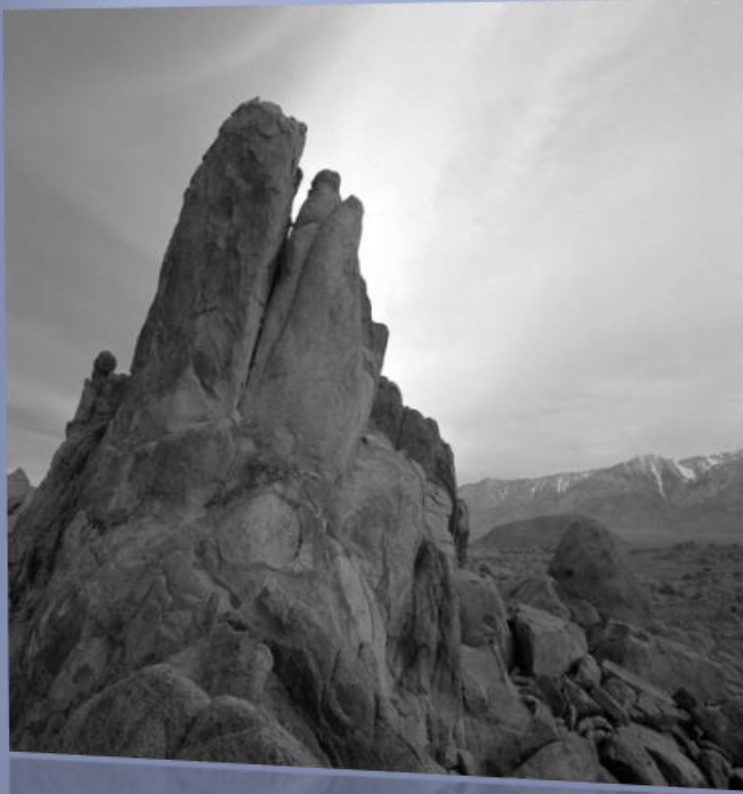


Dr. MBANZABUGABO Jean Baptiste
(BE, MCA, MSc.SE, PhD.)



DATA STRUCTURES AND ALGORITHMS- STUDENT VERSION

*UNIVERSITY OF TOURISM TECHNOLOGY AND BUSINESS STUDIES(UTB) –
DEPARTMENT OF BUSINESS INFORMATION TECHNOLOGY*

**“Just-in-time-erudition, learning by case in point,
and Learning by doing and Demonstrating.”**

PREFACE

An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed.

The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs

“The data structures deal with the study of how the data is organized in the memory, how efficiently it can be retrieved and manipulated and the possible ways in which different data items are logically related”.

“Algorithms play the central role in both the science and practice of computing as utter finite sequence of steps for accomplishing some computational task”.

Data Structure and Algorithms using C book is designed for the use in Undergraduate level, Trimester or Semester course intended to provide Computer Science, Computer Applications, Computing Information System, Information Technology and Business Information Technology Students with a strong foundation in Data Structures and Algorithms, in the logical and mathematic model & Approaches, organasition, presentations and Techniques used in storing, accessing, manipulating and retrieving data within computer Memory.

MOTIVATION

This book was written to address three problem I see in teaching and course contents delivery:

1. Failure to incorporate applications and practical implementation of concepts
2. Omission or inadequate coverage of subject matter and themes
3. Excessive breadth with insufficient depth of topics coverage

INSTRUCTIONAL APPROACH

The philosophies of this book are just-in-time-erudition, learning by case in point, and Learning by doing and Demonstrating.

Just-in-time-erudition means that students learn a concept,theory and/or technique at the time they need to use it. Thererefore, for this book at each part, it includes exercises to help student assess their understanding after reading and practicing about the concepts and techniques.

Learning by case in point means that student are provided numerous examples and illustrations to facilitate learning.

I do bleave that students do not really comprehend a concept until they have successfully applied it.

PREREQUISITES

The book assumes that a reader has gone through an introductory programming course with C Compiler and a standard course on fundamentals of Mathematics and/or discrete Mathematics. With such a background, he or she should be able to handle the book's material without undue difficulty.

LEARNING OUTCOMES

At the end of the subject, it is expected that students will be able to:

- Able to choose proper data type and algorithm to use in application to save memory.
- Able to analyze Algorithms to structure the execution flow of a program
- Able to Create and define data types and their abstraction.
- Differentiating between ordered lists and sorted lists.
- Explaining organization and manipulation of Linear and Non Linear Data Structures.
- To develop an algorithms for real world problem.
- Able to find the system/algorithms efficiency and complexity.

ACKNOWLEDGMENTS

The author acknowledge with thanks all the individuals and curriculum developers whose contributions were vital in the preparation of this book.

Particular acknowledgement is given to the following four contributor:

University of Tourism Technology, Business Studies(UTB) and University of Kigali(UoK),and University of Gitwe(UoG). for providing the description of their highly regarded Bachelor curriculums of Computer Science,Information Technology, Computer Engineering, Computing Information Systems, Computer Applications and Business Information Technology.

RUKUNDO Prince for writing the chapter on Tree based Data Structures.

Dr. Edward Lambert who are well known with the generous help and support contributing the description and procedures.

Students and Lecturers from University of Tourism Technology and Business Studies for providing the valuable suggestions for improving the quality of the book.

BOOK STRUCTURE

The book Introducing the student to Data Structure and Algorithms and it is presented in seven Parts.

The book is designed in such a way that the reader understand from the basics to advanced topics

Part one provides an Overview and fundamental concepts, classifications of Data Structures, Data structures' operations and Alogorithm design.

Part two expresses foundation of memory allocation, Arrays,

Also covers the range of searching and Sorting Techniques and their implimentations

Part three Representation of linked lists based data structure and their implementation using arrays

Part four identifies representational model of stack based data structures and their applications

Part five identifies representational model of queue based data structures and their applications

Part six covers the data structure implementing ADT that simulates a hierarchical tree structure(Trees and Graphs)

Utmost care has been taken in bringing out this book, I hope this book will serve the requirements of all readers. It may happen that in the spite of being watchful, some errors might have crept in. I would be grateful if this is brought out as valuable suggestions for improving the quality of the book for the next coming version.

ABOUT AUTHOR

A highly organized IT professional who has a proven track record and innovative at every moment. With more than 9 years of progressively responsible and professional experience, has offered him a distinguished career earmarked by accomplishments in leading and directing Information Technology operations across broad disciplines, including Networking, hardware, software requirement specifications, MIS software development and testing, IT support, Software project management, Database technologies, promoting change and perfection of IT in Education for Teaching, Learning and Assessment, and technology implementation. His proven track record includes monitoring and planning implementation of new technology projects and evolving technology strategies that align with business needs while ensuring that services meet satisfactions and organization's goals.

