;	0	n	0
Step 8: i = I + 1	1	n	n
Step 9: end outer while	0	1	0
Step 10: return c;	1	1	1
Total Cost:			**$3n^2 + 3n + 4$**

Here, $f(n) = 3n^2 + 3n + 4$

Let, $g(n) = n^2$

If we want to represent it in Ω notation then we have to show that for some positive constant c and n_0

$0 \leq cg(n) \leq f(n)$

$\Rightarrow 0 \leq cn^2 \leq 3n^2 + 3n + 4$

Now if we take n = 1 and c = 3

$\Rightarrow 0 \leq 3 \times 1 \leq 3 \times 1^2 + 3 \times 1 + 4$

Which is true, so we can say that for $n_0 = 1$ and c = 3

$f(n) = \Omega(g(n))$ that is $3n^2 + 3n + 4 = O(n^2)$.

In the analysis of algorithm we may have three different cases depending on the input to the algorithm. They are:

Worst Case: Worst case execution time is an upper bound for execution time with any input. It guarantees that, irrespective of the type of input, the algorithm will not take any longer than the worst case time.

Average case: This gives the average execution time of algorithm. Average case execution time is the execution time taken by an algorithm in average for any random input to the algorithm.

Best Case: Best case execution time is the lower bound for execution time with any input. It guarantees that under any circumstances the execution time of the algorithms will be at best case execution time.

Example 2.6: Consider the following Insertion sort algorithm:

Insertion_Sort(a[n]) Algorithm

```
Step 1:  i = 2
Step 2:  while i < n
Step 3:  num = a[i]
Step 4:  j = i
Step 5:  while ((j >1) && (a[j–1] > num))
Step 6:  a[j] = a[j–1]
Step 7:  j = j – 1
Step 8:  end while (inner)
Step 9:  a[j] = num
Step 10: i = i + 1
Step 11: end while (outer)
```

Worst case Analysis of Insertion Sort: In the *worst case* inputs to the algorithm will be reversely sorted. So the loop statement will run for maximum time. In worst case in every time we will find a[j – 1] > num in line 5 as true. So this statement will run for 2 + 3 + 4 + … + n times total n(n + 1) – 1 times. Statement 6 will run for 1 + 2 + 3 + … + n – 1 times total n(n – 1) times. It is the same for statement 7.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n – 1	n – 1
Step 4	1	n – 1	n – 1
Step 5	1	n(n + 1) –1	n(n + 1) – 1
Step 6	1	n(n – 1)	n(n – 1)
Step 7	1	n(n – 1)	n(n – 1)
Step 8	0	n – 1	0
Step 9	1	n – 1	n – 1

Step 10	1	n – 1	n – 1
Step 11	0	1	0
Total Cost			**$3n^2 + 4n – 4$**

Here, $f(n) = 3n^2 + 4n – 4$

Let, $g(n) = n^2$

If we want to represent it in O notation then we have to show that for some positive constant c and n_0

$0 \leq f(n) \leq cg(n)$

$\Rightarrow 0 \leq 3n^2 + 4n – 4 \leq cn^2$

Now if we take n = 1 and c = 7

$\Rightarrow 0 \leq 3 \times 1^2 + 4 \times 1 – 4 \leq 7 \times 1^2$

Which is true, so we can say that for $n_0 = 1$ and c = 7

$f(n) = O(g(n))$ that is $3n^2 + 4n – 4 = O(n^2)$.

So worst case time complexity of insertion sort is $O(n^2)$.

Average case Analysis of Insertion Sort: In *average case,* inputs to the algorithm will be random. In average case, half of the time we will find a[j - 1] > num is true and false in other half. So, statement 5 will run for (2 + 3 + 4 + … + n)/2 times total (n(n + 1) – 1)/2 times. Statement 6 will run for (1 + 2 + 3 + … + n – 1)/2 times total (n(n – 1))/2 times. Similarly, for statement 7.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n – 1	n – 1
Step 4	1	n – 1	n – 1
Step 5	1	(n(n + 1) – 1)/2	(n(n + 1) – 1)/2
Step 6	1	(n(n – 1))/2	(n(n – 1))/2
Step 7	1	(n(n – 1)/2	(n(n – 1))/2
Step 8	0	n – 1	0
Step 9	1	n – 1	n – 1
Step 10	1	n – 1	n – 1
Step 11	0	1	0
Total Cost			**$^3/_2\, n^2 + {}^7/_2\, n – 4$**

Similarly we can show that average case time complexity of insertion sort is $O(n^2)$.

Best case Analysis of Insertion Sort: In the *best case* inputs will be already sorted. So a[j–1] > num will be false always. So statement 5 will run for n times (only to check the condition is false). Statement 6 will run for 0 times since while loop will be false always. It is the same for statement 7.

Statement	Cost	Frequency	Total cost
Step 1	1	1	1
Step 2	1	n	n
Step 3	1	n – 1	n – 1
Step 4	1	n – 1	n – 1
Step 5	1	n	n
Step 6	1	0	0
Step 7	1	0	0
Step 8	0	n – 1	0
Step 9	1	n – 1	n – 1
Step 10	1	n – 1	n – 1
Step 11	0	1	0
Total Cost			**5n – 3**

Similarly we can show that best case time complexity of insertion sort is O(n).



CHECK YOUR PROGRESS

Q.2: State True or False:

a) $7n^3 + 4n + 27 = O(n^3)$ b) $2n^2 + 34 = \Omega(n^3)$

c) $2n^2 + 34 = O(n^3)$ d) $2n^2 + 34 = \Theta(n^3)$

e) $2n^2 + 34 = \Omega(n)$ f) $2n^2 + 34 = \Theta(n^2)$

g) $2n^7 + 4n^3 + 2n = \Omega(n^3)$ h) $2n^4 + 3n^3 + 17n^2 = O(n^3)$

Q.3: Define time complexity.

Q.4: Define space complexity.



2.6 LET US SUM UP

- An algorithm is a sequence of computational steps that start with a set of input(s) and finish with valid output(s)
- An algorithm is correct if for every input(s), it halts with correct output(s).
- Computational complexity of algorithms are generally referred to space complexity and time complexity of the program
- The space complexity of an algorithm is the amount of main memory needed to run the program till completion.
- The time complexity of an algorithm is the amount of computer time it needs to run the program till completion.
- $O(1) < O(\log(n)) < O(n) < O(n \log(n)) < O(n^2) < O(n^3) \ldots < O(2^n)$.



2.7 FURTHER READING

1) Aho, A. V. & Hopcroft, J. E. (1999). *The Design and Analysis of Computer Algorithms*. Pearson Education India.
2) Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*, MIT Press.
3) Horowitz, E., Sahni, S. & Rajasekaran, S. (2009). *Fundamental of Data Structure in C*.



2.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) False, b) True, c) False, d) False, e) True

Ans. to Q. No. 2: a) True, b) False, c) True, d) False, e) True, f) True, g) True, h) False

Ans. to Q. No. 3: The time complexity of an algorithm is the amount of computer time it needs to run the program till completion.

Ans. to Q. No. 4: The space complexity of an algorithm is the amount of main memory is needed to run the program till completion.



2.9 MODEL QUESTIONS

Q.1: Given an array of n integers, write an algorithm to find the smallest element. Find the number of instruction executed by your algorithm. What are the time and space complexities?

Q.2: Write an algorithm to find the median of n numbers. Find number of instruction executed by your algorithm. What are the time and space complexities?

Q.3: Write an algorithm to sort elements by bubble sort algorithm. What are the time and space complexities?

Q.4: Explain the need of analysis of algorithm.

Q.5: Prove the following:

i) $3n^5 - 7n + 4 = \Theta(n^5)$ ii) $^1/_3 n^4 - 7n^2 + 3n = \Theta(n^4)$

iii) $2n^2 + n + 4 = \Theta(n^2)$ iv) $3n^5 - 7n + 4 = O(n^5)$

v) $3n^5 - 7n + 4 = \Omega(n^5)$

*** ***** ***

UNIT 3: LINKED LIST

UNIT STRUCTURE

3.1 Learning Objectives

3.2 Introduction

3.3 Linked List

3.4 Types of Linked List

- 3.4.1 Singly Linked List
- 3.4.2 Doubly Linked List
- 3.4.3 Circular Linked List

3.5 Traversing a Single Linked List

3.6 Searching a Single Linked List

3.7 Insertion into a Single Linked List

3.8 Deletion from a Single Linked List

3.9 Reversing a Single Linked List

3.10 Let Us Sum Up

3.11 Further Reading

3.12 Answers to Check Your Progress

3.13 Model Questions

3.1 LEARNING OBJECTIVES

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

- define linked list
- describe the different types of linked list
- describe and write programs on the different operations on linked list like traversal, searching, insertion and deletion
- write programs for reversing a linked list.

3.2 INTRODUCTION

We are already familiar with the array data structure. Array is a linear and homogenous data structure. It is very simple to represent a list of elements by using the array but it has some limitations. Once the size of the array is

declared, it cannot be changed during program execution. The unused portion of an array also occupies the memory space. Insertion and deletion operation in an array is time consuming and it needs shifting of elements to rearrange the numbers in order.

We will learn about *linked list* in this unit. The different types of linked lists are also discussed in this unit. In this unit, we will concentrate on the basic operations that are performed on a linked list. We shall discuss the traversing and searching procedure in a single linked list. In addition, insertion and deletion techniques will also be discussed along with reversing a linked list. In the next unit, we will discuss about stacks and applications of stacks.

3.3 LINKED LIST

In general, a series of linearly arranged finite elements having the same type is called a List. The data elements are called nodes. An array is an example of List. In array, by the index the sequential pattern is maintained. We just use the index for accessing and manipulating array elements.

Like the arrays, a linked list is also a linear data structure consisting of nodes where each node has two parts:

i) data part

ii) next part (link part)

The data part can hold data of any type (i.e. int, char, float. ... etc.) and the next part holds the address of the next node. This type of list is called a Linked list because it is a list whose order is given by links from one item to the next. An individual node is shown in the following figure.

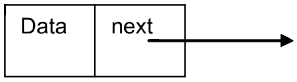


Figure 3.1: A node strucutre

A linked list is represented by a pointer (named 'START') to the first node and the *next* part of the last node is assigned the value 'NULL'. Diagrammatic representation of a linked list is shown below:

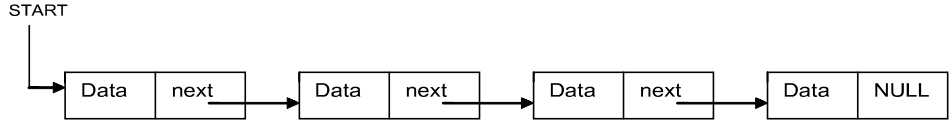


Figure 3.2: A linked list

In the above figure, the linked list contains a pointer variable called START, which contains the address of the first node in the list. Each node has two parts, a *data part* and a *next pointer*. The data part contains information which may be an integer, a character, a string etc., and *next* contains the information of pointers which holds the address of the next node. If the *next* has the value NULL, then it is the last node of the list. So, the last node can be identified by the presence of NULL.

A linked list is therefore a collection of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as part of the data in the structure itself. The link is in the form of a pointer to another structure of the same type. Such a structure is represented as follows:

```
struct node
    {
        int data;
        struct node *next;
    };
```

The first member is an integer data (it can be of any type) and the second member is a pointer to the next node. The next part or link part of a node actually contains memory address, where the data next to the current one is stored. It is also called a pointer to the next data. The structure, which contains a member field that points to the same structure type, is called a *self-referential structure*.

3.4 TYPES OF LINKED LIST

There are three types of linked list. These are:

- Singly linked list
- Doubly linked list
- Circular linked list

3.4.1 Singly Linked List

Singly linked list is a linked list in which two successive nodes of the list are linked with each other in sequential linear manner. The following figure depicts a singly linked list.

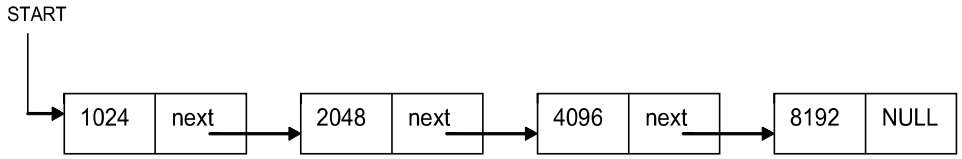


Figure 3.3: A singly linked list

In the figure, the first node is pointed by a pointer named 'START'. The NULL in the last node denotes the end of the list. 1024, 2048, 4096, 8192 are the data items of the list.

3.4.2 Doubly Linked List

Doubly linked list is another type of linked list, in which each node has double set of pointers along with the data part. One pointer is pointing to the next node (successor node) and the other pointer points to the preceding node (predecessor node). It means, in a doubly linked list addresses of next as well as preceding elements are linked with current node. This allows us to traverse the list either in forward direction or in backward direction. But in a singly linked list traverse is possible only in one direction. The following figure depicts a doubly linked list.

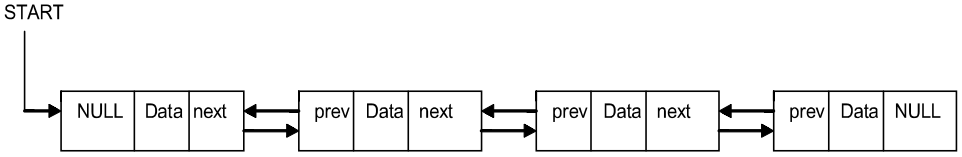


Figure 3.4: A doubly linked list

In the figure, each node has two pointers one representing the address of the successor node and the other representing the address of the predecessor node. Here, in the we have seen that there are two NULL pointers, one for the previous node and the other for the next node are being maintained. When traversing the list from left to right, it identifies the end of the list when *node* $\rightarrow$ *next* is NULL. Similarly, when traversing the list from right to left, the end of the list is indicated when *node* $\rightarrow$ *prev* is NULL. The declaration of a doubly linked list is shown below:

```
struct node
    {   struct node *prev;
        int data;
        struct node *next;
    };
```

3.4.3 Circular Linked List

Circular linked lists are the linked lists which are obtained by linking the last node of the linked list to the first node of the list. So, in a circular linked list, the last node does not contain the NULL pointer; instead it contains the address of the first node. The following figure depicts a circular linked list.

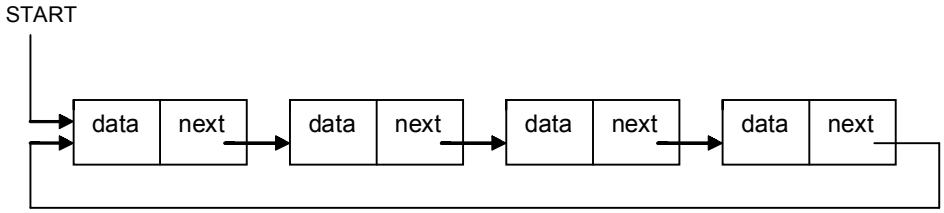


Figure 3.5: A circular linked list

One of the shortcomings of a singly linked list is that after reaching the last node in the list, we cannot go back to the previous node unless the address is available. Also, to reach the first node after reaching the end, there will be a need to traverse the whole list backward if pointers(i.e. addresses) are available. But in a circular linked list, each node can be accessed from any node. It means that the circular list can be traversed starting from any node.

3.5 TRAVERSING A SINGLE LINKED LIST

We have already learnt that linked list is a collection of elements called *node*. Each node has two components: the first component contains the *data* (or *information)* of the element and the second component called the *next field* (or *linked field*) which holds the address of the next node. To indicate the end of the list, we generally assign a NULL value. A node in a single linked list has the following structure:

```
struct node
    {
        int data;
        struct node *next;
    };
```

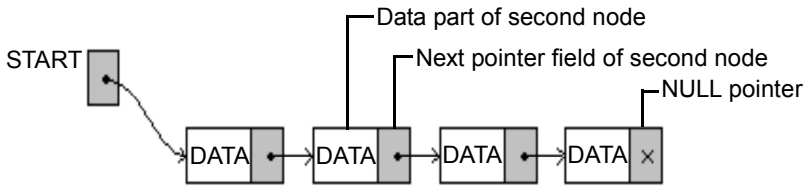


Figure 3.6: Linked List with four nodes

Here member of the structure struct node *link points to the structure itself.

Traversing a linked list: Traversing a linked list means to visit every node or element in the list starting from the first to the last element. For traversing a linked list we consider **ptr** as a pointer variable. The linked list also contains a pointer variable called **start** which contains the address of the first node in the list. **NULL** pointer denoted by x in the figure indicates the end of the list. Initially we assign the value of **start** to **ptr**. Thus, **ptr** will point to the first element of the list. For processing the next element we assign the address of the next element to the **ptr** as

```
ptr = ptr–>next;
```

Now, **ptr** holds the address of the next node of the list. We can traverse each element of the linked list through this assignment untill ptr has **NULL** address (which denotes the last node of the list). Thus, the the traversal of a linked list can be implemented with the following statements:

```
while(ptr!=NULL)
    ptr=ptr–>next;
```

3.6 SEARCHING A SINGLE LINKED LIST

Searching refers to the finding of an item or data from a given list of elements. So in case of a linked list searching means finding a specific node from a specific list. In searching operation we compare the KEY (item to be searched) value with each node value in the list. It can be implemented as:

```
while(ptr != NULL)
{
if(ptr –> data == key) //Key is the data be searched
        printf (“Item found in the list”);
    else
        ptr =ptr –> next;
}
```

3.7 INSERTION INTO A SINGLE LINKED LIST

There may be three possibilities for inserting a node depending upon the position of insertion. The three ways are:

- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between two nodes

Insertion at the beginning: Let us consider a pointer **temp** which points to the node that has to be inserted. The data part of the new node is assigned to the data (i.e., d) through the temp pointer as:

temp–>data=d;

Again, we know that the start pointer points to the first node of the linked list. Therefore, for inserting at the beginning we will have to assign start to the next part of temp i.e.,

temp–>next=start;

So that next part of the inserted node will point to the node which was originally in first position.

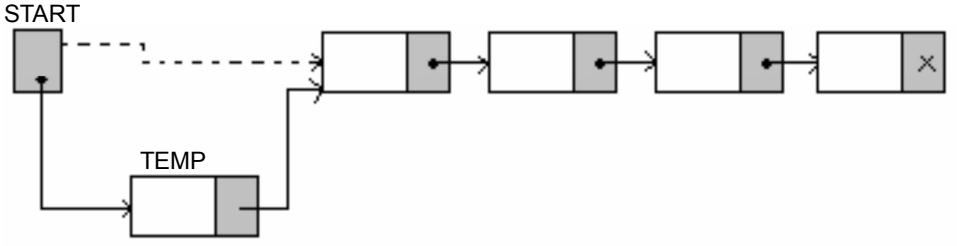


Figure 3.7: Insertion of node at the beginning

Now, inserted node points to the next node which was the first node of the linked list. So the inserted node is the first node of the linked list and start pointer is reassigned as:

start = temp;

So that start points to the newly inserted node.

Insertion at the end: Let us again consider a pointer **temp** which points to the node that has to be inserted. The data part of the new node is assigned to the data (i.e., d) through the temp pointer:

temp–>data=d;

We know that to add the new node at the end position we would need to traverse the whole list to reach the last node. Once we traverse the entire linked list and reach the last node the null value of the last node will be replaced by the temp so that it points to the node to be inserted.

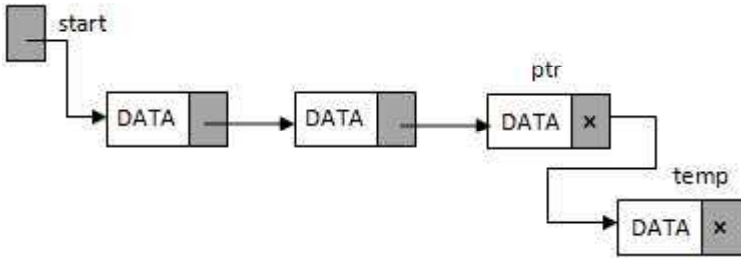


Figure 3.8: Insertion of node at the end

```
while(ptr!=NULL)
    ptr = ptr –> next;
    ptr–>next = temp;
```

Insertion in between: For inserting a node at any position of the linked list we will have to first traverse the linked list for obtaining the node after which we want to insert the new node. Let us consider a pointer **temp** which points to the node that has to be inserted. The data part of the new node is assigned to the data (i.e., d) through the temp pointer as:

temp–>data=d;

Again, we consider a pointer **q** which points to the node after which we have to insert the new node. For inserting the element after the node, we assign the next part of the node to the next part of new inserted node as:

temp–>next=q–>next;

Then the address part of the new inserted node is assigned to the next part of the previous node.

q–>next=temp;

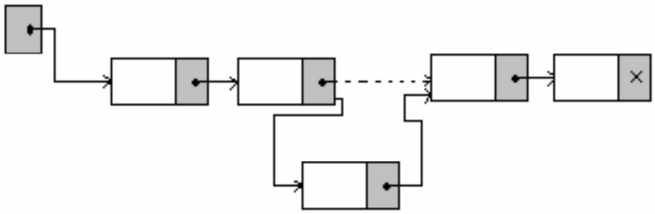


Figure 3.9: Insertion of node in between the linked list

3.8 DELETION FROM A SINGLE LINKED LIST

An element that is to be deleted from a linked list is to be searched first. For this, the traversing operation must be carried out thoroughly on the list. After finding the element there may be three cases for deletion:

- Deletion at beginning
- Deletion at end
- Deletion in between

Deletion at beginning: We know that **start** pointer points to the first element of a linked list. If the element to be deleted is the first element of linked list then we assign the value of start to temp as:

temp=start;

So now temp points to first node which has to be deleted. Now we assign the next part of the deleted node to start as:

start=start->next;

Since start points to the first element of the linked list, so start-> next will point to the second element of the linked list. Now, we should free the element to be deleted which is pointed by temp with the following statement:

free(temp);

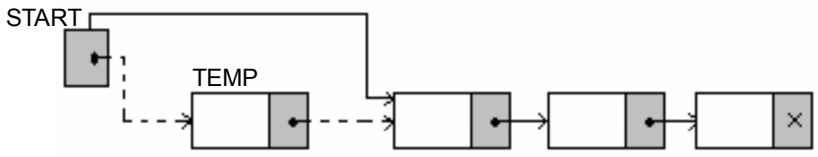


Figure 3.10: Deletion of node at the beginning

Deletion at end: If the element to be deleted is the last element of the linked list then we need to make the second last node as the last node of the linked list. We traverse the linked list using a pointer “temp” that will

point to the last node and another pointer “prev” that points to the second last node. To make the second last node, the last node of the linked list we assigh a value of NULL to the next part of prev and delete the element pointed by temp using the following statements:

```
temp=start;
    while (temp!=NULL)
    {
        prev = temp;
        temp=temp–>next;
    }
    prev–>next = NULL;
free(temp);
```

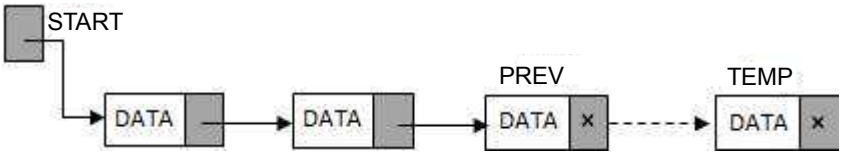


Figure 3.11: Deletion of node at the end of the linked list

Deletion in between: If the element is other than the first element of the linked list then we assign the next part of the deleted node to the next part of the previous node. This can be written as:

```
temp=q–>next;
q–>next=temp–>next;
free(temp);
```

Here, the pointer **q** is pointing to the previous node of the node to be deleted. After the first statement **temp** will point to the node to be deleted, after second statement next of previous node will point to the next node of the node to be deleted.

If the node to be deleted is the last node of the linked list then the second statement will be as:

```
q–>next=NULL;
```

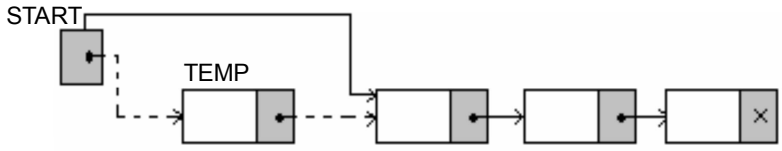


Figure 3.12: Deletion of node in between the linked list

The basic operation performed on a single linked list is implemented in the following C program.

Program 3.1: Program for operation of single linked list:

```
# include<stdio.h>
# include<conio.h>
# include<malloc.h>
struct node
{
int data;
struct node *next;
};
struct node *start;
struct node* create(int ele);
void display(struct node *);
struct node* add_beg(int ele, struct node* start);
struct node* add_end(int ele, struct node* start);
struct node* add_mid(int ele, int pos, struct node*
start);
struct node* del_beg(struct node* start);
struct node* del_end(struct node* start);
struct node*del_mid(int pos, struct node* start);
void search(int d);

int main()
{
int ch,num,pos;
clrscr();
do
{
printf("Enter\n 1 to create\n 2 to display\n 3 to
insert at beginning\n 4 to insert at end\n 5 to
insert after a postion\n 6 to delete at beginning\n
7 to delete at end\n 8 to delete at position\n 9 to
search\n 10 to exit\n");
```

```
scanf(''%d'',&ch);
    switch(ch)
    {
    case 1: printf(''Creation\n Enter number: '');
            scanf(''%d'',&num);
            start = create(num);
            break;
    case 2: printf(''Displaying list\n'');
            display(start);
            break;
    case 3: printf(''Insertion at the
            beginning.\nEnter the
            element:'');
            scanf(''%d'',&num);
            start = add_beg(num,start);
            break;
    case 4: printf(''Insertion at the
            end.\nEnter the element:'');
            scanf(''%d'',&num);
            start = add_end(num,start);
            break;
    case 5: printf(''Insertion at the
            middle.\nEnter the element:'');
            scanf(''%d'',&num);
            printf(''Enter the position of the
            element after
            which element will be added:\n'');
            scanf(''%d'',&pos);
            start = add_mid(num, pos, start);
            break;
    case 6: printf(''Delete at beginning\n'');
            start = del_beg(start);
            break;
```

```
        case 7: printf(''Delete at end\n'');
                    start=del_end(start);
                    break;
        case 8: printf(''Enter the position at which
                to delete:\n'');
                    scanf(''%d'',&pos);
                    start = del_mid(pos,start);
                    break;
        case 9: printf(''Enter the element you want
                to search:'');
                    scanf(''%d'',&num);
                    search(num);
                    break;
        case 10:exit();
        }
    } while(ch != 10);
    getch();
    return 0;
    }
    struct node* create(int ele)
    {
    struct node *cont;
    cont = (struct node *)malloc(sizeof(struct node));

    cont->data = ele;
    cont->next = NULL;

    start= cont;
    return start;
    }

    void display(struct node *temp)
    {
```

```
temp = start;
if(temp!=NULL)
{
    while(temp->next != NULL)
    {
           printf(''%d\n'',temp->data);
           temp = temp->next;
    }
    printf(''%d\n'',temp->data);
    }
else
{
    printf(''\nThe linked list is empty\n'');
}
}

struct node * add_beg(int ele,struct node *start)
{
struct node* cont;

cont = (struct node *)malloc(sizeof(struct node));
cont->data = ele;
cont->next = start;

start=cont;
return start;
}

struct node * add_end(int ele,struct node *start)
{
struct node* temp;
struct node* cont;

cont = (struct node *)malloc(sizeof(struct node));
```

```
cont->data = ele;
cont->next = NULL;
temp = start;

while(temp->next != NULL)
{
    temp = temp->next;
}
temp->next = cont;

return start;
}

struct node * add_mid(int ele,int pos, struct node
*start)
{
struct node* temp1;
struct node* temp2;
struct node* cont;
int count=1;

cont = (struct node *)malloc(sizeof(struct node));
cont->data = ele;
temp1 = start;

while(count < pos)
{
    temp1 = temp1->next;
    count = count+1;
}

temp2=temp1->next;
temp1->next=cont;
```

```
cont->next=temp2;

return start;
}

struct node* del_beg(struct node* start)
{
struct node *temp=start;
if(temp->next==NULL)
    {
           start = NULL;
           free(temp);
    }
else
    {
           start = start->next;
           free(temp);
    }
    return start;
}

struct node* del_end(struct node* start)
{
struct node *temp=start;
struct node *prev=NULL;

           if(start->next == NULL)
           {
               start = NULL;
           }
           else
           {
               while (temp->next!=NULL)
```

```
                    {
                        prev = temp;
                        temp=temp->next;
                    }
                prev->next = NULL;
                free(temp);
                }
                return start;
}
struct node* del_mid(int pos, struct node* start)
{
struct node *temp1=start,*temp2,*prev;
int count=1;

if(pos==1)
{
start=start->next;
}
else
{
    while (count < pos)
    {
            prev=temp1;
            temp1=temp1->next;
            count = count + 1;
    }

    temp2=temp1->next;
    prev->next = temp2;
    free(temp1);
}
return start;
}
```

```
void search(int d)
{
    struct node *ptr = start;
    int pos = 1;
    while(ptr!=NULL)
    {
           if(ptr->data==d)
           {
               printf("Item %d found at position
               %d\n",d,pos);
               return;
           }
           ptr = ptr->next;
           pos++;
    }
    if(ptr == NULL)
           printf(''Item %d not found in
           list\n'',d);
}
```



CHECK YOUR PROGRESS

Q.1: Define linked list. Name the three different types of linked list.

Q.2: Visiting all the nodes in the linked list is known as:

a) Searching b) Traversing

c) Insertion d) none of these

Q.3: A ..... identifies the last logical node in a linked list.

Q.4: Write the functions in C language for the following:

a) Creating a linked list.

b) Inserting a node at the beginning of a list.

3.9 REVERSING A SINGLE LINKED LIST

Reversing a linked list is to simply change the order of the elements in the linked list. The first element in the linked list becames the last element, the second elemnt becomes the second last element and so on till the last element of the previous linked list becomes the first element of the new linked list.

Program 3.2: Program to reverse a linked list

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>

struct node
{
    int data;
    struct node *next;
};

struct node *start;
struct node* create(int ele);
void display(struct node *);
struct node* add_end(int ele, struct node* start);
struct node *reverse(struct node* start);

void main()
{
    int num;
    clrscr();
    start=NULL;

    printf(''Creation\n Enter first elemnt:'');
    scanf(''%d'',&num);
    start = create(num);
```

```
        printf(''Insertion at the end.\nEnter the 2nd
        element:'');
        scanf(''%d'',&num);
        start = add_end(num,start);

        printf(''Insertion at the end.\nEnter the 3rd
        element:'');
        scanf(''%d'',&num);
        start = add_end(num,start);

        printf(''Insertion at the end.\nEnter the 4th
        element:'');
        scanf(''%d'',&num);
        start = add_end(num,start);

        printf(''Displaying linked list\n'');
        display(start);

        printf(''Reversing linked list\n'');
        start = reverse(start);
        printf(''Displaying reversed list\n'');
        display(start);
    getch();
    }

    struct node * create(int d)
    {
        struct node *temp;
        temp= malloc(sizeof(struct node));
        temp->data=d;
        temp->next=NULL;

        if(start==NULL)/*If list is empty */
```

```
            start=temp;
        else
        printf(''Linked list already created'');

        return start;
}

void display(struct node *temp)
{
temp = start;
if(temp!=NULL)
{
        while(temp->next != NULL)
        {
            printf(''%d\n'',temp->data);
            temp = temp->next;
        }
        printf(''%d\n'',temp->data);
}
else
        {
            printf(''\nThe linked list is empty\n'');
        }
}
        struct node * add_end(int ele,struct node *start)
{
        struct node* temp;
        struct node* cont;

        cont=(struct node *)malloc(sizeof(struct node));
        cont->data=ele;
        cont->next=NULL;
        temp = start;
```

```
        while(temp->next != NULL)
        {
            temp = temp->next;
        }
            temp->next = cont;
            return start;
        }

            struct node* reverse(struct node* root)
        {
            struct node* new_root = NULL;

            while (root)
        {
            struct node* temp = root->next;
            root->next = new_root;
            new_root = root;
            root = temp;
        }
        return new_root;
    }
```



3.10 LET US SUM UP

- A **linked list** is a dynamic data structure that consists of a sequence of nodes such that in each node there are two fields: one field holds data items and the other field contains a reference (i.e., a *link*) to the next record in the sequence.
- There are three types of linked list. These are: Singly linked list, Doubly linked list and Circular linked list.
- The following are the primitive operations that can be performed with linked list: traversing, searching, insertion, and deletion

- Traversing is the procedure of passing through (visiting) all the nodes of the linked list from starting node to the ending node. When any given record is to be searched in the linked list, traversing is appied. When the given element is found, the operation can be terminated or continued for next search.
- Searching is a process in which a given element is compared with all the linked list elements.
- Insertion is the procedure through which we can add new node into a linked list at any position i.e., at the begining of the list or at any position of the list.
- Deletion is a process in which we can delete or remove elements from a linked list i.e. deleting the first node, deleting the last node as well as deleting a particular node from the linked list.
- Reversing a linked list is to simply change the order the order of elements in the linked list.



3.11 FURTHER READING

1) Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.
2) Langsam, Yedidyah; Augenstein, Moshe J. & Tenenbaum, Aaron M. *Data Structures Using C and C++*. Prentice-Hall India.
3) Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc.



3.12 ANSWER TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: A **linked list** is a dynamic data structure that consists of a sequence of nodes such that in each node there are two fields: one field holds data items and the other field contains a reference (i.e., a *link*) to the next record in the sequence.

There are three types of linked list: single linked list, double linked list and circular linked list.

Ans. to Q. No. 2: b) Traversing

Ans. to Q. No. 3: NULL,

Ans. to Q. No. 4: a)

```
create(int d)
{
    struct node *q,*temp;
    temp= malloc(sizeof(struct node));
    temp–>data=d;
    temp–>next=NULL;
    if(start==NULL)/*If list is empty*/
        start=temp;
    else
    {   /*Element inserted at the end*/
        q=start;
        while(q–>next!=NULL)
            q=q–>next;
        q–>next=temp;
    }
}
```

b)