His greatest strengths are being able to mould as the situations demand, patience, sincere, committed to hard work, self-learning, research and motivated to innovate.

Having completed his Bachelor of Engineering from the former Kigali Instituted of Science and Technology(KIST), Masters of Computer Applications from Bangalore University and Annamalai University for Masters of Science in Software Engineering, Doctor of Science in computer Science at Atlantic International University(AIU).

Dr. Mbanzabugabo Jean Baptiste has a passion for writing and becoming an author of several books contributing to Rwandan Education in Computer Science, Computing and Information Technology. His current affiliations include being Dean of the Faculty of Applied Science and Technology at University of Tourism Technology and Business Studies(UTB), a Lecturer of IT related courses in various Rwandan Universities and colleges. He can be reached at i_engineer@netzero.com/dean.it@utb.ac.rw, +250788445598.

PART 1

AN OVERVIEW AND GENERAL INTRODUCTION TO DATA STRUCTURES

1.0 INTRODUCTION

2. The procedure of solving problems using a electronic computer system requires us to transform data from one form to another. In most cases large amounts of data is to be processed. The information that is available to the computer consists of selected set of data which is considered relevant and intelligible to the problem at hand or that set in which it is believed that the desired result can be derived and presented. The data represents an abstraction of reality. i.e. **Data** is a set of elementary items. And Information is the data that has been processed for use in decision making.



Figure 1.1 Role of Data and Information

In solving a problem with or without a computer it is necessary to choose abstraction of reality. i.e. to define a set of data which represents the real situation. This choice must be guided by the problem to be solved. Then we will have to identify a choice of representation and interpretation of information. This choice of representation must always be taken based on the operations that are to be performed on the data. There has always been an close connection between the representation of data and the development of algorithms.

Three components are to be identified with the types of data which are used to solve problems:

1. The representation of data.
2. The operations that can be performed.

3. The relationships between various objects

Sometimes data has to be organized into more complex structures so that data can be processed efficiently. The choice of the structure is such that it can be easily accessed, simple in representation and can be efficiently processed. Whenever data is structured one has to identify the representation of data in the computer's memory. This is called as the storage structure or memory representation.

Example of Memory blocks

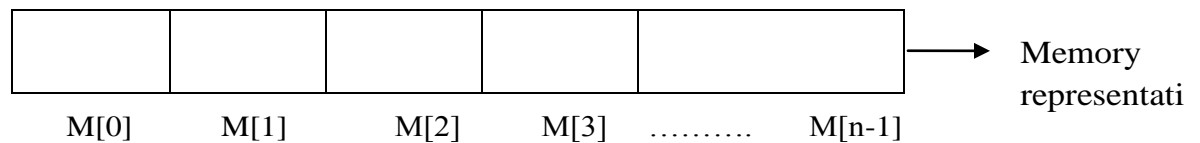


Figure 1.2: Storage representation of Data

To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the “big picture” without getting mislaid in the details by creating models of the problem domain, we are able to utilize a better and more efficient problem-solving process. These models allow us to describe the data that our algorithms will manipulate in a much more consistent way with respect to the problem itself.

1.1 DEFINITIONS AND NEEDS FOR DATA STRUCTURES:

Data is a set of elementary items.

Data Structure: Is a collection of data elements whose organization is characterized by accessing functions that are used to store and retrieve individual data elements.

There are two important need for data structures are first to identify the representation of abstract entities and then to identify the operations, which can be performed with them. The operations help us to determine the class of problem which can be solved with these entities.

Therefore, Data structure is nothing but arrangement of Data in the program, their relationship and the allowed operations used to access and retrieve data from to memory.

“The data structures deal with the study of how the data is organized in the memory, how efficiently it can be retrieved and manipulated and the possible ways in which different data items are logically related”.

Thus, The necessity of data structures can be designed to answer certain questions such as:

1. Does the data structure do what it is supposed to do?
2. Does the representation works according to the requirement specification of the task?
3. Is there a proper description of the representation describing how to use it and how it works?

The above questions when answered create the fundamental goals that are used in designing descriptions of data structures. some of them are:

1. Correctness
2. Efficiency and Robust
4. Adaptability
5. Reusability

By **correctness**, we mean that a data structure is designed to work correctly for all possible inputs that might be encountered. The precise meaning of correctness will always depend on the specific problem the data structure is intended to solve. But correctness should be a primary goal.

Useful data structure and their operations also need to be **efficient**. That is they should be fast and not use more of the computer's resources, such as memory space than required. In a real-time situation the speed of a data structure operation can make the difference between success and failure, a difference that can be quite important.

Every good programmer wants to produce software that is **robust**, which means that a program produces the correct output for all Inputs. For example if a program is expecting a number to be input as an Integer and instead it is input as a floating-point number, then the program should be able to recover from this error. A program that does not handle such unexpected-input errors can be thwarting for the programmer.

Modern software projects, such as those for developing Information Management Systems, Web Applications, and Internet search engines, involve large software systems that are expected to last for many years. Software, therefore, needs to be able to evolve over time in response to changing conditions. These changes can be expected, such as the need to adapt to an increase in CPU speed. Software should also be able to adapt to unexpected events. Thus, another important goal of quality software depend up on the data structure use to organize and store data is that to be **adaptable**.

Going hand-in-hand with adaptability is the prediction that software would be **reusable** that is the same code is a component of different systems in various application situations. Developing quality software can be expensive, and its cost can be reduced somewhat if the software is designed in a way that makes it easily reusable in future applications. Software reuse can be a significant cost-saving and time saving technique.

1.2 CLASSIFICATION OF DATA STRUCTURES

Data Structure as a systematic way of organizing and accessing data.

Data structures can be classified, keeping in view the way the elements are connected to each other, the way when they are stored in memory or the way further division of a data structure.

If the data contains a single value this can be organized using primitive data type or Simple Data Structure. If the data contains set of values they can be represented using non-primitive data types also called as Compound Data structure.

Simple data structure represent the standard data types of any one of the computer programming languages' Variables and pointers, whereas Compound Data structure are constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user.