```
add_beg(int d)
{
    struct node *temp;
    temp=malloc(sizeof(struct node));
    temp–>data=d;
    temp–>next=start;
    start=temp;
}   /*End of add_beg()*/
```



3.13 MODEL QUESTIONS

Q.1: Write a C program to remove the first and the last node of the linked list.

Q.2: Write a C program to create and display the elements of a single linked list.

Q.3: Write a C program to insert a node at the first location of a linked list.

Q.4: Describe the insertion procedure in a single linked list.

Q.5: Describe the delete operation in a single linked list.

*** ***** ***

UNIT 4: STACKS

UNIT STRUCTURE

4.1 Learning Objectives

4.2 Introduction

4.3 Stack

4.4 Operations on Stack

4.5 Implementation of Stack

- 4.5.1 Implementation of Stack using Arrays
- 4.5.2 Implementation of Stack using Linked Lists

4.6 Applications of Stack

4.7 Let Us Sum Up

4.8 Further Reading

4.9 Answers to Check Your Progress

4.10 Model Questions

4.1 LEARNING OBJECTIVES

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

- describe the concept of stack
- explain LIFO structure
- describe the operations associated with a stack
- implement stack using arrays
- implement stack using linked lists
- list applications of stack.

4.2 INTRODUCTION

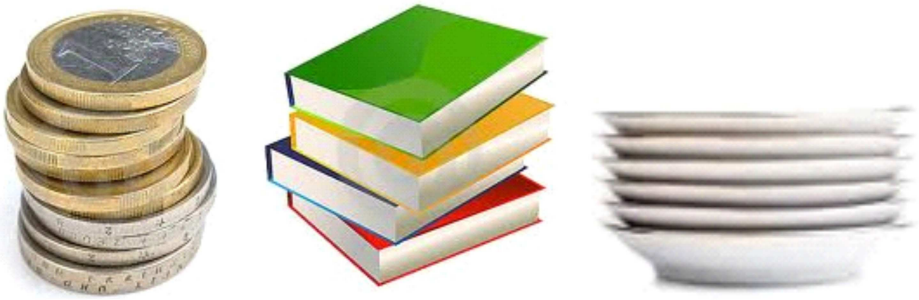
In the previous units, we have learnt about primitive data structures that are available in C. In unit 3, while studying linked list we have seen that insertion and deletion of element is possible at any place in the list. There may arise certain situations in which items may need to be inserted or removed only at one end. Stacks and queues are such data structures where insertion and deletion are done at only one end. In this unit, we will learn

about the stack data structure and its applications. In the next unit, we will explore the concept of queues.

4.3 STACK

Stack is a special type of data structure where elements are inserted from one end and elements are deleted from the same end. The position from where elements are inserted and from where elements are deleted is termed as **top** of the stack. Thus, stack is a homogeneous collection of elements of any one type, arranged linearly with access at one end only.

In order to clarify the idea of a stack let us consider some real life examples of stack. Figure 4.1 depicts three everyday examples of such a data structure. A stack of book is a good analogy: we can't read any book in the stack without first removing the books that are stacked on top of it. In the same way, plates in a hotel are kept on top of each other. When a plate is taken it is usually taken from the top of the stack. Using this approach, the last element inserted is the first element to be deleted out, and hence, stack is also called **Last In First Out** (LIFO) data structure. Although the stack may seem to be a very restricted type of data structure, it has many important applications in computer science.



Stack of coins Stack of books Stack of plates

Figure 4.1: Real life examples of Stack

4.4 OPERATIONS ON STACK

The two basic operations associated with stacks are *Push* and *Pop*.

Push: Data is added to the stack using the *Push* operation.

Pop: Data is removed using the *Pop* operation

Push operation: The procedure of inserting a new element to the top of the stack is known as *push* operation. For example, let us consider the stack shown in figure 4.2 with STACK_SIZE = 4 where we can insert atmost four elements. If we consider TOP as a pointer to the top element in a stack then after every push operation, the value of TOP is incremented by one. After inserting 11, 9, 7 and 5 there is no space to insert any element. Then we say that stack is full. If the stack is full and does not contain enough space to accept the given element or data, the stack is then considered to be in an ***overflow*** state.

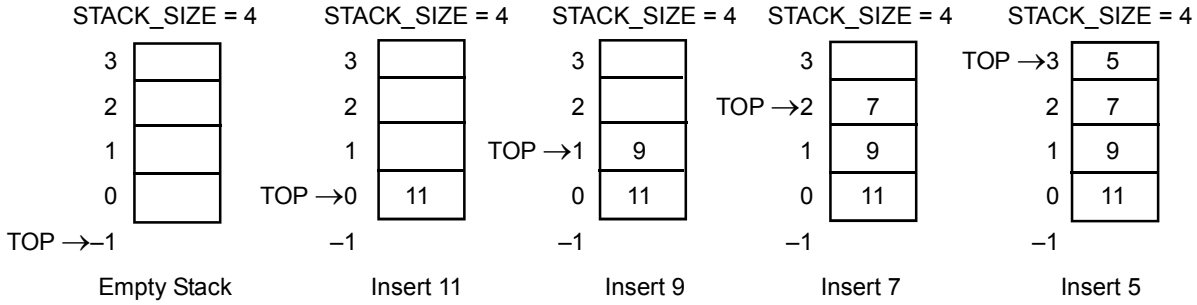


Figure 4.2: Sequence of *Push* operation

Pop operation: The procedure of removing element from the top of the stack is called *pop* operation. Only one element can be deleted from at a time and the element has to be deleted only from the top of the stack. When elements are being deleted, there is a possibility of stack being empty. When stack is empty, it is not possible to delete any more element. Trying to delete an element from an empty stack results in stack **underflow**. Figure 4.3 illustrates the pop operation. After every pop operation, the value of TOP is decremented by one.

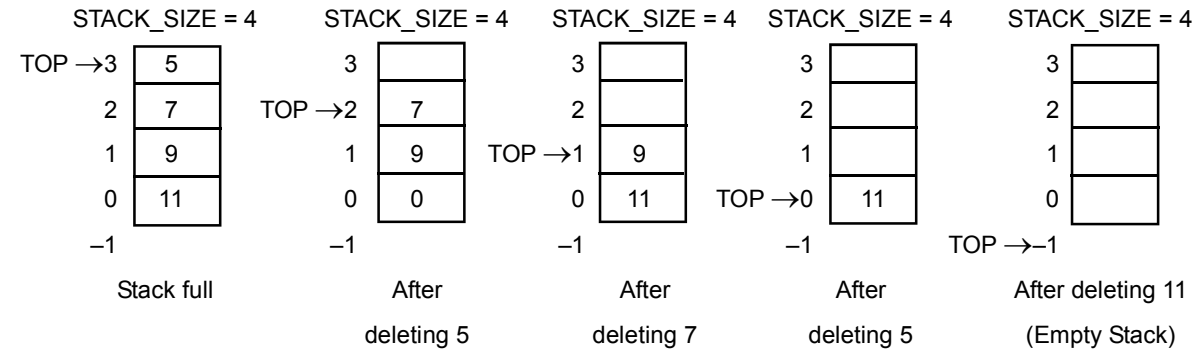


Figure 4.3: Sequence of Pop operation

In the above figure, after deleting 5, 7, 9 and 11, there are no elements in the stack and the stack becomes empty.

4.5 STACK IMPLEMENTATION

A stack can be implemented using either an array or a singly linked list. Thus there are two methods of stack implementation. They are: *Static implementation* and *Dynamic implementation*.

Static implementation can be achieved using arrays. Though array implementation is a simple technique, it provides less flexibility and is not very efficient with respect to memory organization. This is because once a size of an array is declared, its size cannot be modified during program execution. If the number of elements to be stored in a stack is less than the allocated memory, then memory is wasted and if we want to store more elements than declared, array cannot be expanded. It is suitable only when we exactly know the number of elements to be stored.

A stack can be implemented using pointers, as a form of a linked list. **Dynamic implementation** can be achieved by using linked list as it is a dynamic data structure. The limitations of static implementation can be removed using dynamic implementation. The memory is efficiently utilized with pointers. Memory is allocated only after element is inserted to the stack. The stack can grow or shrink as the program demands it to. However, if a small and/or fixed amount of data is being dealt with, it is often simpler to implement the stack as an array.

4.5.1 Implemenation of Stack using Arrays

One of the two ways to implement a stack is by using a one dimensional array. When implemented this way, the data is simply stored in the array. A variable named *"Top"* is used to point to the top element of the stack. Each time data is added or removed, *Top* is incremented or decremented accordingly, to keep track of the current *Top* of the stack. Initially, the value of *Top* is set to –1 to indicate an empty stack.

To push (or insert) an element onto the stack, *Top* is incremented by one, and the element is pushed at that position. When *Top* reaches SIZE-1 and an attempt is made to push a new element, then the stack overflows. Here, SIZE is the maximum size of the stack. Similarly, to pop (or remove) an element from the stack, the element on the *Top* of the stack is displayed, and then *Top* is decremented by one. When the value of *Top* is equal to –1 and an attempt is made to pop an element, the stack underflows.

Program 4.1: Stack implementation using arrays (static implementation of stacks)

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define SIZE 10
void push();
void pop();
void display();
int top= -1; // value of top is initialised to -1
int stack[SIZE];
void main()
{      int choice;
        while(1)
       {
        printf(''\n1.Push\n'');
        printf(''\n2.Pop\n'');
        printf(''\n3.Display\n'');
        printf(''\n4.Quit\n'');
        printf(''\nEnter your choice:'');
        scanf(''%d'',&choice);
        switch(choice)
        {
        case 1: push();
              break;
```

```
            case 2: pop();
                    break;
            case 3: display();
                    break;
            case 4: exit(1);
                    default:printf(''Invalid
                    Choice\n'');
            }
        }
}
void push()
{   int item;
    if(top==(SIZE-1))
        printf(''\nStack Overflow'');
    else
    {
        printf(''\nEnter the item to be pushed in
        stack:'');
        scanf(''%d'',&item);
        top=top+1;
        stack[top]=item;
    }
}
void pop()
{
    if(top==-1)
    printf(''Stack Underflow\n'');
    else
    {
        printf(''\nPopped element is:
        %d\n'',stack[top]);
        top=top-1;
    }
```

```
}
void display()
{
    int i;
    if(top==-1)
        printf(''\nStack is empty\n'');
    else
    {
        printf(''\nStack elements:\n'');
        for(i=top;i>=0;i--)
        printf(''%d\n'',stack[i]);
    }
}
```

Stacks implemented as arrays are useful if a fixed amount of data is to be used. However, if the amount of data is not a fixed size or the amount of the data fluctuates widely during the stack's life time, then an array is a poor choice for implementing a stack. A much more elegant solution to this problem will be covered in the next section.



CHECK YOUR PROGRESS

Q.1: Fill in the blanks:

i) The insertion of an element in a stack is known as the ..... operation.

ii) The removing an element from the stack is known as ..... operation.

Q.2: What would the state of a stack be after the following operations:

create stack

push A onto stack

push F onto stack

push X onto stack

pop item from stack

push B onto stack

pop item from stack

pop item from stack

Q.3: State whether the following statements are true or false:

i) Stack follows a first-in-first-out technique.

ii) Insertion of data into the stack is called the push operation.

iii) Removal of element is termed as pop operation.

4.5.2 Implementation of Stack using Linked List

When a stack is implemented as a linked list, each node in the linked list contains the data and a pointer that gives location of the next node in the list. In this implementation. there is no need to declare the size of the stack in advance since we create nodes dynamically as well as delete them dynamically.

A pointer variable **Top** is used to point to the top element of the stack. Initially, **Top** is set to NULL to indicate an empty stack. Whenever a new element is to be inserted in the stack, a new node is created and the element is inserted into the node. Then, **Top** is modified to point to this new node.

Program 4.2: Program implementing stack using linked list

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<stdlib.h>
void push();     //push() function declaration
void pop();      //pop() function declaration
void display();  //display() function declaration

struct node
{
    int data;
```

```
    struct node *next;
};
struct node *top=NULL;
void main()
{
    int choice;
    clrscr();
    while(1)
    {
    printf(''\n1.Push'');
    printf(''\n2.Pop'');
    printf(''\n3.Display'');
    printf(''\n4.Quit'');
    printf(''\nEnter your choice:'');
    scanf(''%d'',&choice);
    switch(choice)
        {
        case 1: push();
              break;
        case 2: pop();
              break;
        case 3: display();
              break;
        case 4: exit(1);
        default: printf(''Invalid Choice\n'');
        } //end of switch
    } //end of while loop
} //end of main fuction
void push()
{
    struct node *ptr;
    int item;
    ptr=(struct node *)malloc(sizeof(struct node));
```

```
    printf(''Enter a value to be pushed onto the
    stack:'');
    scanf(''%d'',&item);
    ptr->data=item;
    ptr->next=top;
    top=ptr;
} //end of push fuction
void pop()
{
    struct node *ptr,*next;
    if(top==NULL)
        printf(''\nStack is Empty'');
    else
    {
        ptr=top;
        printf(''\nPopped element is: %d\n'',
        ptr->data);
        top=top->next;
        free(ptr);
    }
} //end of pop function definition
void display()
{
    struct node *p;
    p=top;
    if(top==NULL)
        printf(''\nStack is empty\n'');
    else
    {
        printf(''\nStack elements:\n'');
        while(p!=NULL)
        {
              printf(''%d\n'',p->data);
```

```
                p=p->next;
            }
        }
    } //end of display function
```

4.6 APPLICATIONS OF STACKS

Stacks have many applications. For example, as processor executes a program, when a function call is made, the called function must know where to return back to the program, so the current address of program execution is pushed onto a stack. Once the function is finished, the address that was saved is removed from the stack, and execution of the program resumes.

If a series of function calls occur, the successive return values are pushed onto the stack in Last-In-First-Out order so that each function can return back to calling program. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

Stacks are also used by compilers in the process of evaluating expressions and generating machine language code. Two simple applications of stack are described below:

Reversal of string: We can reverse a string by pushing each character of the string on the stack. When the whole string is pushed on the stack we will pop the characters from the stack and we will get the reversed string.

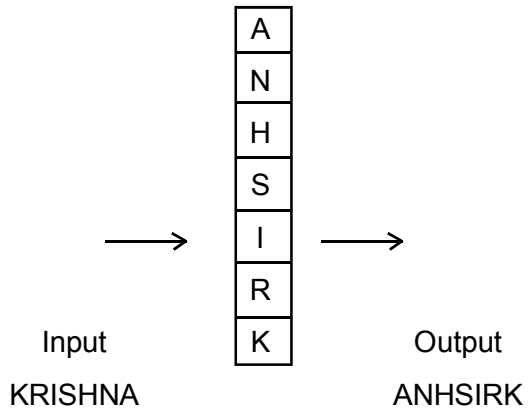


Figure 4.5: Stack

Program 4.3: Program for reversing a string using stack.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define SIZE 25
int top=-1;
char stack[SIZE];
char pop();
void push(char);
void main()
{
    char str[20];
    int i;
    clrscr();
    gets(str);
    printf(''\nEnter a string :'');
 //Push characters of the string str on the stack
    for(i=0; i < strlen(str); i++)
        push(str[i]);
//Pop characters from the stack and store in string
//str
    for(i=0; i<strlen(str); i++)
        str[i]=pop();
    printf(''\nString in reversed order is: '');
    puts(str);
    getch();
 } //end of main
void push(char item)
{
    if(top==(SIZE-1))
        printf(''\nStack Overflow'');
    else
        stack[++top]=item;
```

```
}   //end of push function
char pop()
{
    if(top == -1)
        printf(''\nStack Underflow'');
    else
        return stack[top--];
}   //end of pop function
```

If we enter a string *State University* then the output of the above program will be *ytisrevinU etatS.*

Evaluation of arithmetic expressions: Stacks can also be useful in the evaluation of arithmetic expression. Given an expression in postfix notation. Using a stack they can be evaluated as follows:

- Scan the symbol from left to right
- If the scanned symbol is an operand, push it on to the stack.
- If the scanned symbol is an operator, pop two elements from the stack. The first popped element is op2 and the second popped element is op1. This can be achieved using the statements:

  op2=s[top--]; //First popped element is operand 2

  op1=s[top--]; //second popped element is operand 1
- Perform the indicated operation

  result=op1 op op2 //op is the operator such as +, –, /, *
- Push the result on to the stack.
- Repeat the above procedure till the end of input is encountered.



CHECK YOUR PROGRESS

Q.4: Select the appropriate option for each of the following questions:

i) The push() operation is used

a) to move an element b) to remove an element

c) to insert an element d) none of these

ii) The stack is based on the rule

a) first-in-first-out b) last-in-first-out

c) both (a) and (b) c) none of these

iii) A stack holding elements equal to its capacity and if push is performed then the situation is called

a) Stack overflow b) Stack underflow

c) Pop d) illegal operation

iv) The top pointer is increased

a) when push() operation is done

b) when pop() operation is done

c) both (a) and (b)

d) none of the above

v) The pop operation removes

a) the element lastly inserted

b) first element of the stack

c) any element randomly

d) none of the above



4.7 LET US SUM UP

- Stack is one of the most essential linear data structures and an abstract data type.
- The insertion of element onto a stack is called *Push* and deletion operation is called *pop*.
- The most and the least reachable elements in a stack are respectively known as the *top* and *bottom*.
- The insertion and deletion operations are carried out at one end.
- The recent element inserted is deleted first. If we want to delete a particular element of a stack, it is necessary to delete all the elements appearing before the element. This means that the last item to be added to a stack is the first item to be removed. Accordingly, stack is a set of elements in a *Last-In-First-Out* (**LIFO**) technique.

- When stack has no element, it is called *empty stack*.
- *Overflow* occurs when the stack is full and there is no space for a new element, and an attempt is made to push a new element.
- *Underflow* occurs when the stack is empty, and an attempt is made to pop an element.
- Static can be implemented using arrays and linked lists.



4.8 FURTHER READING

1) Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.
2) Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc.
3) Srivastava, S. K. & Srivastava, D. (2003). *Data Structures Through C in Depth*. BPB Publications.



4.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: i) Push, ii) Pop

Ans. to Q. No. 2: Stack will contain only one item and the item is A.

Ans. to Q. No. 3: i) False, ii) True, iii) True

Ans. to Q. No. 4: i) (c) insert an element;
ii) (b) last-in-first-out
iii) (a) Stack overflow
iv) (a) when push() operation is done
v) (a) the element lastly inserted



4.10 MODEL QUESTIONS

Q.1: What is a stack? What different operations can be performed on stacks?

Q.2: Why are stacks called “LIFO” structures?

Q.3: Explain the *push* and *pop* operations.

Q.4: What are *stack overflow* and *stack underflow*?

Q.5: Define linked implementation of stack.

Q.6: Write a C program to implement stack with array. Perform *push* and *pop* operation.

Q.7: Write a program to perform following operations:
(a) push (b) pop (c) display all elements.

Q.8: Distinguish between static and dynamic implementation of stack.

Q.9: Describe two applications of stack.

Q.10: What are the advantages and disadvantages of linked implementation of a stack relative to the contiguous implementation?

Q.11: Write a program of stack where elements will be pushed from last position of array.

Q.12: Write a program to implement a stack which contains the address of the pointers for allocation of memory. Do the pop operation on stack and free the popped pointers.

Q.13: Write a recursive function in C to compute the sum of n natural numbers.

Q.14: What are the two ways of implementing stacks. Which one is preferred over the other and why?

Q.15: What are the various applications of stacks? Write a C program to implement any one of them.

*** ***** ***

UNIT 5: QUEUE

UNIT STRUCTURE

5.1 Learning Objectives

5.2 Introduction

5.3 Queue

5.4 Array Implementation of Queue

5.5 Circular Queue

5.6 Linked List Implementation of Queue

5.6.1 Using Singly Linked List

5.6.2 Using Doubly Linked List

5.6.3 Using Circular Linked List

5.7 Priority Queues

5.8 Application of Queues

5.9 Let Us Sum Up

5.10 Further Reading

5.11 Answers to Check Your Progress

5.12 Model Questions

5.1 LEARNING OBJECTIVES

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

- describe queue data structure
- explain the usefulness of circular queue and implement it
- describe array implementation of queue
- describe list implementations of queue
- list different applications of queue.

5.2 INTRODUCTION

In the previous units, we have learnt about arrays and linked list as well as stack data structure. Now, in this unit we will learn about another linear data structure known as queue. Different implementation of queue using array and linked list are also described in detail in this unit. We will

also explore the different applications of queue in this unit. In the next unit, we will cover the different searching techniques.

5.3 QUEUE

Queue is a linear data structure in which removal of elements are done in the same order they were inserted i.e., the element will be removed first which is inserted first. A queue has two ends which are front end and rear end. Insertion is to take place at rear end and deletion is to take place at front end. Because of this, queue is also known as First In First Out (FIFO) data structure.

5.4 ARRAY IMPLEMENTATION OF QUEUE

Array implementation of queue can be done in many ways. In this array implementation of queue, a single array and two variables are used. The two variables named 'front' and 'rear' are initialized with a value of '–1'. When we want to insert an element, at first the value of 'rear' is incremented if it is within the range and then the element is inputted into the array at the subscript value equal to the value of 'rear'. The overflow condition for array implementation in C language is **"front' is equal to 0 AND 'rear' is equal to (size of the array) –1".** Now, if in any case if "'rear' is equal to (size of the array) –1 but 'front' is not equal to 0 then to insert a new element, the elements in the queue must be shifted towards the left most direction of the array, so that at the rear end one or more free spaces become available to insert new elements. Here, the underflow condition is "'front' is equal to '–1'". In case of deletion operation 'front' variable is incremented by one if it is not equal to '–1'.

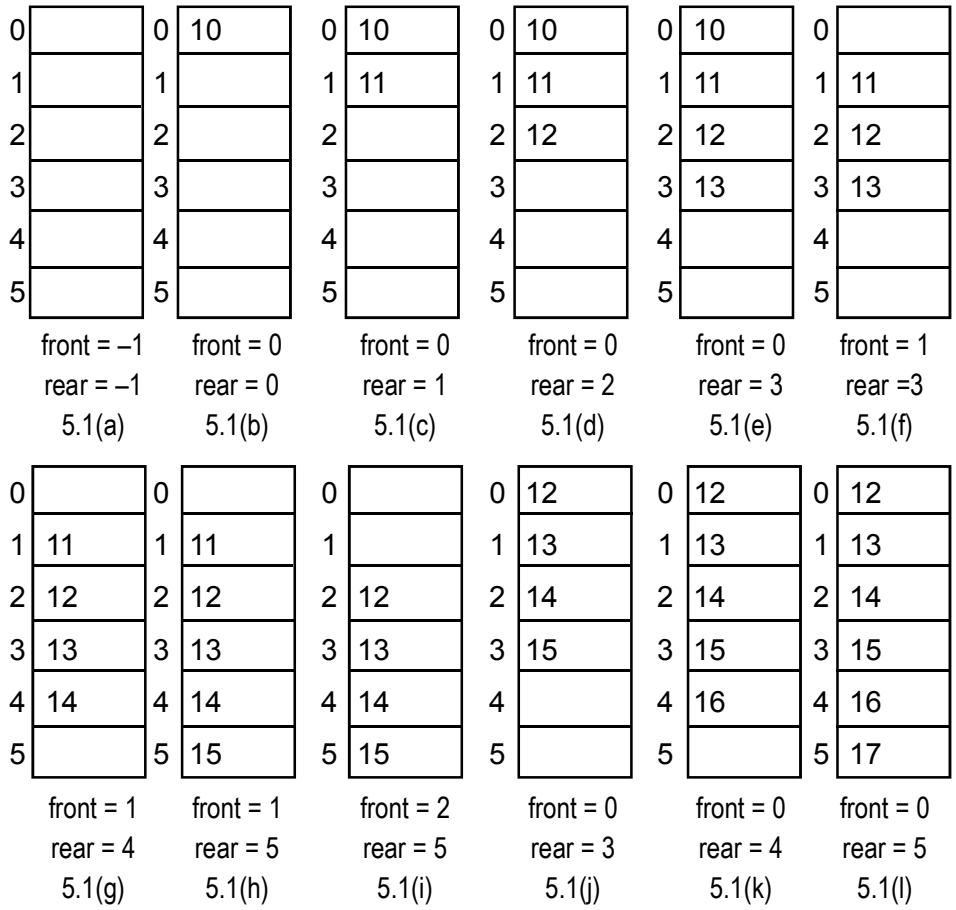


Figure 5.1: (a) Empty queue, (b) Insertion of "10", (c) Insertion of "11", (d) Insertion of "12", (e) Insertion of "13", (f) Deletion of first element, (g) Insertion of "14", (h) Insertion of "15", (i) Deletion of next element, (j) Shifting of elements, (k) Insertion of "16", (l) Insertion of "17"

In figure 5.1(a), an empty queue is represented with front = –1 and rear = –1. In figure 5.1(b), 10 is inserted into the queue and it is the first element in the queue with front = 0 and rear = 0. In figure 5.1(c), 11 is inserted into the queue with front = 0 and rear = 1. Similarly to this, elements 12, 13, 14, 15, 16 and 17 are inserted into the queue in figures 5.1(d), (e), (g), (h), (k) and (l) respectively. In figure 5.1(j), the existing elements in the queue are shifted towards the beginning of the queue to make free space for the insertion of 16 and 17. In figure 5.1(f), 10 is deleted with front = 1 and rear = 3 and in 1(i), 11 is deleted with front = 2 and rear = 5.

Algorithm 5.1: Algorithm for inserting new information into a queue

Here, queue[ ] is the array to store the array elements
'front' is a variable used to represent the front end of the queue
'rear' is a variable to represent the rear end of the queue
'element' is a variable to store the new element to be inputted into the queue.

Insert (queue [ ], front, rear, element)

Step 1: IF front == 0 AND rear == (size of queue)–1 THEN
Step 2: PRINT "Queue is overflow"
Step 3: END OF IF
Step 4: ELSE
Step 5: IF rear == (size of queue)–1 THEN
Step 6: FOR I = 0 TO i == rear – front
Step 7: queue[i] = queue[front +i]
Step 8: END OF FOR
Step 9: rear = rear – front
Step 10: front = 0
Step 11: END OF IF
Step 12: IF front == –1
Step 13: front = 0
Step 14: END OF IF
Step 15: rear = rear+1
Step 16: queue [rear] = element
Step 17: END OF ELSE

Algorithm 5.2: Algorithm for deleting information from a queue

Here, queue [ ] is the array to store the array elements
'front' is a variable used to represent the front end of the queue
'rear' is a variable to represent the rear end of the queue

Delete (queue[ ], front, rear)

Step 1: IF front == –1
Step 2: PRINT "Queue is underflow"
Step 3: END OF IF

Step 4: ELSE

Step 5: front = front + 1

Step 6: IF front > rear

Step 7: front = –1

Step 8: rear = –1

Step 9: END IF

Step 10: END OF ELSE

Program 5.1: C program to implement queue using array

```
#include<stdio.h>
#include<conio.h>
#define max 40

//Structure for creating a queue

struct queue
{
    int front,rear;
    int data[max];
};

//Function prototypes

int insert(struct queue *,int );
int delet(struct queue *);
int isfull(struct queue *);
int isempty(struct queue *);
void init(struct queue *);
void display(struct queue *);

void main()
{
struct queue *q;
int option,elem,flag;
```

```
char cont;
clrscr();
q = (struct queue *)malloc(sizeof(struct queue));
init(q);
do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
        printf("\n*****************");
        printf("\nEnter your option::");
        scanf("%d",&option);
        switch(option)
    {
        case 1: printf("\nEnter the element to
                be inserted into the
                queue::");
                scanf("%d",&elem);
                flag = insert(q,elem);
                if(flag == 1)
                {
                    printf("\n%d is succesfully
                    inserted into the
                    queue",elem);
                    printf("\nAfter insertion ");
                    display(q);
                }
                else
                {
                    printf("\nInsertion is not
                    possible as the
                    queue is full");
```

```
                }
            break;

            case 2: elem = delet(q);
                    if(elem == -99)
                    printf("\nDeletion is not possible
                    as the queue is
                    empty");
                    else
                    {
                        printf("\n%d is deleted from
                        the queue");
                        printf("\nAfter deletion'');
                        display(q);
                    }
            break;

            case 3: display(q);
            break;

        default: printf("\nWrong input...try again");
        }
        printf("\nDo you want to continue...Press 'y'
        or 'Y' to continue");
            cont = getch();
        } while(cont == 'y'||cont == 'Y');
    }

    //Function to initialize a queue
    void init(struct queue *q)
    {
        q->front = -1;
        q->rear = -1;
    }
```

```
//Function to check a queue is overflow or not
int isfull(struct queue *q)
{
    if(q->front == 0 && q->rear == max-1)
        return(1);
    else
        return(0);
}
//Function to check a queue is underflow or not
int isempty(struct queue *q)
{
    if(q->front == -1)
        return(1);
    else
        return(0);
}

//Function to insert a new element into a queue
int insert(struct queue *q,int element)
{
    int i;
    if(isfull(q))
        return(0);
    else
    {
        if(q->rear == max -1)
    {
        for(i = 0; i <= (q->rear)-(q->front); i++)
        q->data[i] = q->data[qv>front+i];

    q->rear = q->rear - q->front;
    q->front = 0;
    }
```

```
            if(q->front == -1)
                q->front = 0;
            q->data[++q->rear] = element;
            return(1);
        }
    }
    //Function to delete a element from a queue
    int delet(struct queue *q)
    {
        int delement;
        if(isempty(q))
        return(-99);
            else
            {
            delement = q->data[q->front];
        q->front++;
        if(q->front > q->rear)
            {
            q->front = -1;
            q->rear = -1;
            }
            return(delement);
        }
    }

    //Function to display the elements available in
    the queue
    void display(struct queue *q)
    {
        int i;
        if(isempty(q))
            printf("\nThe queue is empty");
        else
```

```
        {
            printf("\nthe elements in the queue
            are:\n");
            for(i = q->front; i <=q->rear; i++)
            printf("%5d",q->data[i]);
        }
    }
```

5.5 CIRCULAR QUEUE

In the earlier implementation of queue, we have learnt that shifting of queue elements must be done to the left most direction of the array to make free space available at the rear end in case of insertion of new elements when 'rear' is equal to 'size of the array' –1 and 'front' is not equal to 0. So, in case of a queue with a large number of elements, the shifting of queue elements is a huge wastage of time. This is a drawback of the queue. So to remove this drawback, circular queue implementation is used. In case of circular queue implementation, if 'rear' is equal to (size of the queue) –1 and 'front' is not equal to 0 then a new element is inserted into the array at the subscript value 0^{th} position.

Let us imagine the array as a circular list. Here, if the value of 'rear' or 'front' is {(size of queue) –1} then next value of 'rear' or 'front' is 0, otherwise the next value of 'rear' or 'front' will be one more than the earlier value. In this implementation, the 'front' and the 'rear' are initialized with the value '–1'. The overflow condition is "'front' is equal to the next value of the rear's recent value" and underflow condition is "'front' is equal to '–1'". In this implementation, if 'front' is not equal to –1 and 'front' is equal to 'rear' then it means that there is only one element is available in the circular queue. But in some other implementation, if 'front' is equal to 'rear' then it means that the circular queue is empty.

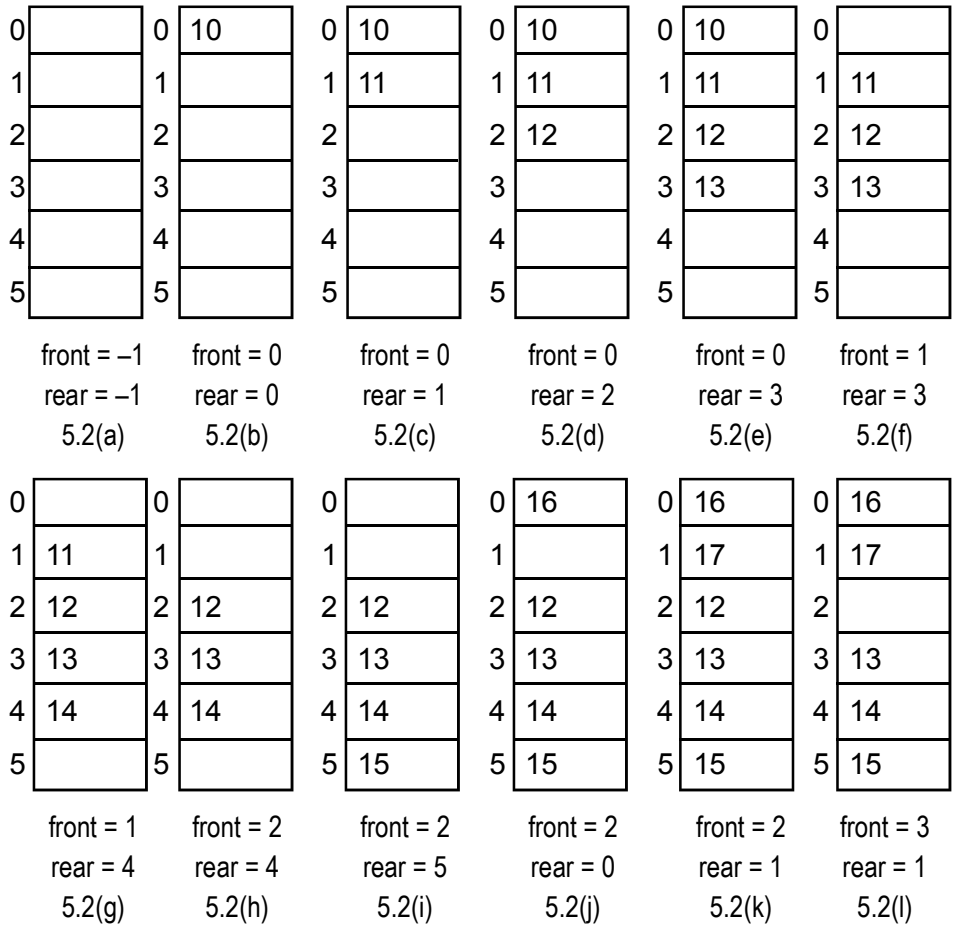


Figure 5.2: (a) Empty queue, (b) Insertion of "10", (c) Insertion of "11", (d) Insertion of "12", (e) Insertion of "13", (f) Deletion of first element, (g) Insertion of "14", (h) Deletion of next element, (i) Insertion of "15", (j) Insertion of "16", (k) Insertion of "17", (l) Deletion of next element

In figure 5.2(a), an empty circular queue is represented with front = –1 and rear = –1. In figure 5.2(b), 10 is inserted into the queue and it is the first element in the queue with front = 0 and rear = 0. Here 'front' is equal to 'rear' which means only one element is available in the circular queue. In figure 5.2(c), 11 is inserted into the queue with front = 0 and rear = 1. So like this, 12, 13, 14, 15, 16 and 17 are inserted into the circular queue in figures 5.2(d), (e), (g), (i), (j) and (k) respectively. In figure 5.2(f), 10 is deleted from the circular queue with front = 1 and rear = 3. In figure 5.2(h), 11 is deleted from the circular queue with front = 2 and rear = 4. In figure 5.2(l), 12 is deleted from the circular queue with front = 3 and rear = 1.

Algorithm 5.3: Algorithm for inserting new information into a circular queue

Here, cqueue[ ] is the array to store the queue elements.

'front' is a variable used to represent the front end of the circular queue.

'rear' is a variable to represent the rear end of the circular queue.

'element' is a variable to store the new element into the circular queue.

Insertcq (cqueue[ ], front, rear, element)

Step 1: IF front == (rear+1)% (size of cqueue)

Step 2: PRINT "Queue overflow"

Step 3: END OF IF

Step 4: ELSE

Step 5: rear = (rear+1)%(size of cqueue)

Step 6: IF front == –1

Step 7: front = 0

Step 8: END OF IF

Step 9: cqueue[rear] = element

Step 10: END OF ELSE

In this algorithm, the next value for 'front' and 'rear' can be calculated using modulus (%) operator. Modulus (%) operator gives the reminder of any division operation. When the value of 'rear' or 'front' is equal to (size of the queue) –1 then the next value of 'rear' or 'front' is equal to ((rear or front) +1)% (size of the queue) i.e., 0 otherwise it will be one more than the recent value of 'rear' or 'front'.

Algorithm 5.4: Algorithm for deleting information from a circular queue

Here, cqueue[ ] is the array to store the array elements

'front' is a variable used to represent the front end of the queue

'rear' is a variable to represent the rear end of the queue

deletecq(cqueue[ ],front,rear)

Step 1: IF front == –1

Step 2: PRINT “Queue is underflow”

Step 3: END OF IF

Step 4: ELSE

Step 5: IF front == rear)

Step 6: front = –1

Step 7: rear = –1

Step 8: END OF IF

Step 9: ELSE

Step 10: front = (front+1)%(size of cqueue)

Step 11: END OF ELSE

Step 12: END OF ELSE

Program 5.2: Program to implement circular queue

```
#include<stdio.h>
#include<conio.h>
#define max5

//Structure to create a circular queue

struct cirqueue
    {
    int front,rear;
    int data[max];
    };
typedef struct cirqueue cirqueue;

// Function prototypes

int insert(cirqueue **,int );
int delet(cirqueue **);
int isfull(cirqueue **);
int isempty(cirqueue **);
void init(cirqueue **);
void display(cirqueue *);
```

```
void main()
{
    cirqueue *q;
    int option,elem,flag;
    char cont;
    clrscr();
    q = (cirqueue *)malloc(sizeof(cirqueue));
    init(&q);
    do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
        printf("\n*****************");
        printf("\nEnter your option::");
        scanf("%d",&option);
        switch(option)
        {

           case 1: printf("\nEnter the element to
                   be inserted into the queue::");
                   scanf("%d",&elem);
                   flag = insert(&q,elem);
                   if(flag == 1)
                   {
                   printf("\n%d is succesfully
                   inserted into the queue",elem);
                   printf("\nAfter insertion ");
                   display(q);
                   }
                   else
                   {
```

```
                    printf("\nInsertion isnot
                    possible as the queue is full");
                    }
                    break;
            case 2: elem = delet(&q);
                    if(elem == -99)
                    printf("\nDeletion is not
                    possible as the queue is empty");
                    else
                    {
                    printf("\n%d is deleted from the
                    circular queue",elem);
                    printf("\nAfter deletion ");
                    display(q);
                    }
                    break;
            case 3: display(q);
                    break;
        default:printf("\nWrong input...try
        again");
        }
        printf("\nDo you want to continue..Press
        'y' or 'Y' to continue");
        cont = getch();
    } while(cont == 'y' || cont == 'Y');
}

//Function to initialize a circular queue
void init(cirqueue **q)
{
    (*q)->front = -1;
    (*q)->rear = -1;
}
```

```
//Function to check a circular queue is overflow
or not
int isfull(cirqueue **q)
{
    if((*q)->front == ((*q)->rear+1)%max)
        return(1);
    else
        return(0);
}

//Function to check a circular queue is underflow
or not
int isempty(cirqueue **q)
{
    if((*q)->front == -1)
        return(1);
    else
        return(0);
}

//Function to insert a new element into the circular
queue
int insert(cirqueue **q,int element)
{
    int i;
    if(isfull(q))
    return(0);
    else
    {
        (*q)->rear = ((*q)->rear+1)%max;
        if((*q)->front == -1)
        (*q)->front = 0;
        (*q)->data[(*q)->rear] = element;
```

```
            return(1);
        }
    }

    //Function to delete an element from the circular
    queue
    int delet(cirqueue **q)
    {
        int delement;
        if(isempty(q))
            return(-99);
        else
        {
            delement = (*q)->data[(*q)->front];
            if((*q)->front == (*q)->rear)
        {
            (*q)->front = -1;
            (*q)->rear = -1;
        }
        else
            (*q)->front = ((*q)->front+1)%max;
        return(delement);
        }
    }

    //Function to display elements available in the
    circular queue
    void display(cirqueue *q)
    {
        int i;
        if(isempty(&q))
        printf("\nThe queue is empty");
        else
```

```
    {
        printf("\nthe elements in the queue
        are:\n");
        I = q->front;
        while(i != q->rear)
        {
            printf("%5d", q->data[i]);
            i = (i+1)%max;
        }
        printf("%5d", q->data[i]);
    }
}
```

5.6 LINKED LIST IMPLEMENTATION OF QUEUE

In the following section, queue is implemented with the three types of linked list.

5.6.1 Using Singly Linked List

In this implementation, queue is a singly linked list with two pointers ‘rear’ and ‘front’. The pointer ’front’ points the first element in the list and the pointer ‘rear’ points the last elements in the list. Both ‘front’ and ‘rear’ are initialized to NULL. Here, the underflow condition is ‘front’ is equal to NULL. In deletion operation, the node pointed by ‘front’ is deleted and then ‘front’ will point to the next node of the deleted node if available. In insertion operation the new node is inserted after the node pointed by rear i.e. at the last position and ‘rear’ will point the new last node in the list.

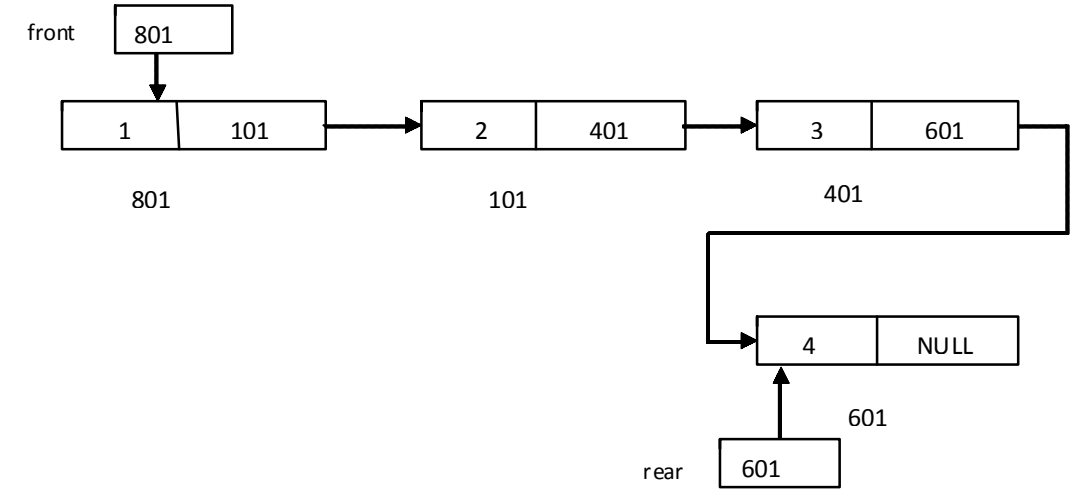


Figure 5.3: (a) Example of a queue implemented using singly linked list

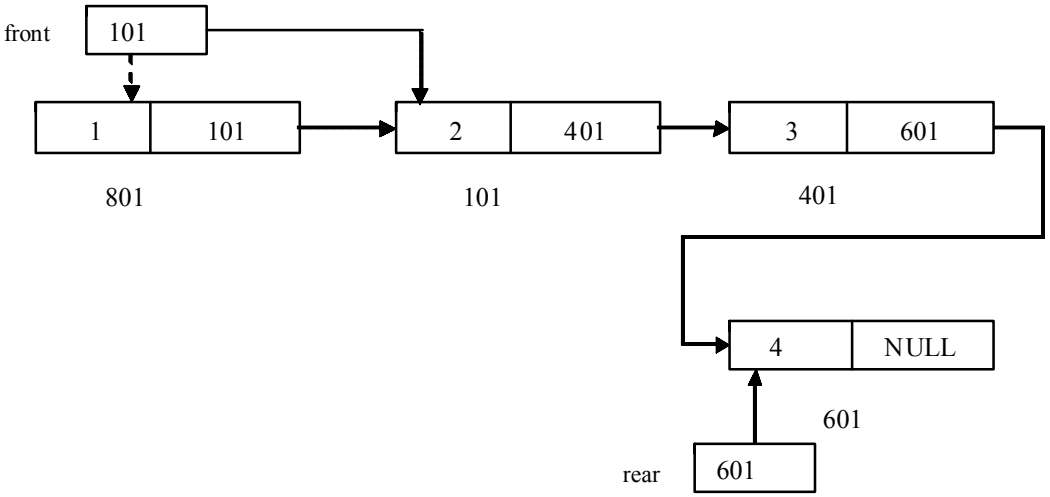


Figure 5.3: (b) Deletion of queue element

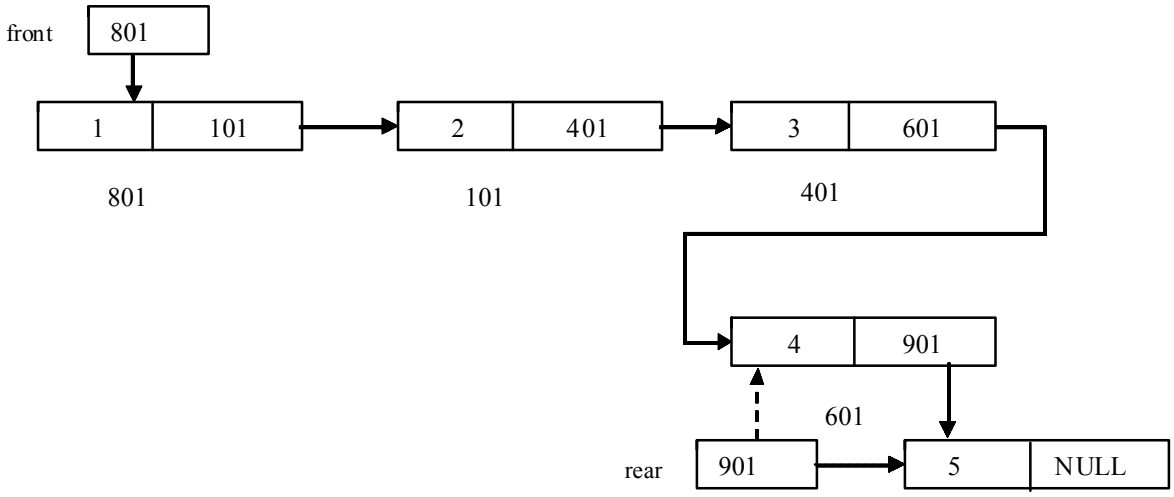


Figure 5.3: (c) Insertion of new queue element

Figure 5.3 (a) shows an example of queue. In figure 5.3(b), the node pointed by the pointer 'front' with data "1" is deleted from the queue which is implemented using singly linked list. In figure 5.3(c), a new node with data "5" is inserted into the queue which is implemented using singly linked list.