The data structures are classified in the following Categories:

1. Primitive data structures
2. Non-primitive data structures.

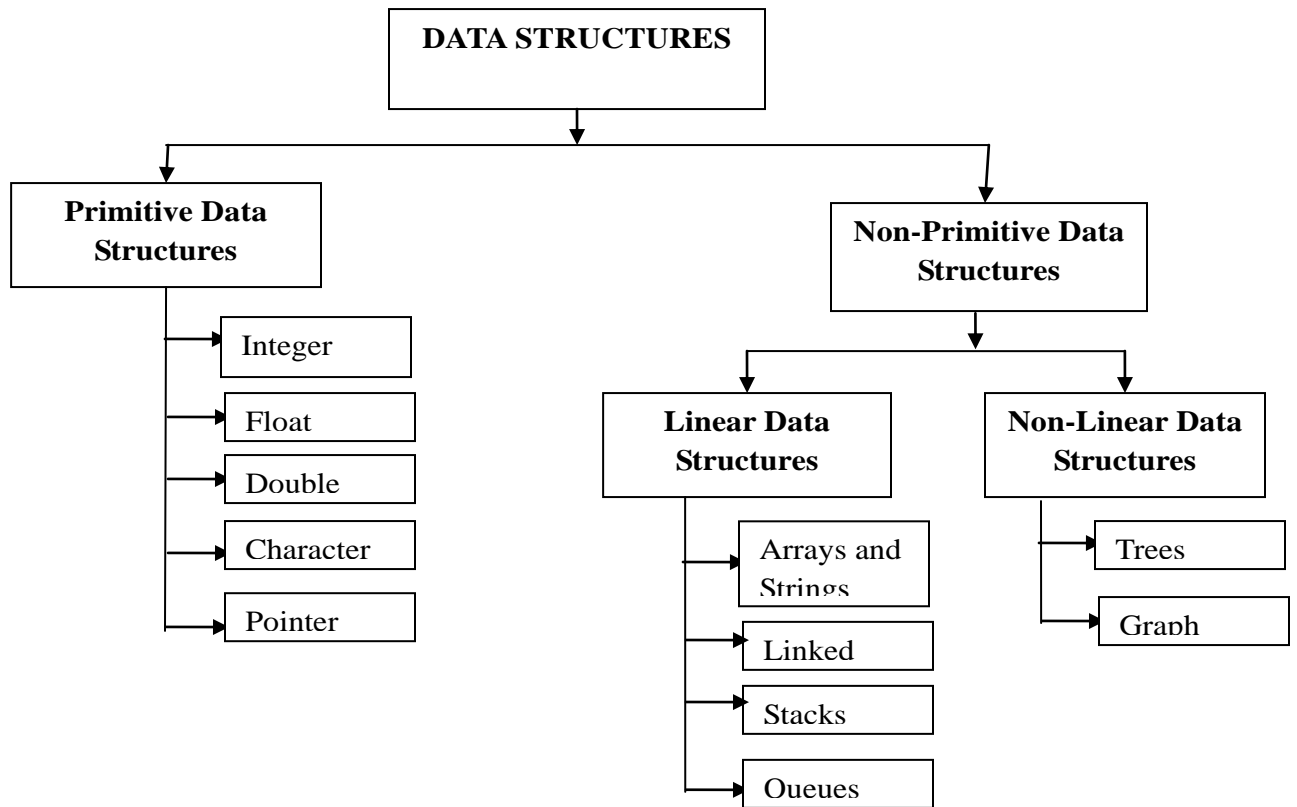


Figure 1.3 Classification of Data Structure

3.

1.2.1 Primitive data Structures

Primitive data structures are those structure which are readily available in a programming language i.e.. They can be directly operated upon by programming instructions. Here we are concerned with structuring of data at their most primitive level within a computer.

Definition

Primitive data structures: The primitive data types are the basic data types that are available in most of the programming languages. The primitive data types are used to represent single values.

1.2.1.1 Storage representation of Primitive data Structures

The storage representation(Memory representation) and the possible operations (arithmetic operations, relational operations) for these types of structures are

predefined and the user cannot change this. The storage structure of these data structures may vary from one machine to another. The different primitive data structures are Boolean, integer, float, double, character, and pointer.

Boolean This is used represent logical values either true or false.

When we **represent positive Integer** numbers, its nothing easier as eating ice cream. Just as your can remember the memory measurement follow binary principle ie times two- times two.

Example: 1024 512 256 128 64 32 16 8 4 2 1 then, the four bits representation of 13 is = 8+4+1

1024	512	256	128	64	32	16	8	4	2	1
							1	1	0	1

This means that we mark 8, 4 and 1 with 1 bit and remaing digit marked with 0, simulary can be applied all positive integer numbers.

When we **represent negative Integer numbers** using a binary system, the notation can either follow Signed Magnitude, one's complement or two's complement method, this is widely used way of representing a range of integers covering positive and negative values.

In the one's complement changing each bit in its absolute position to its opposite bit represent the negative number. Thus the pattern 01101 of a decimal number 13 Would become 10010. This represents -13. It means that the leftmost digit no longer represents the power *of 2* but is used to represent the sign of the number.

In the two's complement the negative number is represented by generating the one's complement and then adding the binary number 1. Thus the pattern 01101, which represents 13, would become 10011. This represents -13.

INTEGER	Signed Magnitude	ONE'S COMPLEMENT STORAGE REPRESENTATION	TWO'S COMPLEMENT STORAGE REPRESENTATION

13	00001101	00001101	00001101
-13	10001101	11110010	1111001010011

Figure 1.4: Integer Storage representation in number system with Signed Magnitude, One and Two's Complements

Floats and doubles are stored in mantissa and exponent form except that instead of the exponent representing the power of 10, it represents a power of 2, since base 2 is computer's natural format. The number of bytes used to represent a floating-point number depends on the precision of the variable.

Suppose we want to represent the decimal number 7.375 Its binary equivalent can be obtained as shown below:

Converting the floating-point number, The exponent is an integer stored in unsigned binary format after adding a positive integer bias. This ensures that the stored exponent is always positive. .

Characters

The use of computers is not to always work on numeric data we may have to sometimes use characters also. The ASCII representation of characters uses a 8 bit pattern for each character. If 8 bit patterns are used it is possible for us to represent 256 different possible characters with 256 different bit patterns.

Pointers

A pointer A pointer is a value that designates the address (i.e., the location in memory), of some value. Pointers are variables that hold a memory location to a data structure. Thus pointer gives us the address of the some other data locations. A pointer points to a memory location and through the use of proper operators (& for address, * for value), the object value at the address to which it points can be retrieved or written. For example, Figure 1.5 illustrates a pointer attached with some locations in memory. If one is interested in fetching the information, then one can easily obtain the information by using operator (&) with the pointer variable.