Algorithm 5.5: Algorithm for inserting new information to a queue:

ADDRESS(ptr) means address part of the node pointed by the pointer "ptr" which points the next node in the queue implemented using singly linked list.

DATA(ptr) means data part of the node pointed by the pointer "ptr".

"newnode" is the pointer which points the node to be inserted into the queue implemented using singly linked list.

'element' is a variable to store the new element to be inputted into the queue implemented using singly linked list.

Insert (front, rear, element)

Step 1: ALLOCATE MEMORY FOR newnode
Step 2: DATA(newnode) = element
Step 3: ADDRESS(newnode) = NULL
Step 4: IF front == NULL
Step 5: front = newnode
Step 6: rear=newnode
Step 7: END OF IF
Step 8: ELSE
Step 9: ADDRESS (rear) = newnode
Step 10: rear = ADDRESS (rear)
Step 11: END OF ELSE

Algorithm 5.6: Algorithm for deleting information from a queue:

ADDRESS (ptr) means address part of the node pointed by the pointer "ptr" which points the next node in the queue implemented using singly linked list.

"temp" is a pointer to point any node of a queue implemented using singly linked list.

delete(front, rear)

Step 1: IF front == NULL

Step 2: PRINT “Queue is underflow”

Step 3: END OF IF

Step 4: ELSE

Step 6: temp = front

Step 7: front = ADDRESS(front)

Step 8: IF front == NULL THEN

Step 9: rear = NULL

Step 10: END OF IF

Step 11: DEALLOCATE MEMORY FOR temp

Step 12: END OF ELSE

Program 5.3: C program to implement queue using singly linked list

```
#include<stdio.h>
#include<conio.h>

//Structure to create a node of a queue

struct lqueue
    {
    int data;
    struct lqueue*next;
    };
typedef struct lqueue lqueue;

//Function prototypes
int insert(lqueue **,lqueue **,lqueue **,int );
int delet(lqueue **,lqueue **,lqueue **);
int isempty(lqueue *);
void init(lqueue **,lqueue **,lqueue **);
void display(lqueue *);
```

```
void main()
{
    lqueue *qhead, *front, *rear;
    int option,elem,flag;
    char cont;
    clrscr();
    init(&qhead,&front,&rear);
    do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
        printf("\n*****************");
        printf("\nEnter your option::");
        scanf("%d",&option);
        switch(option)
        {

        Case 1: printf("\nEnter the element to be
                inserted into
                the queue::");
                scanf("%d",&elem);
    flag = insert(&qhead,&front,&rear,elem);
        if(flag == 1)
                {
                printf("\n%d is succesfully
                inserted into the queue",elem);
                printf("\nAfter insertion'');
                display(qhead);
                }
                else
                {
```

```
                printf("\nInsertion isnot
                successfull");
                }
                break;
        Case 2: elem = delet(&qhead, &front,&rear);
                if(elem == -99)
                printf("\nDeletion is not possible
                as the queue is empty");
                else
                {
                printf("\n%d is deleted from the
                queue",elem);
                printf("\nAfter deletion'');
                display(qhead);
                }
                break;
        Case 3: display(qhead);
                break;
        default: printf("\nWrong input...try
        again");
        }
            printf("\nDo you want to continue..Press
            'y' or 'Y' to continue");
            cont = getch();
    } while(cont == 'y' || cont == 'Y');
}
//Function to initialize a singly linked list
void init(lqueue **qhead,lqueue **f,lqueue **r)
{
    *qhead = NULL;
    *f = NULL;
    *r = NULL;
}
```

```
//Function to check a queue is underflow or not
int isempty(lqueue *qhead)
{
    if(qhead == NULL)
        return(1);
    else
        return(0);
}

//Function to insert a new node into a queue
int insert(lqueue **qhead,lqueue **f,lqueue **r,int
element)
{
    lqueue *newnode;
    newnode = (lqueue *)malloc(sizeof(lqueue));
    if(newnode == NULL)
        return(0);
    newnode->next = NULL;
    newnode->data = element;
    if(*qhead == NULL)
    {
        *f = newnode;
        *r = newnode;
        *qhead = newnode;
    }
    else
    {
        (*r)->next = newnode;
        (*r) = (*r)->next;
    }
    return(1);
}
```

```
//Function to delete a node from a queue
int delet(lqueue **qhead,lqueue **f,lqueue **r)
{
    int delement;
    lqueue *temp;
    if(isempty(*qhead))
        return(-99);
    else
    {
        temp = *f;
        delement = temp->data;
        (*f) = (*f)->next;
        if(*f == NULL)
        *r = NULL;
        *qhead = *f;
        free(temp);
        return(delement);
    }
}

//Function to display all the elements available
in the queue
void display(lqueue *qhead)
{
    lqueue *temp;
    int i;
    temp = qhead;
    if(isempty(qhead))
        printf("\nThe queue is empty");
    else
    {
        printf("\nthe elements in the queue
        are:\n");
```

```
            while(temp != NULL)
            {
            printf("%5d",temp->data);
            temp = temp->next;
            }
        }
}
```

5.6.2 Using Doubly Linked List

In this implementation queue is a doubly linked list with two pointers 'rear' and 'front'. The pointer 'front' points the first element in the list and the pointer 'rear' points the last elements in the list. Both 'front' and 'rear' are initialized to NULL. Here, the underflow condition is 'front' is equal to NULL. In deletion operation, the node pointed by 'front' is deleted and then 'front' will point to the next node of the deleted node if available. In insertion operation the new node is inserted after the node pointed by rear i.e. at the last position and 'rear' will point the new last node in the list.

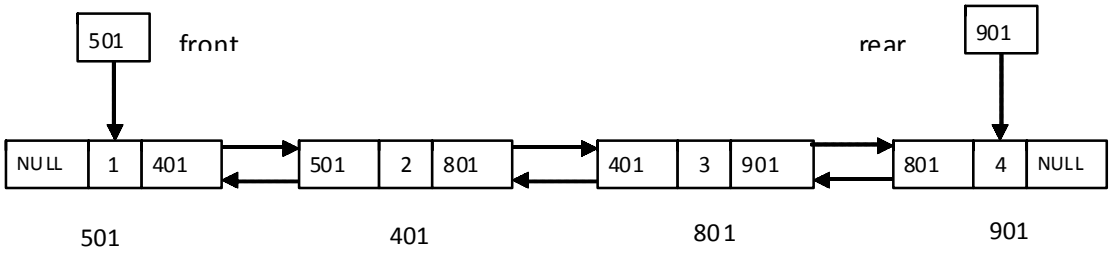


Figure 5.4: (a) Example of a queue implemented using doubly linked list

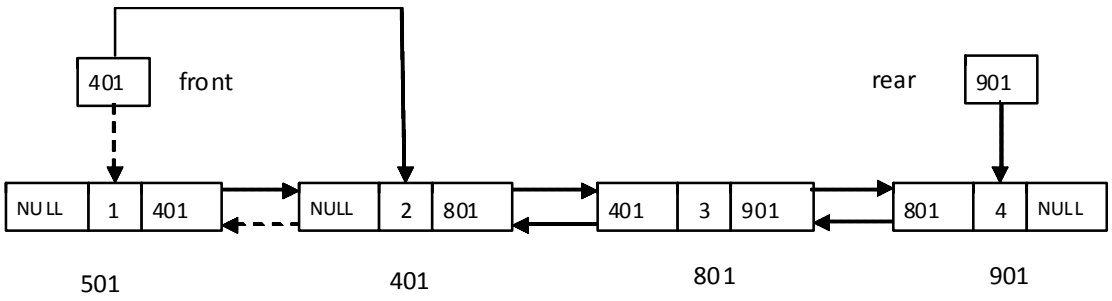


Figure 5.4: (b) Deletion of queue element

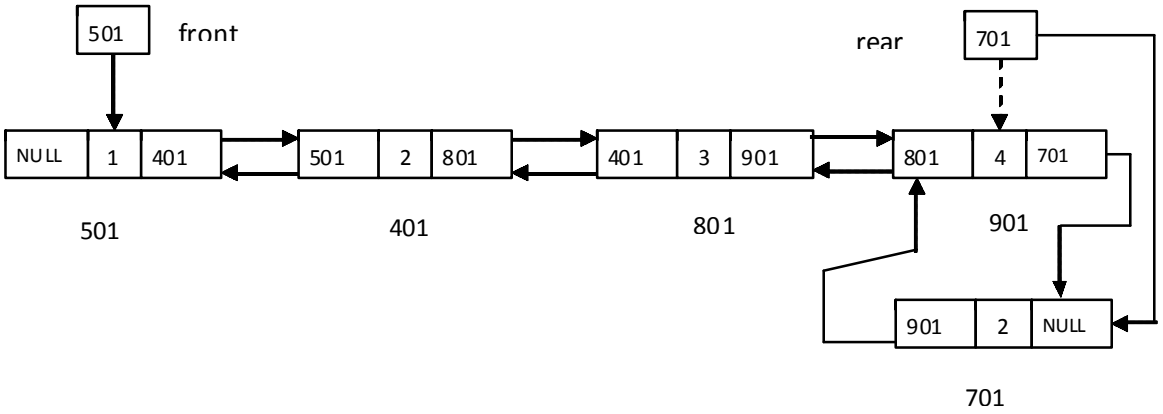


Figure 5.4: (c) Insertion of new queue element

Figure 5.4 (a) shows an example of queue. In figure 5.4(b), the node pointed by the pointer 'front' with data "1" is deleted from the queue which is implemented using doubly linked list. In figure 5.4(c), a new node with data "2" is inserted into the queue which is implemented using doubly linked list.

Algorithm 5.7: Algorithm for inserting new information from a queue

ADDRESSNEXT(ptr) means the address part of a node pointed by the pointer "ptr" which points the next node in the queue implemented using doubly linked list.

ADDRESSPREVIOUS(ptr) means the address part of a node pointed by the pointer "ptr" which points the previous node in the queue implemented using doubly linked list.

DATA(ptr) means the data part of a node pointed by the pointer "ptr" of a queue implemented using doubly linked list.

'element' is a variable to store the new element to be inputted into the queue implemented using doubly linked list.

insert(front, rear, element)

Step 1: ALLOCATE MEMORY FOR newnode

Step 2: DATA(newnode) = element

Step 3: ADDRESSNEXT(newnode) = NULL

Step 4: ADDRESSPREVIOUS(newnode) = NULL

Step 5: IF front == NULL THEN

Step 6: front = newnode

Step 7: rear = newnode

Step 8: END OF IF

Step 9: ELSE

Step 10: ADDRESSPREVIOUS(newnode) = rear

Step 11: ADDRESSNEXT(rear) = newnode

Step 12: rear = ADDRESSNEXT(rear)

Step 13: END OF ELSE

Algorithm 5.8: Algorithm for deleting information from a queue

ADDRESSNEXT(ptr) means the address part of a node pointed by the pointer "ptr" which points the next node in the queue implemented using doubly linked list.

ADDRESSPREVIOUS(ptr) means the address part of a node pointed by the pointer "ptr" which points the previous node in the queue implemented using doubly linked list.

"temp" is a pointer to point any node of a queue implemented using doubly linked list.

delete(front, rear)

Step 1: IF front == NULL THEN

Step 2: PRINT "Queue is underflow"

Step 3: END OF IF

Step 4: ELSE

Step 5: temp = front

Step 6: front = ADDRESSNEXT(front)

Step 7: IF front == NULL

Step 8: drear = NULL

Step 9: END OF IF

Step 10: ELSE

Step 11: ADDRESSPREVIOUS(front) = NULL

Step 12: END OF ELSE

Step 13: DEALLOCATE MEMORY FOR temp

Step 14: END OF ELSE

Program 5.4: C program to implement queue using doubly linked list

```
#include<stdio.h>
#include<conio.h>

//Structure to create a node of a doubly linked
list
struct dlqueue
    {
        int data;
        struct dlqueue *prev;
        struct dlqueue *next;
    };
typedef struct dlqueue dlqueue;

//Function prototypes
int insert(dlqueue **,dlqueue **,dlqueue **,int);
int delet(dlqueue **,dlqueue **,dlqueue **);
int isempty(dlqueue *);
void init(dlqueue **,dlqueue **,dlqueue **);
void display(dlqueue *);

void main()
{
    dlqueue *qhead, *front, *rear;
    int option,elem,flag;
    char cont;
    clrscr();
    init(&qhead,&front,&rear);
    do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
```

```
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
        printf("\n*****************");
        printf("\nEnter your option::");
        scanf("%d",&option);
        switch(option)
        {
        Case 1: printf("\nEnter the element to be
        inserted into the queue::");
                scanf("%d",&elem);
    flag = insert(&qhead,&front,&rear,elem);
        if(flag == 1)
                {
                printf("\n%d is succesfully
                inserted into the queue",elem);
                printf("\nAfter insertion'');
                display(qhead);
                }
                else
                {
                printf("\nInsertion isnot
                successfull");
                }
                break;
        Case 2: elem = delet(&qhead,&front,&rear);
                if(elem == -99)
                printf("\nDeletion is not possible
                as the queue is empty");
                else
                {
                printf("\n%d is deleted from the
                queue",elem);
                printf("\nAfter deletion ");
```

```
                display(qhead);
                }
                break;
        Case 3: display(qhead);
                break;
        default:printf("\nWrong input...try
        again");
        }
        printf("\nDo you want to continue...Press
        'y' or 'Y' to continue");
        cont = getch();
    } while(cont == 'y' || cont == 'Y');
}

//Function to initialize a doubly linked list
void init(dlqueue **qhead,dlqueue **f,dlqueue **r)
{
    *qhead = NULL;
    *f = NULL;
    *r = NULL;
}

//Function to check a queue is underflow or not
int isempty(dlqueue *qhead)
{
    if(qhead == NULL)
        return(1);
    else
        return(0);
}

//Function to insert a new node into a queue
int insert(dlqueue **qhead,dlqueue **f,dlqueue **r,
int element)
```

```
{
    dlqueue *newnode;
    newnode = (dlqueue *)malloc(sizeof(dlqueue));
    if(newnode == NULL)
        return(0);
    newnode->next = NULL;
    newnode->prev = NULL;
    newnode->data = element;
    if(*qhead == NULL)
    {
        *f = newnode;
        *r = newnode;
        *qhead = newnode;
    }
    else
    {
        newnode->prev = *r;
        (*r)->next = newnode;
        (*r) = (*r)->next;
    }
    return(1);
}

//Function to delete a node from a queue
int delet(dlqueue **qhead,dlqueue **f,dlqueue **r)
{
    int delement;
    dlqueue *temp;
    if(isempty(*qhead))
        return(-99);
    else
    {
        temp = *f;
```

```
            delement = temp->data;
            (*f) = (*f)->next;
            if(*f == NULL)
              *r = NULL;
        else
            (*f)->prev = NULL;

            *qhead = *f;
            free(temp);
            return(delement);
        }
    }

    //Function to display all the elements available
    in a queue
    void display(dlqueue*qhead)
    {
        dlqueue *temp;
        int i;
        temp = qhead;
        if(isempty(qhead))
            printf("\nThe queue is empty");
        else
        {
            printf("\nthe elements in the queue
            are:\n");
            while(temp != NULL)
            {
                printf("%5d",temp->data);
                temp = temp->next;
            }
        }
    }
```

5.6.3 Using Circular Linked List

In this implementation, queue is a circular linked list with two pointers 'rear' and 'front'. The pointer 'front' points the first element in the list and the pointer 'rear' points the last elements in the list. Both 'front' and 'rear' are initialized to NULL. The underflow condition is same with the singly linked list implementation. Here, the address field of the node which is pointed by 'rear' points the node which is pointed by 'front'.

In deletion operation, the node pointed by 'front' is deleted and then 'front' will point to the next node of the deleted node if available otherwise 'front' and 'rear' will be NULL. The address field of the node pointed by 'rear' will point the node which is pointed by 'front' after deletion operation. In insertion operation the new node is inserted after the node pointed by 'rear' i.e. at the last position and then 'rear' will point the new last node in the list. The address part of the new node will point the node pointed by front.

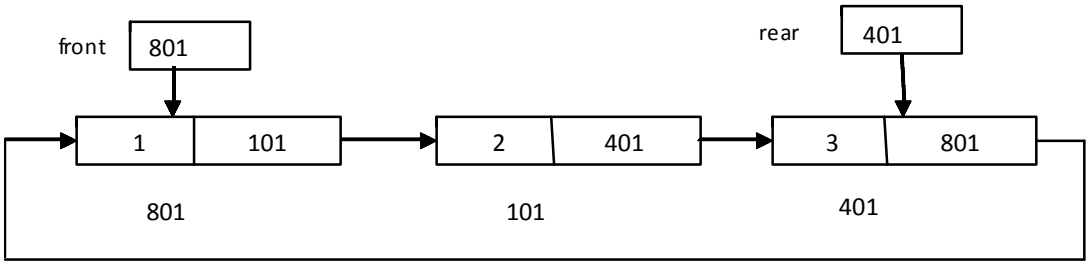


Figure 5.5: (a) Example of a queue implemented using circular linked list

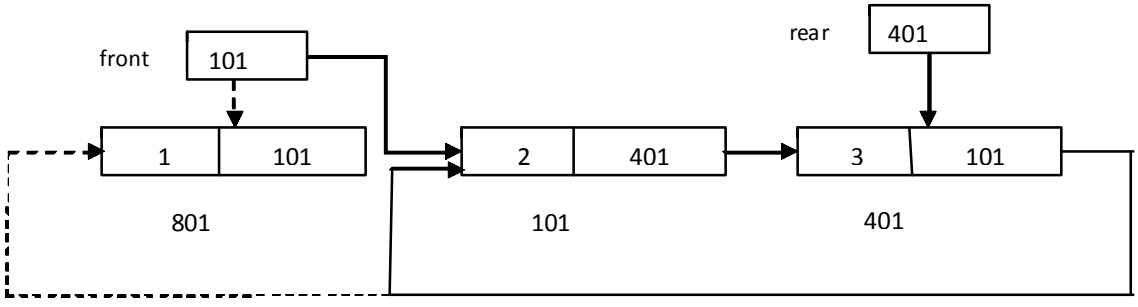


Figure 5.5: (b) Deletion of queue element

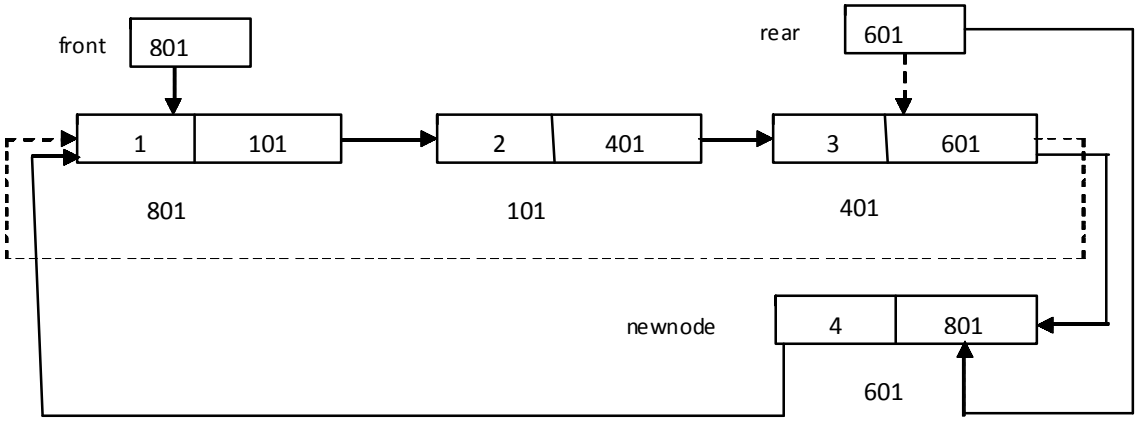


Figure 5.5: (c) Insertion of new queue element

Figure 5.5 (a) shows an example of queue. In figure 5.5(b), the node pointed by the pointer 'front' with data "1" is deleted from the queue which is implemented using circular linked list. In figure 5.5(c), a new node with data "4" is inserted into the queue which is implemented using circular linked list.

Algorithm 5.9: Algorithm for inserting new information from a queue

ADDRESS(ptr) means address part of the node pointed by the pointer "ptr" which points the next node in the queue implemented using circular linked list.

DATA(ptr) means data part of the node pointed by the pointer "ptr".

"newnode" is the pointer which points the node to be inserted into the queue implemented using circular linked list.

'element' is a variable to store the new element to be inputted into the queue implemented using circular linked list.

insert(front, rear, element)

Step 1: ALLOCATE MEMORY FOR newnode

Step 2: DATA(newnode) = element

Step 3: ADDRESS(newnode) = NULL

Step 4: IF front == NULL THEN

Step 5: front = newnode

Step 6: rear = newnode

Step 7: ADDRESS(rear) = front

Step 8: END OF IF

Step 9: ELSE

Step 10: ADDRESS(newnode) = front

Step 11: ADDRESS(rear) = newnode

Step 12: rear = ADDRESS(rear)

Step 13: END OF ELSE

Algorithm 5.10: Algorithm for deleting information from a queue

ADDRESS(ptr) means address part of the node pointed by the pointer "ptr" which points the next node in the queue implemented using circular linked list.

"temp" is a pointer to point any node of a queue implemented using circular linked list.

delete(front, rear)

int delement;

Step 1: IF front == NULL THEN

Step 2: PRINT"Queue underflow"

Step 3: END OF IF

Step 4: ELSE

Step 5: temp = front

Step 6: IF ADDRESS(front) == front THEN

Step 7: rear = NULL

Step 8: front = NULL

Step 9: END OF IF

Step 10: ELSE

Step 11: front = ADDRESS(front)

Step 12: ADDRESS(rear) = front

Step 13: END OF ELSE

Step 14: DEALLOCATE MEMORY FOR temp

Step 15: END OF ELSE

Program 5.5: C program to implement queue using circular linked list

```
#include<stdio.h>
#include<conio.h>

//Structure to create a node of a queue
struct crqueue
    {
    int data;
    struct crqueue*next;
    };
typedef struct crqueue crqueue;

//Function prototypes
int insert(crqueue **,crqueue **,crqueue **,int);
int delet(crqueue **,crqueue **,crqueue **);
int isempty(crqueue *);
void init(crqueue **,crqueue **,crqueue **);
void display(crqueue *);

void main()
{
    crqueue*qhead,*front,*rear;
    int option,elem,flag;
    char cont;
    clrscr();
    init(&qhead,&front,&rear);
    do
    {
        printf("\n*****************");
        printf("\n1.Insertion     *");
        printf("\n2.Deletion      *");
        printf("\n3.Display       *");
```

```
printf("\n*****************");
printf("\nEnter your option::");
scanf("%d",&option);
switch(option)
{
    Case 1: printf("\nEnter the element to
            be inserted into the queue::");
            scanf("%d",&elem);
flag = insert(&qhead,&front,&rear,elem);
        if(flag == 1)
            {
            printf("\n%d is succesfully
            inserted into the queue",elem);
            printf("\nAfter insertion'');
            display(qhead);
            }
            else
            {
            printf("\nInsertion isnot
            successfull");
            }
            break;
        Case 2: elem = delet(&qhead,&front,
            &rear);
            if(elem == -99)
            printf("\nDeletion is not
            possible as the queue is empty");
            else
            {
            printf("\n%d is deleted from
            the queue",elem);
            printf("\nAfter deletion'');
                display(qhead);
```

```
                    }
                    break;
            Case 3: display(qhead);
                    break;
        default:printf("\nWrong input...try
        again");
    }
        printf("\nDo you want to continue.
        Press 'y' or 'Y' to continue");
        cont = getch();
    } while(cont == 'y' || cont == 'Y');
}

//Function to initialize a circular linked list
void init(crqueue **qhead,crqueue **f,crqueue **r)
{
    *qhead = NULL;
    *f = NULL;
    *r = NULL;
}

//Function to check a queue is underflow or not
int isempty(crqueue *qhead)
{
    if(qhead == NULL)
        return(1);
    else
        return(0);
}

//Function to insert a new node into the queue
int insert(crqueue **qhead,crqueue **f,crqueue **r,
int element)
```

```
{
    crqueue *newnode;
    newnode = (crqueue *)malloc(sizeof(crqueue));
    if(newnode == NULL)
        return(0);
    newnode->data = element;
    if(*qhead == NULL)
    {
        *f = newnode;
        *r = newnode;
        (*r)->next = *f;
        *qhead = newnode;
    }
    else
    {
        newnode->next = *f;
        (*r)->next = newnode;
        (*r) = (*r)->next;
    }
        return(1);
}

//Function to delete a node from the queue
int delet(crqueue **qhead,crqueue **f,crqueue **r)
{
    int delement;
    crqueue *temp;
    if(isempty(*qhead))
        return(-99);
    else
    {
        temp = *f;
        delement = temp->data;
```

```
            (*f) = (*f)->next;

                if((*f)->next == *f)
                {
                *r = NULL;
                *f = NULL;
                }
            else
                (*r)->next = *f;

            *qhead = *f;
            free(temp);
            return(delement);
        }
    }

    //Function to display all the elements available in
    a queue
    void display(crqueue *qhead)
    {
        crqueue *temp;
        int i;
        temp = qhead;
        if(isempty(qhead))
            printf("\nThe queue is empty");
        else
        {
            printf("\nthe elements in the queue
            are:\n");
            printf("%5d",temp->data);
            temp = temp->next;
            while(temp != qhead)
                {
```

```
                printf("%5d",temp->data);
                temp = temp->next;
            }
        }
    }
```

5.7 PRIORITY QUEUES

Priority queue is a type of queue where each element has a priority value and the deletion of the elements is dependent upon the priority value. In case of max-priority queue, the highest priority value will be deleted first and in case of min-priority queue the element which has the minimum priority value will be deleted first.

One application of max-priority queue is to schedule jobs on a shared computer. A min-priority queue can be used in an even-driven simulator. We will learn about priority quenes in detail in Unit 13.

5.8 APPLICATION OF QUEUE

There are several applications of queue available in computer system. Some of these are given as follows:

- In printers, queue is used to print the different files.
- Queue is used to access files from a disk system.
- In a multiprogramming environment, queue is used for CPU scheduling or job scheduling of operating system.
- In any type of ticket reservation system, queue can be used for issuing tickets to the customers.
- Queue is used in the implementation of breadth first traversal of graph.

Queue is used in many other real world systems which are used in some scientific research, military operations etc.



CHECK YOUR PROGRESS

Q.1: Multiple choice questions:

I) "FRONT==REAR" pointer refers to–

A) Empty stack B) Empty queue

C) Full stack D) Full queue

II) Last in last out–

A) Array B) Stack

C) Queue D) None of the above

III) Queue cannot be used for–

A) the line printer B) access to disk storage

C) function call D) Both A and B

IV) Insertion into an queue is done at the–

A) Front end B) Rear end

C) Top end D) Both A and C

V) In case of a circular queue, if rear is 10 and front is 20 and the maximum number of element can be stored is N then at the current situation the total number of element stored in the queue is–

A) N–9 B) N–19

C) N–20 D) None of the above

Q.2: Fill in the blanks:

I) ..... is used to implement Breadth first search algorithm.

II) Deletion is done at ..... end in a queue.

III) At rear end ..... is done in case of a queue.

IV) If in a circular queue, front==(rear+1)%(size of the queue) then it means that .....

V) Printing files is an application of .....

Q.3: State whether the following statements are true or false:

I) Reversing strings can be done using queue

II) Conversion of infix notation to postfix notation can be done using queue.

III) Circular Queue can be implemented using circular linked list.

IV) Queue is also known as FIFO.

V) Queue is used in job scheduling of operating system.



5.9 LET US SUM UP

- Queue is a linear data structure in which the element will be removed first which is inserted first. Queue is also known as first in first out (FIFO) data structure.
- A queue has two ends which are front end and rear end. Insertion is to take place at rear end and deletion is to take place at front end.
- Queue can be implemented using both array and linked list.
- Circular queue is a type of array implementation of queue where the array is imagined as a circular data structure i.e. the next subscript value of the last subscript value of the array is '0'.
- Printing files, accessing files from a disk system, job scheduling by operating system etc. are some of the applications of queue.
- Priority queue is a type of queue where the deletion of elements depends upon some priority value associated with each element in the queue.
- In case of max-priority queue, the element that has the highest priority value will be deleted first and in case of min-priority queue the element that has the lowest priority value will be deleted first.



5.10 FURTHER READING

1) Horowitz, E., Sahni, S. & Anderson-Freed, S. (1983). *Fundamentals of Data Structures*. (Vol. 20). Rockville, MD: Computer Science Press.

2) Langsam, Y., Augenstein, M. & Tenenbaum, A. M. (1996). *Data Structures Using C and C++*. (Vol. 2). New Jersey: Prentice Hall.



5.11 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: I) (B), II) (C), III) (C), IV) (B), V) (A)

Ans. to Q. No. 2: I) Queue, II) Front, III) Insertion,
IV) The queue is overflow, V) queue

Ans. to Q. No. 3: I) False, II) False, III) False, IV) True, V) True.



5.12 MODEL QUESTIONS

Q.1: Define queue.

Q.2: What are the different types of queue?

Q.3: Why is circular queue needed? Implement circular queue in C.

Q.4: Write a C program to implement queue using circular linked list.

Q.5: Write down algorithms for insert and delete operation on a queue.

Q.6: Give two applications of queue.

Q.7: Where are priority queues used in real life situations?

*** ***** ***

UNIT 6: SEARCHING

UNIT STRUCTURE

6.1 Learning Objectives

6.2 Introduction

6.3 Searching

6.4 Types of Searching

6.4.1 Linear Search

6.4.2 Binary Search

6.5 Let Us Sum Up

6.6 Further Reading

6.7 Answers to Check Your Progress

6.8 Model Questions

6.1 LEARNING OBJECTIVES

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

- describe searching techniques
- explain linear seach and binary search
- search an element in both sorted and unsorted list
- compare the advantages and disadvantages of linear and binary search techniques.

6.2 INTRODUCTION

In our day-to-day life there are various applications in which the process of searching is to be carried out. Searching a name of a person from the given list, searching a specific card from the set of cards, etc., are a few examples of searching.

In computer science, there are many applications of searching process. Many advanced algorithms and data structures have been devised for the sole purpose of making searches more efficient. As the data sets become larger and larger, good search algorithms will become more important. While solving a problem, a programmer may need to search a

value in an array. This unit will focus on different searching tehniques like linear searching and binary searching. In the next unit, we will focus on different sorting techniques.

6.3 SEARCHING

Searching is a technique of finding an element from a given data list or set of elements. To illustrate the search process, let us consider an array of 20 elements. These data elements are stored in successive memory locations. We need to search a particular element from the array. Let **e** be the element to be searched. The element **e** is compared with all the elements in the array starting from the first element till the last element. When the exact match is found then the search process is terminated. In case no such element exists in the array, the process of searching should be abandoned. Suppose, we are to search the element 8 in the array. In that case the given element is present in the array and thus the search process is said to be successful as per Figure 6.1(a).

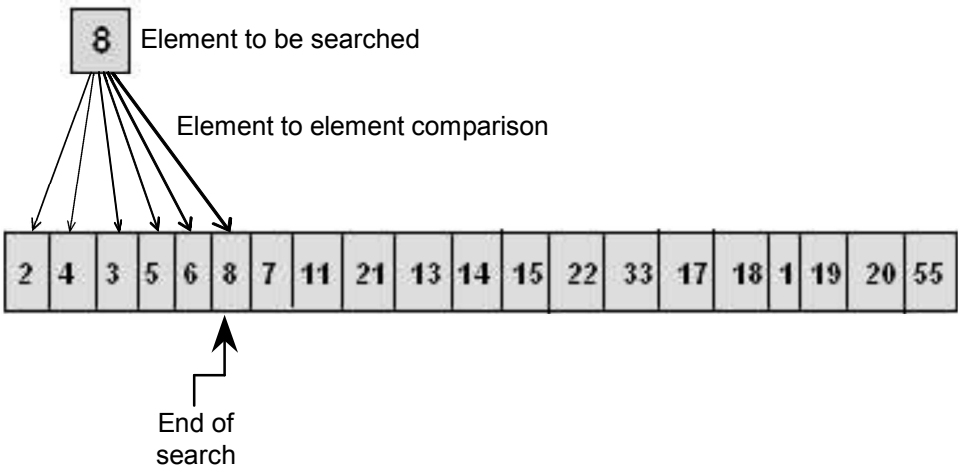


Figure 6.1(a): Successful search

Again let us consider our element to be searched is 66. The search is said to be unsuccessful as the given element does not exist in the array as per Figure 6.1(b).

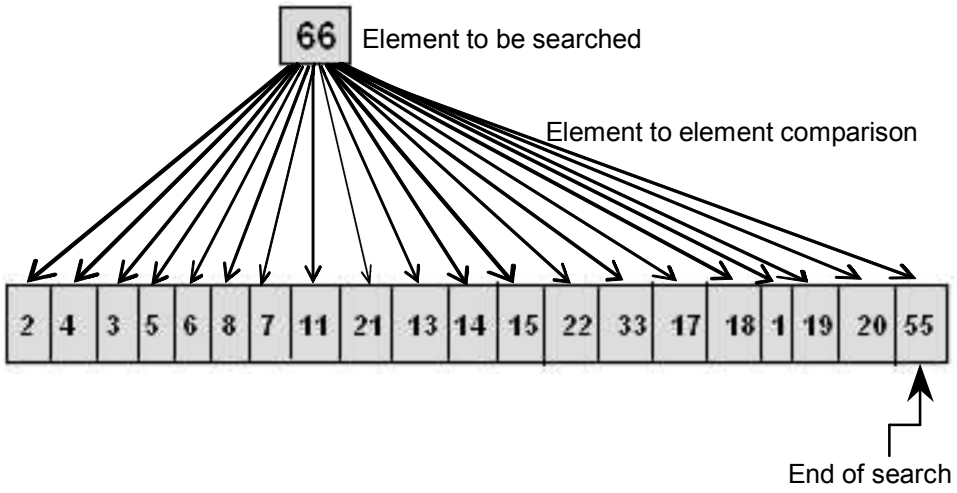


Figure 6.1(b): Unuccessful search

6.4 TYPES OF SEARCHING

The two most important searching techniques are:

- Linear or Sequential search
- Binary search

6.4.1 Linear Search

Linear search, also known as *sequential search*, is a technique in which the array traversed sequentially from the first element until the value is found or the end of the array is reached. While traversing, each element of the array is compared with the value to be searched, and if the value is found, the search is said to be successful.

Linear search is one of the simplest searching techniques. Though, it is simple and straightforward, it has some limitations. It consumes more time and reduces the retrieval rate of the system. The linear or sequential name implies that the items are stored in systematic manner. It can be applied on sorted or unsorted linear data structure.

Algorithm 6.1: Algorithm of Linear Search

Let us start with an array or list, L which may have the item in question.

Step 1: If the list L is empty, then the list has nothing. The list does not have the item in question. Stop here.

Step 2: Otherwise, look at all the elements in the list L.

Step 3: For each element:

If the element equals the item in question, the list conains the item in question. Stop here. Otherwise, go onto next element.

Step 4: The list does not have the item in question.