Base address		<table> <tr> <th>Address</th><th>Index</th><th>Information</th></tr> <tr> <td>0xff02</td><td>1</td><td>G</td></tr> <tr> <td>0xff03</td><td>2</td><td>H</td></tr> <tr> <td>0xff04</td><td>3</td><td>K</td></tr> <tr> <td>0xff05</td><td>4</td><td>J</td></tr> <tr> <td>.....</td><td>...</td><td>...</td></tr> <tr> <td>.....</td><td>...</td><td>...</td></tr> <tr> <td>0xff0a</td><td>9</td><td>R</td></tr> <tr> <td>0xff0b</td><td>10</td><td>E</td></tr> </table>	Address	Index	Information	0xff02	1	G	0xff03	2	H	0xff04	3	K	0xff05	4	J	0xff0a	9	R	0xff0b	10	E
Address	Index	Information																											
0xff02	1	G																											
0xff03	2	H																											
0xff04	3	K																											
0xff05	4	J																											
.....																											
.....																											
0xff0a	9	R																											
0xff0b	10	E																											

Figure1.2. Pointer and base address

Example 1.1 : Consider the following program segment in C.

```
int *ptr;

int Info;

Info=*ptr;
```

Where **ptr** contains the address 0xff02 and the information stored at this location is 11, after this assignment we will get 11 in **Info**.

1.2.1. 2. Operations on primitive Data Structures

Some of the common operations on primitive Data Structure are:

- i. **Creation Operation:** This operation is used to create a storage representation for a particular data structure. This operation is normally performed with the help of a declaration statement available in the programming language.

Example 1.2: `int n=45;`

This statement prompts memory space to be created for `n`.

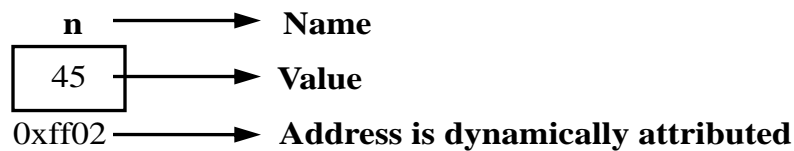


Figure 1.6 Memory space referred by, name `n` where integer value can be stored.

- ii. **Selection Operation:** This is most frequently used operation to access data within and from a data structure. For compound structures method of access is one of the important properties of a structure. In case of files the access can be sequential or random depending on the nature of files. This operation is normally performed using the name of the structure and format specified as well as address operator.

Example 1.3: `scanf ("%d", &a); printf ("%d",a);`

- iii. **Update Operation:** This operation is used to change or modify the data in a structure. An assignment operation is a good example of an update operation.

Example 1.4: `y=5;`

This above statement modifies the value of `y` to store the new value 5 in it.

- iv. **Destroy Operation:** This operation is used to destroy or disassociate a particular data structure from its storage representation. In some programming languages this operation is not supported and/or it is

automatically performed. In C one can destroy data structure by using the function called free(). This aids in efficient use of memory.

1.2.2. Non-Primitive Data Structures

Non-Primitive Data Structures are those structures, which are not readily available in a programming language i.e., they cannot be directly operated upon by programming instructions. The storage representation and the possible operations for these types of structures are not predefined and the user has to define them. The different types of non-primitive data structures supported in C programming are **arrays, strings, Unions, linked lists, stacks and queues**

Definition
Non-primitive data structures: The data structures that are composed of primitive data structures. These datatypes are used to store group of values.

Non-primitive data structures are further classified into two types.

1. Linear data structure and
2. Non-linear data structure

1.2.2.1. Linear Data Structures

A linear data structures demonstrate an important property called as adjacency and sequence between the elements. The concept of adjacency may signify either a linear or sequential relationship i.e., if we are able to identify the position of an element, we should be able to identify the position of the previous element and the next element. In Within this book we will look into details **arrays and string, linked list, stacks, queues** linear based data structures their operation as well as various algorithm that implement those operations.

Definition
Linear data structures: A data structure is said to be linear if and only if there is a adjacency relationship between the elements.

1.2.2.2. Non-Linear Data Structures

A non-linear data structures can exhibit any property other than adjacency between the elements. Non-linear data structure may exhibit either a hierarchical relationship or a parent child relationship. The different non-linear data structures are **trees** and **graphs**.

Definition
Non-Linear data structures: A data structure in which insertion and deletion is not possible in a linear fashion and Elements are stored based on the hierarchical relationship among the data.

1.2.2.3. OPERATIONS ON NON-PRIMITIVE DATA STRUCTURES

Some of the common operations on Non-Primitive Data Structure are:

- i. **Traversing:** It is the process of visiting each element in the data structure exactly once to perform certain operations on it. ie. Data Structure can be for example visited compare elements and/or to compute the sum, average, product of its data element
- ii. **Sorting:** It is the process of arranging the elements of a particular data structure in some form of logical order. The order may be either ascending or descending or alphabetic order depending on the data items present.
- iii. **Merging:** It is the process of combining the elements in two or more structures into a single structure.
- iv. **Searching:** It is the process of finding the location of the element with a given key value in a particular data structure or finding the location of an element, which satisfies the given condition.
- v. **Insertion:** It is the process of adding a new element to the structure. Most of the times this operation is performed by identifying the position where the new element is to be inserted.
- vi. **Deletion:** It is the process of removing an item from the structure.

1.3. ALGORITHM

The field of Computer Science can be defined as the study of algorithms because the main use of computers is to solve problems for us. A good algorithm is like a sharp knife, which does exactly what it is suppose to do with a minimum applied effort.

Definition
ALGORITHM: A set of ordered steps or procedures necessary to solve a problem. But general agreement about what the concept means, An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

The reference to "instructions" in the definition implies that there is something or someone capable of understanding and following the instructions given.

The nonambiguity requirement for each step of an algorithm cannot be compromised.

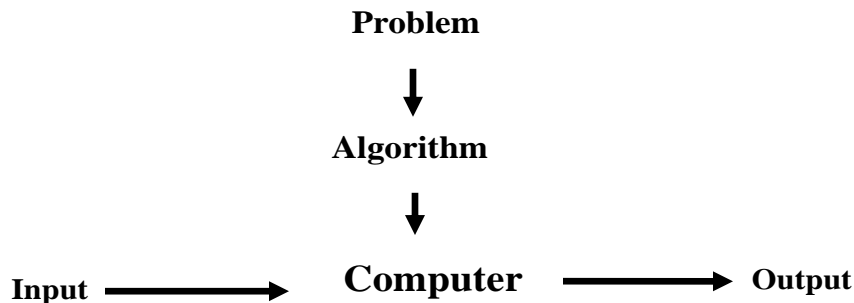


Figure 1.3: Notion of algorithm

The range of inputs for which an algorithm works has to be specified carefully.

- The same algorithm can be represented in several different ways.
- Several algorithms for solving the same problem may exist.
- Algorithms for the same problem can be based on very different ideas and
- can solve the problem with dramatically different speeds.

Every problem as we understand can be solved using different methods or techniques.

Thus each method may be represented using an algorithm. The important question to answer is How to choose the best algorithm? The design of a solution requires two goals to be looked at, most of the times they are conflicting. They are:

1. Design an algorithm that is easy to understand, convert into programming level and debug.
2. Design an algorithm that makes use of computer resources efficiently.

The first goal is concerned with a study in computer science called as Software Engineering and the second is concerned with the choice of data structures and the analysis of algorithms. Let us restrict our study to the second goal.

The main aim of using a computer is to transform data from one form to another. The algorithm describes the process of transforming data. That is why we often hear that a computer is referred to as a "Data Processing Machine". Raw data is the input to a computer and the algorithm shows the method used to transform the raw data into refined data or information. The organization of data thus plays an important role in the efficiency of algorithms since any organized data can be easily accessed and processed.

We have to organize the data either as a logical model or as a mathematical model as described by the definition of data structures.

The chosen model should reflect all the relationships and properties that exist between data and they can be accessed and implemented easily. While choosing a model we should also see to that as to what kind of operations can be performed on the data. Therefore Data Structures can also be defined as arrangement of data and their relationship.

Efficiency of algorithms depends upon the data structures that are selected for data representation. The data structure has to be finally represented in the memory. This is called as memory representation of data structures. While selecting the memory representation of data structures it should worth memory space and it should also be easy to access.

1.3.1 COMPLEXITY OF ALGORITHM

Every algorithm we write should be analyzed before it is implemented as a program. The process of analyzing algorithms forms a major field of study in computer science. There are two main criteria's or reasons upon which we can judge an algorithm. They are:

1. The correctness of the algorithm and
2. The simplicity of the algorithm

The correctness of an algorithm can be analyzed by tracing the algorithm with certain sample data and by trying to answer certain questions such as.

1. Does the algorithm do what we want it to do?
2. Does the algorithm work when the data structure used is empty?
3. Does the algorithm work when the data structure used is full?
4. Does the algorithm work for all possibilities that can occur between a full structure and an empty structure?

The simplicity of an algorithm can be analyzed by trying to understand whether the algorithm can be implemented in a better and much simpler way. In order to analyze this we will have to consider the time requirements and the space requirements of the algorithm. These are the two parameters on which the efficiency of the algorithms is measured. Space requirements are not a major problem today because memory is very cheap. So time is the only criteria for measuring efficiency of the algorithm as we have to maximize the utilization of the CPU and response time to be minimized to be faster. Consider the scenario given below:

The time complexity of an algorithm is not measured by finding out how much time a particular algorithm requires for performing its task because the speed of different computers may be different (a slower computer may take more time whereas a faster computer may take less time for the same algorithm). The time is measured by counting the number of key operations in the regards to line of codes, which are performed. Normally input and output operations are not considered as key operations. Comparison or assignment operations are considered as key operations. That is because key

operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operation.

Statement	Complexity
<code>x=x+1;</code>	We assume that the statement <code>x=x+1</code> is not contained within any loop either explicitly or implicitly. Then, its frequency count is 1.
<code>for (i=1; i<=n; i++)</code> <code>for (j=1; j<=n; j++)</code> <code>x=x+1;</code>	This will be executed n^2 times.

Figure 1.4: Evaluation of the complexity of algorithm

The complexity of an algorithm is the function of $f(n)$, which gives running time of the algorithm in terms of the number of key operations performed.

Let us take an example to understand how the analysis of time complexity helps us to decide the amount of work done by the algorithm based on which it is possible to select the best algorithm.

Example 1.5: Consider the task of finding the largest of three numbers. This problem can be solved using many methods, let us look at some of the methods and then analyze them.

Method 1: Step 1: `L=num1`

Step 2: `if (num2>L) L=num2`

Step 3: `if (num3>L) L=num3`

Method 2: Step 1: `if (num1>num2)`

`if (num1>num3) L=num1`

```

        ilse L=num3
    else
        ef (num2>num3) L=num2
        else L=num3

```

Method 3: Step 1: if (num1>num2) and (num1>num3) L=num1

Step 2: if (num2>num1) and (num2>num3) L=num2

Step 3: if (num3>num1) and (num3>num2) L=num3

Let us now look at the relative efficiency of the three methods. The first method requires us to perform two comparisons and three assignment operations. The second method requires three comparisons and four assignment operations. The last method requires six comparisons and three assignment operations.

From the above methods if we take the comparison operation as the key operation. The Generalisation to find the largest of "n" numbers is shown and is ideally to say that the first method requires (n - 1) comparisons. The second method may also require about (n - 1) comparisons but it looks difficult to analyze. The third method would require each number to be compared with each other number thus it would require about $n * (n - 1)$ comparisons i.e., it would have to perform n times more work.

As a result of this analysis we conclude that method 1 is the best method that is efficient and easy to generalize.

We can consider algorithms to be procedural solutions to problems

From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

There are a few types of problems that arise in computing applications quite often. We review them in the next section. If the problem in question is one of them, you might be able to use a known algorithm for solving it. Of course, it helps to understand how such an algorithm works with a certain data structure.

Some algorithms do not demand any ingenuity in representing their inputs. But others are, in fact, predicated on ingenious data structures. In addition, some of the algorithm depend intimately on structuring or restructuring data specifying a problem's instance.

1.3.2 CODING AN ALGORITHM

Most algorithms are destined to be ultimately implemented as computer programs. Programming an algorithm presents both a peril and an opportunity. The peril lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently. Some influential computer scientists strongly believe that unless the correctness of a computer program is proven with full mathematical rigor, the program cannot be considered correct. Hence the steps would be useless if they can be verified to prove the desired output.

Also note that throughout this book, I assume that inputs to algorithms may fall within their specified ranges or not and hence When implementing algorithms as programs following steps it satisfies the requirement and verifications are proven with the output.