Program 6.1: Program to search an element in an array applying linear search

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int arr[50],n,i,item;
    printf(''How many elements you want to enter
    in the array :'');
    scanf(''%d'',&n);
        for(i=0; i<n; i++)
        {
        printf(''\nEnter element%d :'', i+1);
        scanf(''%d'',&arr[i]);
        }
    printf(''\nEnter the element to be searched :'');
    scanf(''%d'',&item);
    for(i=0; i<n; i++)
    {
        // searched item is compared with array element
        if(item == arr[i])
        {
            printf(''\n%d found at position%d\n'',
            item, i+1);
```

```
                break;
            }
        }
        if(i == n)
            printf(''\nItem%d not found in array\n'',
            item);
        getch();
}
```

In the above program suppose the user enters few numbers using the first *for* loop (Number of elements should be less or equal to 50 as the array size is 50). The element which is to be searched is stored in the variable *item*. By using a second *for* loop the element is compared with each element of the array. If the element in *item* variable is matched with any of the element in the array then the location is displayed. Otherwise *item* is not present in the array.

Analysis of Linear Search: We have carried out linear search on lists implemented as arrays. Whether the linear search is carried out on lists implemented as arrays or linked list or on files, the criteria part in performance is the comparison loops (i.e., **Step 3** of the algorithm). Obviously the fewer the number of comparisons, the sooner the algorithm will terminate.

The fewest possible comparisons is equal to **1** when the required item is the first item in the list and which will be the *best case*. Thus, in this case, the complexity of the algorithm is **O(1)**. The maximum comparisons will be equal to **n** (total numbers of elements in the list) when the required item is the last in the list or not present in the list. This will be the worst case. In both cases, the average complexity of linear search is **O(n)**.

If the required item is in position **i** in the list, then only **i** comparisons are required. Hence, in average case, the number of comparisons done by linear search will be:

$$\frac{1+2+3+.....+i+.....+N}{N}$$
$$=\frac{N(N+1)}{2\times N}$$
$$=\frac{N+1}{2}$$



CHECK YOUR PROGRESS

Q1: Select the appropriate option for each of the following questions:

i) Linear search is efficient in case of–

a) short list of data b) long list of data

c) both (a) and (b) d) none of these

ii) The process of finding a particular record is called–

a) indexing b) searching

c) sorting d) none of these

6.4.2 Binary Search

The binary search approach is different from the linear search. The binary search technique is used to search for a particular element in a sorted array or list. In this technique, two partitions of lists are made and then the given element is searched and hence, it is known as binary search.

Let us consider a list which is sorted in ascending order. It would work to search from the beginning until the *item* is found or the end is reached, but it makes more sense to remove as much of the working data set as possible so that the item is found more quickly. If we started at the middle of the list we could determine which half the *item* is in (because the list is sorted). This effectively divides the working range in half with a single test. By repeating the procedure, the result is a highly efficient search algorithm called *binary search*.

The binary search is based on the **divide-and-conquer** approach. In this technique, the element to be searched (say, *item*)

is compared with the middle element of the array. If *item* is equal to the middle element, then search is successful. If the *item* is smaller than the middle element, then *item* is searched in the segment of the array before the the middle element. However, if the *item* is greater than the middle element, *item* is searched in the array segment after the middle element. This process will be in iteration until the element is found or the array segment is reduced to a single element that is not equal to *item*.

To illustrate the process of binary search, let us assume a sorted array of 11 elements as shown in figure 6.2(a). Suppose we want to search the element 56 from the array of elements.

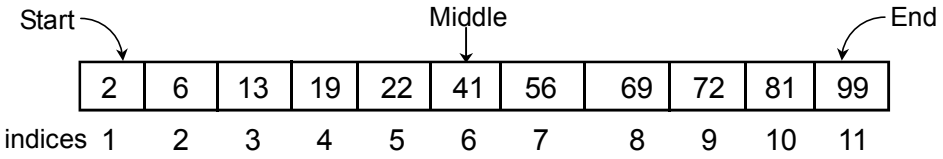


Figure 6.2(a): Sorted array of elements

For this we will take 3 variables *Start*, *End* and *Middle*, which will keep track of the status of start, end and middle value of the portion of the array, in which we will search the element. The value of the middle will be as :

$$\text{Middle} = \frac{\text{Start} + \text{End}}{2}$$

Initially, *Start* =1, *End* = 11 and the *Middle* = *(1 + 11)/2 = 6*. The value at index 6 is 41 and it is smaller than the target value i.e., (56). The steps are as follows:

Step 1: The element 2 and 99 are at Start and End positions respectively.

Step 2: Calculate the middle index which is as:

Middle = (Start + End)/2

Middle = (1+11)/2

Middle = 6

Step 3: The key element 56 is to be compared with the *Middle* value. If the key is less than the value at *Middle* then the

key element is present in the first half else in other half; in our case, the key is on the right half. Now we have to search only on right half.

Hence, Now *Start = Middle + 1 = 6 + 1 = 7.*

End will be same as earlier.

Step 4: Calculate the middle index of the second half

Middle = (Start + End)/2

Middle = (7 + 11)/2

Middle = 9

Step 5: Again, the *Middle* divides the second half into the two parts, which is shown in figure 6.2(b).

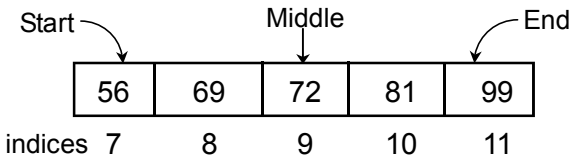


Figure 6.2(b): After divining array in middle

Step 6: The key element 56 is lesser than the value at *Middle* which is 72, hence it is present in the left half i.e., towards the left of 72. Hence now, *End = Middle – 1 = 8*. *Start* will remain 7.

Step 7: At last, the *Middle* is calculated as:

Middle = (Start + End)/2

Middle = (7 + 8)/2

Middle = 7

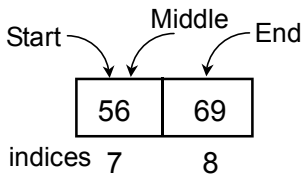


Figure 6.2(c): After further division of array in middle

Step 8: Now if we compare our key element 56 with the value at *Middle*, then we see that they are equal. The key element is searched successfully and it is in 7th location.

Algorithm 6.2: Binary Search Algorithm

Input: (List, Key)

Step 1: WHILE (List is not empty) DO
(Select "middle" entry in list as test entry

Step 2: IF (Key = test entry) THEN (Output ("Found it") Stop)

Step 3: IF (Key < test entry) THEN (apply BinarySearch to part of list preceding test entry)

Step 4: IF (Key > test entry) THEN (apply BinarySearch to part of list following test entry)

Output: ("Not Found")

Analysis of Binary Search: The binary search algorithm reduces the array to one-half in each iteration. Therefore, for an array containing **n** elements, there will be $\log_2 n$ iterations. Thus, the complexity of binary search algorithm is $O(\log_2 n)$. This complexity will be the same irrespective of the position of the element, even if the element is not present in the list.

Program 6.2: Program to search an element using binary search

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int arr[20],start,end,middle,n,i,item;
    clrscr();
    printf(''How many elements you want to enter
    in the array:'');
    scanf(''%d'',&n);
    for(i=1;i<=n;i++)  {
        printf(''Enter element%d:'', i);
        scanf(''%d'',&arr[i]);
    }
    printf(''\nEnter the element to be searched:'');
    scanf(''%d'',&item);//element to be searched
    is stored in variable item
```

```
    start = 1;
    end = n;
    middle = (start + end)/2;
    while(item!=arr[middle] && start<=end)
    {
        if(item>arr[middle])
        start = middle+1;
        else
        end = middle-1;
        middle = (start+end)/2;
    }
    if(item==arr[middle])
        printf(''\n%d is found at position%d\n'',
        item,middle);
    if(start>end)
        printf(''\n%d is not found in array\n'',
        item);
    getch();
}
```

Program 6.3: Program to search a name from a list of 10 names using binary search

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    int start=1, end=10, mid, i, flag=0, value;
    char name[15][10],nm[15];
    clrscr();
    printf(''\nEnter 10 names:\n'');
    for(i=1; i<11; i++)
        scanf(''%s'', & name[i]);
```

```
    printf(''\nEnter the name to search:'');
    scanf(''%s'', & nm);
    mid = (start+end)/2;
    while(strcmp(nm,name[mid])!=0 && start<=end)
    {
        value=strcmp(nm,name[mid]);
        if(value>0)
        {
            start=mid+1;
            mid=(start+end)/2;
        }
        else
        {
            end=mid-1;
            mid=(start+end)/2;
        }
    }
        if(strcmp(nm,name[mid])==0)
    {
        flag=1;
    }
    if(flag==1)
        printf(''\nThe name %s is found
        successfully'',nm);
    else
        printf(''\nThe name%s is not found'',nm);
    getch();
}
```

Advantages and Disadvantages: Linear search algorithm is easy to write and efficient for short lists. It does not require sorted data. However, it is lengthy and time consuming for long lists. There is no way of quickly establishing that the required item is not in the list or of finding all occurances of a required item at one place. The linear

search situation will be in worst case if the element is at the end of the list.

A **binary search** halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. The advantage of a binary search over a linear search is astounding for large numbers. In binary search, array/list of elements should be in shorted order.

For an array of a million elements, binary search, O(logn), will find the target element with a worst case of only 20 comparisons. Linear search, O(n), on average will take 5,00,000 comparisons to find the element.



CHECK YOUR PROGRESS

Q.2: What are the advantages of sequential search?

Q.3: State whether the following statements are true (T) or false (F):

i) Element should be in sorted order in case of binary search.

ii) Binary search cannot be applied in character array.

iii) Sequential search is worst case and average case O(n).

iv) Binary search is worst case O(logn).

Q.4: In the following function code which type of search algorithm is applied?

```
int find (int a[ ], int n, int x)
{
    int i;
    for (i = 0; i<n; i++)
    {
        if (a[i] == x )
            return i;
    }
    return 0;
}
```

Q.5: Write a function in C to find an element **x** in an array of **n** elements where the array, size of the array and the element is passed as arguments.



6.5 LET US SUM UP

- The process of finding the occurance of a particular data item in a list is known as **searching**.
- If the search element is present in the collected elements or array then the search process is said to be successful. The search is said to be unsuccessful if the given element does not exist in the array.
- Linear search is a search algorithm that tries to find a certain value in a set of data. It operates by checking every element of a list (or array) one at a time in sequence until a match is found.
- The linear search was seen to be easy to implement and relatively efficient to use for small lists. But very time consuming for long unsorted lists.
- Binary search only works on sorted lists (or arrays). It finds the middle element, makes a comparison to determine whether the desired value comes before or after it and then searches the remaining half in the same manner.
- The binary search is an improvement, in that it eliminates half the list from consideration at each iteration. The prerequisite for it is that the list should be sorted order.



6.6 FURTHER READING

1) Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc..
2) Robert, L. K., Tondo, C. L., & Leung, B. P. (2002). *Data Structures And Program Design*.



6.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: i) (a), ii) (b)

Ans. to Q. No. 2: Sequential search is easy to implement and relatively efficient to use for small lists. It does not require a sorted list of elements or data.

Ans. to Q. No. 3: i) True, ii) False, iii) True, iv) True

Ans. to Q. No. 4: Sequential (Linear) search is applied.

Ans. to Q. No. 5:

```
int find (int a[ ], int n, int x )
{
    int i = 0;
    while(i < n)
    {
        int mid = (n+i) / 2;
        if (a[mid] < x)
            n = mid;
        else if (a[mid] > x )
            i = mid+1;
        else
            return mid;
        }
        return 0;
}
```



6.8 MODEL QUESTIONS

Q.1: Write a program to find the given number in an array of 20 elements. Also display how many times a given number exists in the list.

Q.2: What is searching? What are the advantages and disadvantages of sequential search technique?

Q.3: What are the advantages of binary search technique?

Q.4: Differentiate between linear and binary search.

Q.5: Write a program to demonstrate binary search. Use integer array and store 10 elements. Find the given element.

Q.6: Write down the best, worst and average case time complexity of Sequential search.

Q.7: Write a program to demonstrate successful and unsuccessful search. Display appropriate messages.

*** ***** ***

UNIT 7: SORTING

UNIT STRUCTURE

7.1 Learning Objectives

7.2 Introduction

7.3 Sorting

7.4 Insertion Sort

7.5 Selection Sort

7.6 Bubble Sort

7.7 Quick Sort

7.8 Let Us Sum Up

7.9 Further Reading

7.10 Answers to Check Your Progress

7.11 Model Questions

7.1 LEARNING OBJECTIVES

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

- define sorting
- explain insertion sort algorithm
- describe selection sort algorithm
- explain bubble sort algorithm
- describe quick sort algorithm
- compute the complexity of the above sorting algorithms.

7.2 INTRODUCTION

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output.

We have leaned about binary search and linear searching techniques in the previous unit. This unit, we will introduce you to the fundamental concepts of sorting and we shall discuss about the various sorting algorithms i.e., insertion sort, selection sort, bubble sort and quick sort including their complexity. The next block of units will deal with the concept of different types of trees and graphs.

7.3 SORTING

Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabatically, with character data. In real life we come accross several examples of sorted information. For example, in telephone directory the names of the subscribers and their phone numbers are written in alphabetical order. The records of the list of these telephone holders are to be sorted by their names. By using the directory, we can access the telephone number and the address of the subscriber very easily. In the same way in dictionary the words are placed in lexicographical order which is nothing but a sorting.

Let L be a list of *n* elements L_1, L_2, ..., ..., ..., L_n in memory. *Sorting* a list refers to the operation of rearranging the contents of L so that they are increasing in order (numerically or lexicographically), that is,

$$L_1 \leq L_2 \leq L_3 \leq \ldots \ldots \ldots L_n$$

Since L has *n* elements, there are *n!* ways that the contents can appear in L. These ways correnspond precisely to the n! permutations of 1, 2, ..., ..., ...n.

Let us assume that an array ELEMENT contains 10 elements as follows:

ELEMENT: 56, 37, 43, 21, 34, 16, 59, 25, 90, 64

After sorting, ELEMENT must appear in memory as follows:

ELEMENT: 16, 21, 25, 34, 37, 43, 56, 59, 64, 90

Since ELEMENT consists of 10 elements, there are 10!=3628800 ways that the numbers 16, 21, ..., ..., ..., 90 can appear in ELEMENT.

7.4 INSERTION SORT

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and it is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one.

Let A be an array with *n* elements A[0], A[1], ..., ..., ..., A[N – 1] in memory. The insetrtion sort algorithm scans the the array from A[0] to A[N –1], and the process of inserting each element in proper place is as follows:

Pass 1: A[0] by itself is sorted because of one element.

Pass 2: A[1] is inserted either before or after A[0], so that A[0], A[1] are sorted.

Pass 3: A[2] is inserted into its proper place in A[0], A[1], i.e., before A[0], between A[0] and A[1], or after A[1], so that: A[0], A[1], A[2] are sorted.

Pass 4: A[3] is inserted into its proper place in A[0], A[1], A[2] so that: A[0], A[1], A[2], A[3] are sorted.

.....

.....

Pass N: A[N – 1] is inserted into its proper place in A[0], A[1], ..., ..., ..., A[N – 2] so that: A[0], A[1], ..., ..., ..., A[N – 1] is sorted.

Insertion sort algorithm is frequently used when *n* is small.

The element inserted in the proper place is compared with the previous elements and placed in between the i^{th} element and $(i + 1)^{th}$ element if,

element $\leq i^{th}$ element

element $\geq (i+1)^{th}$ element

Let us take an example of the following elements:

	82	42	49	8	92	25	59	52
Pass 1	**82**	42	49	8	92	25	59	52
Pass 2	82	**42**	49	8	92	25	59	52
Pass 3	42	82	**49**	8	92	25	59	52
Pass 4	42	49	82	**8**	92	25	59	52
Pass 5	8	42	49	82	**92**	25	59	52
Pass 6	8	42	49	82	**92**	25	59	52
Pass 7	8	25	42	49	82	92	**59**	52
Pass 8	8	25	42	49	59	82	92	**52**
Sorted elements	8	25	42	49	59	82	92	52

Finally, we get the sorted array.The following program uses the insertion sort technique to sort a list of numbers.

Program 7.1: Program of sorting using insertion sort

```
#include<stdio.h>
#include<conio.h>

void main()
{
    int a[25], i, j, k, n;
    clrscr();
    printf("Enter the number of elements :'');
    scanf("%d",&n);
    for (i=0; i<n; i++)
    {
        printf("Enter element %d :'',i+1);
        scanf("%d", &a[i]);
    }
    printf("Unsorted list is:\n");
    for (i = 0; i < n; i++)
        printf("%d'', a[i]);
    printf("\n");
        /*Insertion sort*/
```

```
for(j=1; j<n; j++)
{
    k=a[j]; /*k is to be inserted at proper
    place*/
    for(i=j-1; i>=0 && k<a[i]; i--)
    a[i+1]=a[i];
    a[i+1]=k;
    printf("Pass %d, Element %d inserted in
    proper place\n", j,k);
for (i = 0; i < n; i++)
    printf("%d'', a[i]);
    printf("\n");
}
    printf("Sorted list is :\n");
    for (i = 0; i < n; i++)
        printf("%d '', a[i]);
    printf("\n");
    getch();
}   /*End of main()*/
```

Analysis: In insertion sort we insert the elements *i* before or after and we start comparison from the first element. Since the first element has no other elements before it, so it does not require any comparision. Second element requires 1 comparision, third element requires 2 comparisions, fourth element requires 3 comparisions and so on. The last element requires n – 1 comparisions. So the total number of comparisions will be–

$$1 + 2 + 3 + \ldots \ldots \ldots + (n - 2) + (n - 1)$$

It's a form of arithmetic progression series, so we can apply the formula sum = $\frac{n}{2}\{2a + (n - 1)d\}$.

Where, d = common difference i.e. first term – second term, a = first term in the series, n = total term.

Thus, sum = $\frac{(n-1)}{2}\{2 \times 1 + (n - 1) -1) \times 1\}$

$$= \frac{(n-1)}{2}\{2 + n - 2)$$

$$= n^2 \frac{(n-1)}{2}$$

Which is of $O(n^2)$.

It is the worst case behaviour of insertion sort where all the elements are in reverse order. If we compute the average case of the above algorithm then it will be of $O(n^2)$. The insertion sort technique is very efficient if the number of elements to be sorted are very less.

7.5 SELECTION SORT

Let us have a list containing *n* elements in unsorted order and we want to sort the list by applying the selection sort algorithm. In selection sort technique, the first element is compared with all the remaining (n – 1) elements. The smallest element is placed at the first location. Again, the second element is compared with the remaining (n – 2) elements and pick out the smallest element from the list and placed in the second location and so on until the largest element of the list.

Let A is an array with *n* elements A[0], A[1], ..., ..., A[N – 1]. First we will search the position of smallest element from A[0], ..., ..., ..., A[N – 1]. Then we will interchange that smallest element with A[0]. Now you will search the position of the second smallest element (because the A[0] is the first smallest element) from A[1], ..., ..., ..., A[N – 1], then interchange that smallest element with A[1]. Similarly the process will be for A[2], ..., ..., ..., A[N – 1]. The whole process will be as–

Pass 1: Search the smallest element from A[0], A[1], ..., ..., A[N – 1]
Interchange A[0] with the smallest element
Result: A[0] is sorted.

Pass 2: Search the smallest element from A[1], A[2], ..., ..., A[N – 1]
Interchange A[1] with the smallest element
Result: A[0], A[1] is sorted.

.....

.....

Pass N – 1: Search the smallest element from A[N – 2], A[N – 1]

Interchange A[N -2] with the smallest element

Result: A[0], A[1], ..., ..., A[N – 2], A[N – 1] is sorted.

Thus A is sorted after N – 1 passes.

Let us take an example of the following elements:

Pass 1	**75**	35	42	13	87	24	64	57
Pass 2	13	**35**	42	75	87	**24**	64	57
Pass 3	13	24	**42**	75	87	**35**	64	57
Pass 4	13	24	35	**75**	87	**42**	64	57
Pass 5	13	24	35	42	**87**	75	64	57
Pass 6	13	24	35	42	57	**75**	**64**	87
Pass 7	13	24	35	42	57	64	**75**	87
Sorted elements	13	24	35	42	57	64	**75**	87

Program 7.2: Program of sorting using selection sort

```
#include<stdio.h>
#include<conio.h>

void main()
    {
        int a[25], i, j, k, n, temp, smallest;
        clrscr();
        printf("Enter the number of elements :'');
        scanf("%d", &n);
        for (i = 0; i < n; i++)
        {
            printf("Enter element %d:'', i+1);
            scanf("%d", &a[i]);
        }
            /*Display the unsorted list*/
            printf("Unsorted list is : \n");
            for (i = 0; i < n; i++)
                printf("%d '', a[i]);
            printf("\n");
```