Example 1.6: Recall that the greatest common divisor of two nonnegative, not -both-zero integers m and n , denoted $\text{gcd}(m, n)$, is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero. Euclid of Alexandria (third century B.C.) outlined an algorithm for solving this problem.

In modern terms, Euclid's algorithm is based on applying repeatedly the equality of two numbers.

$$\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$$

Euclid's algorithm for computing $\text{gcd}(m, n)$ can be structured like this

Step 1: If $n = 0$ then, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2: Divide m by n and assign the value of the remainder to r .

Step 3 Assign the value of n to m and the value of r to n . Go to Step 1.

Or in this way:

First Let us define the Greatest Common Divisor $GCD(m,n)$ recursively.

Step 1: If $m=n$ then , return m or n and stop; otherwise proceed to step 2

Step 2: If $m > n$ then , return $GCD(m-n,n)$, otherwise proceed to step 3

Step 3: If $m < n$ then , return $GCD(m,n-m)$

The above algorithm can be converted into its corresponding C code as follow:

```
#include<stdio.h>
#include<conio.h>
int GCD(int,int)
void main()
{
int m,n;
clrscr();
printf("Enter two positive numbers:");
scanf("%d%d",&m,&n);
printf("\n The GCD of %d and %d is %d", m,n,GCD(m,n));
getch();
}
Int GCD(int a,int b)
{
If(a==b) return a;
```

```
If(a>b) return GCD(a-b,b);  
else return GCD(a,b-a);  
}
```

Output

Enter two positive numbers: 25 15

The GCD of 24 and 15 is 5

SELF ASSESSMENT QUESTIONS

1. What do you mean by storage or memory representation of data?
2. What is meant by data structure?
3. Explain the need for correctness and efficiency while describing data structures.
4. How are data structures classified?
5. What is meant by primitive data Structure?
6. What is meant by non-primitive data structure?
7. How are non-primitive data structures further classified?
8. What is linear data structure?
9. What is non-linear data Structure?
10. Mention the different operations possible on primitive data structure.
11. Explain the creation and selection operations on primitive data structures with suitable illustrations.
12. What are the operations Possible on non-primitive data structures?
13. What is an algorithm?
14. How are algorithms important in the representation of solutions to problems?
15. What are the questions that are to be considered while tracing algorithms?
16. What do you mean by complexity of an algorithm?
17. With algorithm to generate N Fibonacci numbers using recursion function

PART 2

MEMORY ALLOCATION AND ARRAYS STRUCTURES

2.0 INTRODUCTION

A program operates on data. Data is stored in memory. While a program is executing, different values may be stored in the same memory location at different times. This kind of memory location is called a variable, and its content is the variable value. The symbolic name that we associate with a memory location is the variable name or variable identifier.

Definition
Variable: A location in memory, referenced by a program variable name (identifier), where a data value can be stored (this value can be changed during program execution).

Any variable, which is declared and initialized, has three things associated with it

1. A memory location with to hold the value of the variable,
2. The initialized value, which is stored in the location and
3. The address of that memory location.

All the three things are equally important. The name of a variable, which represents the memory location, is used to output the value stored in the variable and the address of the variable is used to input a value to the variable. Consider the following declaration statement in C.

```
int a;
```

In this declaration, **a** is the name of the variable that is declared to be of type **integer**. An integer variable **a** occupies two bytes of memory.

We have always used variables to store values, however a variable can also be used to store the address of another variable such variables are termed as pointer variables.

Thus a pointer is a variable, which can contain the address of another variable. The program below highlights this fact very clearly.

Program 2.1: To show the creation of a pointer variable.

```
#include <stdio.h>

Void main()

{

    int a=28;

    int *ptr;

    ptr=&a;

    printf("Address of the variable a=%u \n", &a);

    printf("Value of the variable a=%d \n", a);

    printf("Address present in the pointer variable ptr=%u \n", ptr);

}
```

OUTPUT

Address of the variable a=65119

Value of the variable a=28

Address present in the pointer variable ptr=65119

Observing the program closely highlights the following points:

1. The address of a variable is accessed with the help of the "&" operator.
2. Using the name of the variable we can access the value of a variable.

3. A pointer variable is created by including the operator "*" when the variable is declared.
4. A pointer variable can hold only the address of another variable and not the value of another variable.

2.1. POINTER DECLARATION

We have already seen the use of a pointer and its usage; in this section we try to understand all the concepts clearly. *A pointer is a variable that contains the address of the memory location of another variable.* To create a pointer variable we use the syntax as shown in the Figure below. First we will have to specify the type of data stored in the location identified by the pointer. Then a variable is created along with an asterisk. The asterisk tells the compiler that you are creating a pointer variable. Finally, you give the name of the variable.

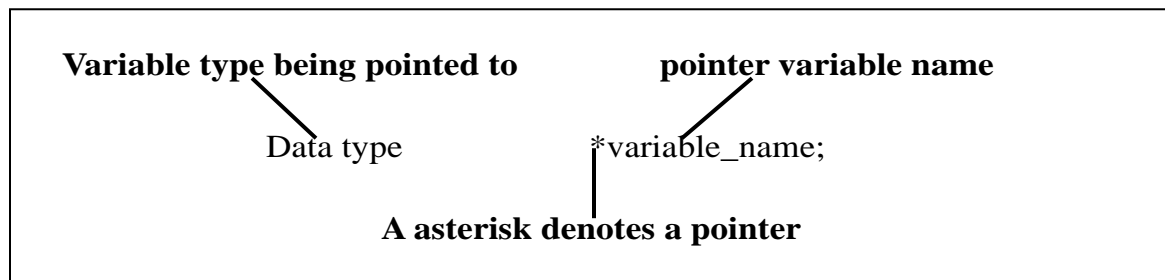


Figure 2.1.1 Declaration Syntax of a pointer variable

2.2. POINTER OPERATOR

A pointer operator is used to classify a variable as a pointer and not as a normal variable. Pointer variables can only store the address of another memory location, while a normal variable can only store a value and not an address. The classification is done by representing the variable with a combination of * (asterisk) with the variable name.

For example:

```
int    *ptr;
```

The base type of the pointer defines which type of variables the pointer is pointing to. Technically, any type of operator can point anywhere in memory. All pointer arithmetic is done relative to its base type. So it is important to declare the pointers correctly.

2.3. ADDRESS OPERATOR

Once we declare a pointer variable, we must make it to point to something. We can do this by assigning to the pointer the address of the variable you want to point to as in.

```
ptr=&s;
```

This places the memory address of the variable **s** into the pointer variable **ptr**. If **s** is stored in memory 65119 address, then the variable **ptr** dynamically has the value 65119 as the address of **s** and can easily access the value of **s**.

We can also assign an address to the variable **ptr** directly. Thus the instruction

```
ptr=65119
```

would generate a compiler error because it is attempting to assign an integer value to the pointer. The only assignment you can make to the variable **ptr** is the address of a variable, using the address operator, as

```
ptr=&s;
```

However, we can assign a value to the pointer ***ptr**, as in

```
ptr=280;
```

This means "Place the value 280 in the memory address pointed by the "" variable **ptr**." Since the pointer contains the address 65119, the value 280 is implicitly placed in that memory location. And since this is the location of the variable **s**, the value of **s** is also becoming 280. This shows that the change in memory **s** will immediately reflect in **ptr**.

Program 2.2: To display the contents of the variable and their address using a pointer variable.

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

```
void main ()  
  
{  
  
    int  num, *int_ptr;  
  
    float x, *float_ptr;  
  
    char  ch, *char_ptr;  
  
    num=123;  
  
    x=12.34;  
  
    ch='A';  
  
    int_ptr = &num;  
  
    float_ptr = &x;  
  
    char_ptr = &ch;  
  
    printf("Num %d is stored at the address %u\n", *int_ptr, int_ptr);  
  
    printf("Float %f is stored at the address %u\n", *float_ptr, float_ptr);  
  
    printf("Character %c is stored at the address %u\n", *char_ptr, char_ptr);  
  
}
```

OUTPUT

Num 123 stored at the address 65524

Value 12.340000000 stored at the address 65520

Character A stored at the address 65519

2.4. MEMORY SPACE REQUIREMENT FOR POINTER VARIABLES

The amount of memory space allotted when a variable is created and compiled depends on the data type of the variable. In this section we try to understand how much space will be allotted to a pointer variable of different types of data. The program below highlights this concept clearly.

Program 2.3: To show the amount of space required to store variables and space reserved for pointers.

```
#include <stdio.h>

void main ()
{
    int  a = 5, *int_ptr;
    char b = 'w', *char_ptr;
    float c = 17.53, *float_ptr;
    int_ptr = &a; float_ptr = &b; char_ptr = &c;
    printf("Value of integer = %d\n", a);
    printf("Value of character = %c\n", b);
    printf("Value of float = %f\n", c);
    printf("Amount of space for int ptr = %u bytes\n", sizeof(int_ptr));
    printf("Amount of space for char ptr = %u bytes\n", sizeof(char_ptr));
    printf("Amount of space for float ptr = %u bytes\n", sizeof(float_ptr));
}
```


OUTPUT

Value of integer = 5

Value of char = w

Value of float = 17.530000

Amount of space for int_ptr = 2 bytes

Amount of space for char_ptr = 2 bytes

Amount of space for float_ptr = 2 bytes

What is observed from the above described concept is a very important point. The amount of space to store different variables may vary. However the size of all the addresses available is the same and depends on the word length of the computer being used. Thus we observe that all the outputs are the same i.e., a pointer is created to hold the address of another memory location and the size of the address is the same immaterial of the type of data it holds, thus the outputs are the same.

2.5. POINTERS AND FUNCTIONS

Pointers are used very much with functions. Also sometimes complex function can easily be represented and accessed only with a pointer. Basically, Arguments can be passed and values can be accessed by using one of the following methods:

1. Passing values of the arguments (Call by Value)
2. Passing the addresses of the arguments (Call by Reference)

Definition of some key terms

Function: Function is a self contained block of codes that performs an inherent task of a program.

Formal Parameter: parameters that are present in a function definition

Actual Parameter: List of arguments presented whenever the function is invoked.

2.5.1 Calling by value

When a function is invoked, a correspondence is established between the formal and actual parameters. A temporary storage is created where the value of the actual parameter is stored. The formal parameter picks up its value from this storage area. This mechanism of data transfer, between the actual and formal parameters, allows the actual parameters to be an expression, arrays, etc. Such parameter is called value parameters and mechanism of data transfer is referred to as **Call-By-Value**. The corresponding formal parameter represents a local variable in the called function. The current value of the corresponding actual parameter becomes the initial value of the formal parameter. The value of formal parameter may then change in the body of the subprogram by assignment or input statements. This will not change the value of the actual parameter.

Definition
Value parameter: A parameter that receives a copy of the value of corresponding argument.

Program 2.4: A C program to illustrate the function using call by value mechanism.

```
#include <stdio.h>

void myfunction (int, int);

main ()
{
    int    a,b;
    a = 20;
    b = 30;
    printf ("a = %d b = %d before function call \n", a,b);
    myfunction (a,b);
    printf ("a = %d b = %d after function call \n", a,b);
}
```

```
/* Call by value function */
```

```
Void myfunction(int x, int y)
```

```
{
```

```
X=x+5;
```

```
Y=y-3;
```

```
}
```

OUTPUT

a = 20 b = 30 before function call

a = 20 b = 30 after function call

2.5.2. Call by reference

Whenever a function call, it is done as if we pass the address of a variable to a function, the parameters receiving the address should be pointers. The process of calling a function using pointers to pass the address of variable is known as **Call-By-Reference**. The function, which is called by 'reference', can change the value of the variables i.e., any changes made to the copied variables will dynamically affect the original variables.

Definition

Reference parameter: A parameter that receives the memory address of the caller's argument.

Program 2.5. A C program to illustrate the function using call by reference mechanism.

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

```
main ()
```

```
{
```

```
int a,b;
```

```

void function (int *, int *);

a = 20;
b = 30;

printf (“a = %d b = %d before function call \n”, a,b);

function(&a,&b);

printf (“a = %d b = %d after function call \n”, a,b);
}

/* Call by reference function */
void function(int *x, int *y)
{
    *x = *x + *x;
    *y = *y + *y;
}

```

OUTPUT

a = 20 b = 30 before function call

a = 40 b = 60 after function call

When the function is called, the address of the variable **a**, not its value, is passed into the function. In the function the receiving variables are declared as a pointer thus the address of the variable is passed. Since x represents the address of **a**, the value of **a** is changed from 20 to 40. Therefore, the output of the above program segment will be as shown above.

Program 2.6: A C program to interchange the contents of the two variables using call by value and call by reference.