```
            /*Selection sort*/
        for(i = 0; i < n-1 ; i++)
        {
            /*Find the smallest element*/
                smallest = i;
            for(k = i+1; k < n ; k++)
            {
                if(a[smallest] > a[k])
                    smallest = k ;
            }
            if(i ! = smallest)
            {
                temp = a [i];
                a[i] = a[smallest];
                arr[smallest] = temp ;
            }
            printf(''After Pass %d elements are
            : '', i+1);
            for (j = 0; j < n; j++)
                printf(''%d '', a[j]);
                printf(''\n'');
        } /*End of for*/
        printf("Sorted list is : \n");
        for (i = 0; i < n; i++)
            printf(''%d '', a[i]);
        printf(''\n'');
    getch();
} /*End of main()*/
```

Analysis: As we have seen selection sort algorithm will search the smallest element in the array and then that element will be at proper position. So in Pass 1 it will compare n – 1 elements, in Pass 2 comparison will be n – 2 because the first element is already at proper position. Thus we can write the function for comparison as:

$$F(n) = (n-1) + (n-2) + (n-3) + \ldots \ldots \ldots + 3 + 2 + 1$$

This is an arithmetic series, solving the series we will get,

$$F(n) = \frac{(n-1)}{2}[2(n-1) + \{(n-1)-1\}\{(n-2)-(n-1)\}]$$

$$= \frac{(n-1)}{2}[2n - 2 + (n-1-1)(n-2-n+1)]$$

$$= \frac{(n-1)}{2}[2n - 2 + (n-2)(-1)]$$

$$= \frac{(n-1)}{2}[2n - 2 - n + 2]$$

$$= \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Thus, the number of comparisons is proportional to (n^2). It is the worst case behaviour of selection sort. If we compute the average case of the above algorithm then it will be of $O(n^2)$. The best thing with the selection sort is that in every pass one element will be at correct position, very less temporary variables will be required for interchanging the elements and it is simple to implement.

7.6 BUBBLE SORT

Bubble sort is a commonly used sorting algorithm. In this algorithm, two successive elements are compared and interchanging is done if the first element is grearter than the second one. The elements are sorted in ascending order.

Let A is an array with *n* elements A[0], A[1],, ..., ..., ..., A[N – 1]. The bubble sort algorithm works as follows:

Step 1: First compare A[0] and A[1] and arrange them so that A[0]<A[1]. Then compare A[1] and A[2] and arrange them so that A[1]<A[2]. Then compare A[2] and A[3] and arrange them so that A[2]<A[3]. Continue until we compare A[N – 2] and A[N – 1] and arrange them int the desired order so that A[N – 2]<A[N – 1].



The process of sequentially traversing through all or part of a list is frequently called a *pass*, so each of the above steps is called a *pass*.

Observe that Step 1 involves *n – 1* comparisions. During the Step 1, the largest element is "bubbled up" to the *n*th position or "sinks" to the *n*th position. When Step 1 is completed, A[N – 1] will contain the largest element.

Step 2: Repeat Step 1 and finally the second largest element will occupy A[N -2]. In this step there will be *n – 2* comparisons.

Step 3: Repeat Step 1 and finally the third largest element will occupy A[N -3]. In this step there will be *n – 3* comparisons.

.....

.....

Step N–1: Compare A[1] and A[2] and arrange them so that A[1]<A[2]

After *n – 1* steps, the list will be sorted in ascending order.

Let us take an example of the following elements:

13 32 20 62 68 52 38 46

We will apply the bubble sort algorithm to sort the elements.

Pass 1: We have the following comparisions:

a) Compare A0 and A1, 13<32, no change

b) Compare A1 and A2, 32>20, interchange

13 20 32 62 68 52 38 46

c) Compare A2 and A3, A2<A3, no change

d) Compare A3 and A4, A3<A4, no change

e) Compare A4 and A5, A4>A5, interchange

13 20 32 62 52 68 38 46

f) Compare A5 and A6, A5>A6, interchange

13 20 32 62 52 38 68 46

g) Compare A6 and A7, A6>A7, interchange

13 20 32 62 52 38 46 68

Pass 2: 13 20 32 62 52 38 46 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

c) Compare A2 and A3, 32<62, no change

d) Compare A3 and A4, 62>52, interchange

13 20 32 52 62 38 46 68

e) Compare A4 and A5, 62>38, interchange

13 20 32 52 38 62 46 68

f) Compare A5 and A6, 46>62, interchange

13 20 32 52 38 46 62 68

At the end of the Pass 2 the second largest element 62 has moved to its proper place.

Pass 3: 13 20 32 52 38 46 62 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

c) Compare A2 and A3, 32<52, no change

d) Compare A3 and A4, 52>38, interchange

13 20 32 38 52 46 62 68

e) Compare A4 and A5, 52>46, interchange

13 20 32 38 46 52 62 68

Pass 4: 13 20 32 38 46 52 62 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

c) Compare A2 and A3, 32<38, no change

d) Compare A3 and A4, 38<46, no change

Pass 5: 13 20 32 38 46 52 62 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

c) Compare A2 and A3, 32<38, no change

Pass 6: 13 20 32 38 46 52 62 68

a) Compare A0 and A1, 13<20, no change

b) Compare A1 and A2, 20<32, no change

Pass 7: 13 20 32 38 46 52 62 68

a) Compare A0 and A1, 13<20, no change

Since the list has 8 elements, it is sorted after the 7th Pass. The list was actually sorted after the 4th Pass.

Program 7.3: Program of sorting using bubble sort

```
#include<stdio.h>
#include<conio.h>

void main()
    {
        int a[25], i, j, k, temp, n, xchanges;
        clrscr();
        printf(''Enter the number of elements :'');
        scanf(''%d'',&n);
        for (i = 0; i < n; i++)
        {
            printf(''Enter element%d :'', i+1);
            scanf(''%d'',&a[i]);
        }
        printf(''Unsorted list is:\n'');
        for (i = 0; i < n ; i++)
            printf(''%d '', a[i]);
        printf(''\n'');

        /* Bubble sort*/
        for (i = 0; i < n-1 ; i++)
        {
            xchanges=0;
        /* Bubble sort*/
        for (i = 0; i < n-1 ; i++)
        {
            xchanges=0;
            for (j = 0; j <n-1-i; j++)
            {
                if (a[j] > a[j+1])
                {
                    temp = a[j];
```

```
                    a[j] = a[j+1];
                    a[j+1] = temp;
                    xchanges++;
                } /*End of if*/
            } /*End of inner for loop*/
            if(xchanges==0)  /*If list is sorted*/
                break;
            printf(''After Pass %d elements are :
            '', i+1);
            for (k = 0; k < n; k++)
                printf(''%d '', a[k]);
            printf(''\n'');
        } /*End of outer for loop*/

        printf(''Sorted list is :\n'');
        for (i = 0; i < n; i++)
            printf(''%d '', a[i]);
        getch();
        printf(''\n'');
    } /*End of main()*/
```

Analysis: As we have seen bubble sort algorithm will search the largest element in the array and placed it at proper position in each Pass. So in Pass 1 it will compare n – 1 elements, in Pass 2 comparison will be n – 2 because the first element is already at proper position. Thus, we can write the function for comparison as

$F(n) = (n – 1) + (n – 2) + (n – 3) + ... + 3 + 2 + 1$

This is an arithmetic series, solving the series we will get,

$$F(n) = \frac{n}{2}\{2a + (n – 1)d\}$$

$$= \frac{(n-1)}{2}\{2 \times 1 + (n – 1) – 1) \times 1\}$$

$$= \frac{(n-1)}{2}\{2 + n – 2)$$

$$= \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Thus, the time required to execute the bubble sort algorithm is proportional to (n^2), where *n* is the number of input items.

7.7 QUICK SORT

This is the most widely used sorting algorithm, invented by C.A.R. Hoare in 1960. This algorithm is based on partition. Hence it falls under the divide and conquer technique. In this algorithm, the main list of elements is divided into two sub-lists. For example, a list of *n* elements are to be sorted. The quick sort marks an element in the list called **pivot** or **key**. Consider the first element *P* as a pivot. Shift all the elements whose value is less than *P* towards the left and elements whose value is greater than *P* to the right of *P*. Now, the pivot element divides the main list into two parts. It is not necessary that the selected key element must be in the middle. Any element from the list can act as key or pivot element.

Now, the process for sorting the elements through quick sort is as:

1) Take the first element of list as pivot.
2) Place pivot at the proper place in list. So one element of the list i.e., pivot will be at its proper place.
3) Create two sublists left and right side of pivot.
4) Repeat the same process untill all elements of the list are at proper position in the list.

For placing the pivot element at proper place we have to do the following process:

1) Compare the pivot element one by one from right to left for getting the element which has value less than the pivot element.
2) Interchange the element with pivot element.
3) Now the comparision will start from the interchanged element position from left to right for getting the element which has higher value than pivot.
4) Repeat the same process untill pivot is at its proper position.

Let us take a list of element and assume that 48 is the pivot element.

48 29 8 59 72 88 42 65 95 19 82 68

We have to start comparision from right to left. Now the first element less than 48 is 19. So interchange it with pivot i.e., 48.

19 29 8 59 72 88 42 65 95 48 82 68

Now the comparision will start from 19 and will be from left to right. The first element greater than 48 is 59. So interchange it with pivot.

19 29 8 48 72 88 42 65 95 59 82 68

Now the comparision will start from 59 and will be from right to left. The first element less than 48 is 42. So interchange it with pivot.

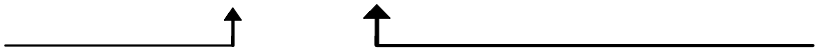
19 29 8 42 72 88 48 65 95 59 82 68

Now the comparision will start from 42 and will be from left to right. The first element greater than 48 is 72. So interchange it with pivot.

19 29 8 42 48 88 72 65 95 59 82 68

Now the comparision will start from 72 and will be from right to left. There is no element less than 48. So, now 48 is at its proper position in the list. So we can divide the list into two sublists, left and right side of pivot.

19 29 8 42 48 88 72 65 95 59 82 68



Sublist 1 Sublist 2

Now, the same procedure will be followed for the sublist 1 and sublist 2 and finally you will get the sorted list.

Program 7.4: Program to demonstrate the quick sort allgorithm

```
#include<stdio.h>
#include<conio.h>

enum bool { FALSE,TRUE };
void main()
    {
        int elem[20],n,i;
        clrscr();
        printf(''Enter the number of elements :'');
```

```
            scanf(''%d'',&n);
            for(i=0; i<n; i++)
            {
                printf(''Enter element %d : '',i+1);
                scanf(''%d'',&elem[i]);
            }

            printf(''Unsorted list is :\n'');
            display(elem,0,n-1);
            printf(''\n'');

            quick(elem,0,n-1);

            printf(''Sorted list is :\n'');
            show(elem,0,n-1);
            getch();
            printf(''\n'');

        }/*End of main()*/

    quick(int a[ ],int low, int up)
        {
            int piv, temp, left, right;
            enum bool pivot_placed=FALSE;
            left=low;
            right=up;
            piv=low;    /*Take the first element of
            sublist as piv */

            if(low>=up)
                return;
            printf(''Sublist : '');
            show(a,low,up);
```

```
/*Loop till pivot is placed at proper place in
the sublist*/

while(pivot_placed==FALSE)
    {
        /*Compare from right to left */
    while( a[piv]<=a[right] && piv!=right )
            right=right-1;
        if( piv==right )
            pivot_placed=TRUE;
/* Bubble sort*/
for (i = 0; i < n-1 ; i++)
{
    xchanges=0;
if( a[piv] > a[right] )
    {
        temp=a[piv];
        a[piv]=a[right];
        a[right]=temp;
        piv=right;
    }
/*Compare from left to right */
    while( a[piv]>=a[left] && left!=piv )
        left=left+1;
    if(piv==left)
        pivot_placed=TRUE;
    if(a[piv] < a[left])
    {
        temp=a[piv];
        a[piv]=a[left];
        a[left]=temp;
        piv=left;
    }
} /*End of while*/
```

```
            printf(``-> Pivot Placed is%d -> '', a[piv]);
            show(a,low,up);
            printf(``\n'');

                quick(a, low, piv-1);
                quick(a, piv+1, up);
        }/*End of quick()*/

    show(int a[ ], int low, int up)
    {
        int i;
        for(i=low; i<=up; i++)
            printf(``%d '',a[i]);
    }
```

Analysis: The time required by the quick sorting method (i.e., the efficiency of the algorithm) depends on the selection of the pivot element. Suppose the pivot element is chosen in the middle position of the list so that it divides the list into two sublists of equal size. Now, repeatedly applying the quick sort algoritm on both the sublists we will finally have the sorted list of the elements.

Now, after 1^{st} step total elements in correct position is = $1 = 2^1 - 1$

after 2^{nd} step total elements in correct position is = $3 = 2^2 - 1$

Now, after 1^{st} step total elements in correct position is = $1 = 2^1 - 1$

after 3^{rd} step total elements in correct position is = $7 = 2^3 - 1$

.....

.....

after l^{th} step total elements in correct position is = $2^l - 1$

Therefore, $2^l - 1 = n - 1$

or $2^l = n$

or $l = \log_2 n$

Here, the value of *l* is the number of steps. If *n* is comparision per step then for $\log_2 n$ steps we get = $n\log_2 n$ comparisions. Thus, **the overall time complexity of quick sort is = $n\log_2 n$.**

The above calculated complexity is called the **average case complexity**. *So, the condition for getting the average case complexity is choosing the pivot element at the middle of the list*.

Now, let us suppose we have one condition that the given list of the elements is initially sorted. We consider the first element of the list as the pivot element. In this case, the number of steps needed for obtaining the finally sorted list is = (n – 1).

Again, the number of comparisions in each step will be almost *n* i.e., it will be of *O(n).*

So, the total number of comparisions for (n – 1)steps is = (n – 1)O(n)

$= O(n^2)$

Thus, the time complexity in this case will be = $O(n^2)$. This is called the **worst case complexity** of the quick sort algorithm. *So, the condition for worst case is if the list is initially sorted*.



CHECK YOUR PROGRESS

Q.1: Selection sort and quick sort both fall into the same category of sorting algorithms. What is this category?

A) O(n log n) sorts B) Divide-and-conquer sorts

C) Interchange sorts D) Average time is quadratic.

Q.2: What is the worst-case time for quick sort to sort an array of n elements?

A) O(logn) B) O(n)

C) O(nlogn) D) $O(n^2)$

Q.3) When is insertion sort a good choice for sorting an array?

A) Each component of the array requires a large amount of memory.

B) Each component of the array requires a small amount of memory.

C) The array has only a few items out of place.

D) The processor speed is fast.

Q.4: Give the time complexities of Insertion Sort, Selection Sort, Bubble Sort and Quick Sort.



7.8 LET US SUM UP

- Sorting is a method of arranging data in ascending or descending order.
- The Insertion Sort is so named because on each iteration of its main loop it inserts the next element in its correct position relative to the subarray that has already been processed.
- The complexity of Insertion Sort is $O(n^2)$.
- The complexity of Selection Sort is $O(n^2)$.
- The Bubble Sort name comes from the idea that the larger elements 'bubble up' to the top (the high end) of the array like the bubbles in waters.
- The Bubble Sort is slower than the Selection Sort, and the Insertion Sort (in most cases) is a little faster.
- The complexity of Bubble Sort is $O(n^2)$.
- The Quick Sort works by partitioning the array into two pieces separated by a single element that is greater than all the elements in the left piece and smaller than all the elements in the right piece.
- The complexity of Quick Sort is $O(nlogn)$.



7.9 FURTHER READING

1) Kamthane, A. N. (2004). *Introduction to Data Structures in C*. Pearson Education India.
2) Langsam, Y., Augenstein, M. & Tenenbaum, A. M. (1996). *Data Structures using C and C++* (Vol. 2). New Jersey: Prentice Hall.
3) Seymour, L. (2010). *Data Structures with C*. Tata McGraw Hill.



7.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: C

Ans. to Q. No. 2: D

Ans. to Q. No. 3: C

Ans. to Q. No. 4: The complexity of Insertion Sort is $O(n^2)$. The complexity of Selection Sort is $O(n^2)$. The complexity of Bubble Sort is $O(n^2)$. The complexity of Quick Sort is $O(nlogn)$.



7.11 MODEL QUESTIONS

Q.1: Explain the different types of sorting memthods.

Q.2: What is sorting? Write a function for Bubble Sort.

Q.3: Write a program for sorting a given list by using Insertion Sort:

2, 32, 45, 67, 89, 4, 3, 8, 10

Q.4: Write a program to sort the following members in ascending order using Selection Sort:

12, 34, 5, 78, 4, 56, 10, 23, 1, 45, 65

Q.5: Write a program to sort the following list using Quick Sort method:

4, 3, 1, 6, 7, 2, 5, 8

Q.6: Show all the passes using the Quick Sorting technique with the following list–

a) 26, 5, 37, 1, 61, 11, 15, 48, 19

b) 4, 3, 1, 6, 7, 2, 5, 8

Q.7: Show all the passes using the Bubble Sorting technique with the following list–

234, 54, 12, 76, 11, 87, 32, 12, 45, 67, 76

Q.8: Show all the passes using the Selection Sorting technique with the following list–

12, 34, 5, 78, 4, 56, 10, 23, 1, 45, 65

Q.9: Show all the passes using the Insertion Sorting technique with the following list–

13, 33, 27, 77, 12, 43, 10, 432, 112, 90

Q.10: Compare the performance of quick sort and selection sort.

Q.11: How does a quick sort performs on:

a) An array that is already sorted

b) An array that is sorted in reversed order

Q.12: The Bubble sort, the Selection sort, and the Insertion sort are all $O(n^2)$ algorithms. Which is the fastest and which is the slowest among them?

*** ***** ***

Name of the Paper: .....
Course Code: .....

Centre for Internal Quality Assurance (CIQA)
Krishna Kanta Handiqui State Open University
City Office: Housefed Complex, Guwahati-781006

Learner's Feedback on Course

Dear Learner,

Regarding the course as mentioned above, we would like to know your opinions and comments so as to improve the quality of self learning materials in future. Please respond to the following statements by ticking the number you feel most reflect your opinion. After completion of the additional comments, please detach the page and send/mail the same to us at the address given below.

The Director, Centre for Internal Quality Assurance, KKHandiqui State Open University
Housefed Complex, Dispur, Guwahati-781006
(E-mail id: ciqa@kkhsou.in)

1) Approximately how many hours did you spend for studying the units in the course? ☐☐☐

2) Please give your opinions (by ✓ mark) to the following items based on your reading of the block:

Sl. No.	Statements	Strongly Agree	Agree	Neither Agree nor Disagree	Disagree	Strongly Disagree
I)	The SLMs of the course fulfil the learning objectives set out in the beginning of each unit					
II)	The units of the SLM could satisfy my academic needs and expectations					
III)	The Unit writers have excellent knowledge about the course contents					
IV)	Language and contents of the units were presented at a level which I could readily understand					
V)	Ample opportunity for participation in the activities provided in the units					
VI)	Used enough Illustrations (Diagrams, tables etc.) for conceptual clarity					
VII)	Quality of content is engaging, relevant, and up-to-date					
VIII)	The self check questions are very helpful					
IX)	The Possible/model questions and the answers to check my progress have benefited me a lot					

Additional Comments: *(Please feel free to provide your open comments)*

1) Which aspects of the SLM, according to you, worked well?

.....

.....

2) What sort of changes/improvements do you feel KKHSOU could implement to improve the overall quality of the SLM?

.....

.....

Thank you for taking the time to complete this form.