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

```
void swap_val(int , int );    /* Function prototype */
```

```

void swap_ref(int * , int * );

void main ()

{

    int  a,b;

    printf("Enter two numbers \n");

    scanf ("%d %d", & a,&b);

    printf ("a = %d b = %d before function call \n", a,b);

    swap_byval (a,b);

    printf ("a = %d b = %d after function call using call by value \n", a,b);

    swap_byref (&a,&b);

    printf ("a = %d b = %d after function call using call by reference \n", a,b);

}

/*    Function to exchange two values using call by value    */

void swap_byval(int x, int y)

{

    int  temp;

    temp = x;

    x = y;

    y = temp;

}

/*    Function to exchange two values using call by reference    */

```

```
void swap_byref(int *x, int *y)
{
    int    temp;

    temp = *x;

    *x = *y;

    *y = temp;
}
```

OUTPUT

Enter two numbers

100 200

a = 100 b = 200 before function call

a = 100 b = 200 after function call using call by value

a = 200 b = 100 after function call using call by reference

Using the technique of call by reference in an intelligent manner it is possible for us to make a function return more than one value at any instant of time, whereas till now our function could return only one value.

2.6. STATIC AND DYNAMIC MEMORY ALLOCATION

In the previous section we have described the storage classes which determined how memory for variables is allocated by the compiler. When a variable is defined in the source program, the type of the variable determines how much memory the compiler allocates. When the program executes, the variable consumes this amount of memory regardless of whether the program actually uses the memory allocated. This is particularly true for arrays. However, in many problems, it is not clear at the outset how much memory the program will actually need. Some time, we have to declare memories to be "large enough"

to hold the maximum number of elements we expect our application to handle. If too much memory is allocated and then not used, there is a waste of memory. If not enough memory is allocated, the program is not able to handle the input data.

2.6.1. Static memory allocation

The method of allocating space when the variable is created is called as **static allocation** of space.

Definition
Static allocation: creation of storage space in memory for a variable at compiler-time (cannot be changed at run-time).

2.6.2. Dynamic memory allocation

In programming we may come across situations where we may have to deal with data, which is dynamic in nature. The number of data items may change during the executions of a program. The number of customers in a queue can increase or decrease during the process at any time. When the list grows we need to allocate more memory space to accommodate additional data items. Such situations can be handled more easily by using dynamic techniques. The method of allocating space while running the program is called as dynamic allocation of space. Dynamic data items at run time, thus optimizing usage of storage space.

Definition
Dynamic allocation: Creation of storage space in memory for a variable during run-time.

The above said concept can be achieved with the help of the four standard library functions **malloc()**, **calloc()**, **realloc()** and **freer()**.

C function	Task to be performed by the function
Malloc	Allocates memory requests size of bytes and returns a pointer to the first byte of allocated space.
calloc	Allocates space for an array of elements initializes them to zero and returns a pointer to the memory.
free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

2.6.2.1. Allocating a block of memory (malloc function)

A block of memory may be allocated using the function malloc. The malloc function reserves a block of memory of specified size and returns a pointer of type void. This means that we can assign it to any type of pointer. It takes the following form:

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

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.

Example 2.3: `ptr = (int *) malloc (100*sizeof(int));`

2.7 ARRAYS BASED MEMORY STRUCTURES

2.7.0 INTRODUCTION

An Array is a finite, ordered sequence of data items known as elements. "Ordered" in this definition means that each element has a position in the list. In other words, there is a first element in the array, a second element, and so on. All elements of the array must have the same data type, although there is no conceptual objection to lists whose elements have different data types.

An array is said to be **empty** when it contains no elements. The number of elements currently stored is called the **length or the size** of the array.

When presenting the contents of an array, our notation will be to enclose as of the list elements in parentheses and separate them by commas. For example, The list.

$$(a_0, a_1, \dots, a_{n-1})$$

contains **n** elements. The subscript indicates an element's position within the list. For all $i \geq 0$, the element with subscript **i** immediately follows the elements with subscripts **i-1**. Using this notation, the empty list would appear as ().

In this context we recognise One dimensional and multiple dimensional array types.

2.7.1 ARRAY IMPLEMENTATION

There are two standard approaches to implementing arrays, the array-based or sequential list, and the Linked List.

Definition
Array: A structured data type composed of a fixed number of components of the same type, with each component is directly accessed by the index.

The array-based list implementation stores the elements of the list in contiguous array positions. Array position corresponds to element positions. This implies that an array of some fixed size will be allocated, so the size of the array must be known when the list object is created. Since each list object may have a differently sized array, this size must be remembered by each list object. At any given time the list actually holds some number of elements that must be less than the maximum allowed by the array.

The possible operations on the array based list are:

- **Traversal:** Processing each item in an array exactly once.
- **Search:** Find the location of ITEM with a given value in an array.
- **Insertion:** Inserting an ITEM into an array.
- **Deletion:** Deleting an ITEM from the array.
- **Sorting:** Arranging the elements in some type of order.
- **Merging:** Combine one or more arrays to form single array.

2.7.1.1 ONE DIMENSIONAL ARRAY

Linear array is finite set of *homogeneous data elements* of size **N**, such that

- (a) The elements of the array are referenced by an *index* set consisting of **N** consecutive numbers.
- (b) The elements of the array are stored respectively in successive memory locations. The number **N** is called the **Length** of the array.
- (c) All elements have the same name.

Thus, an array is a linear data structure. There exists a linear relationship between the array elements. The array elements lie in the computer memory in a linear order. The syntax to define and create an array is: **Syntax:**

Data_type <i>Array-Name</i> [size];
--

Example 2.7.1: If we want to define an array which has 10 elements then the declaration will be:

```
int    number[10];
```

In the above example number is an array name with size 10. An array number has been defined integer in which we can store maximum 10 integer data. The size of an array is also called its length of the array. The subscripts of an array is also called index. The index set consists of integers. The length or the size of the array is obtained by the formula

$$\text{LENGTH} = \text{UB} - \text{LB} + 1$$

UB is the upper bound and **LB** is the lower bound. Here $\text{UB} = 9$ and $\text{LB} = 0$, hence the $\text{LENGTH} = 9 - 0 + 1 = 10$. Here index set is an integer. Array indexing is done by integers starting from 0 to $n-1$ where n represents the total number of elements to be allocated. The elements of the array are $A[0]$, $A[1]$, $A[9]$ in C.

In general form we represent array as

$A(0), A(1), A(2), \dots, A(9)$

OR

$A[0], A[1], A[2], \dots, A[9]$

2.7.1.2 REPRESENTATION OF LINEAR ARRAY IN MEMORY

Let A be a linear array of size 5. It can be represented in memory by $N=5$ sequential memory locations. To access each memory location index point is initialized to zero and set to point to first location, last index is set to be $5-1=4$

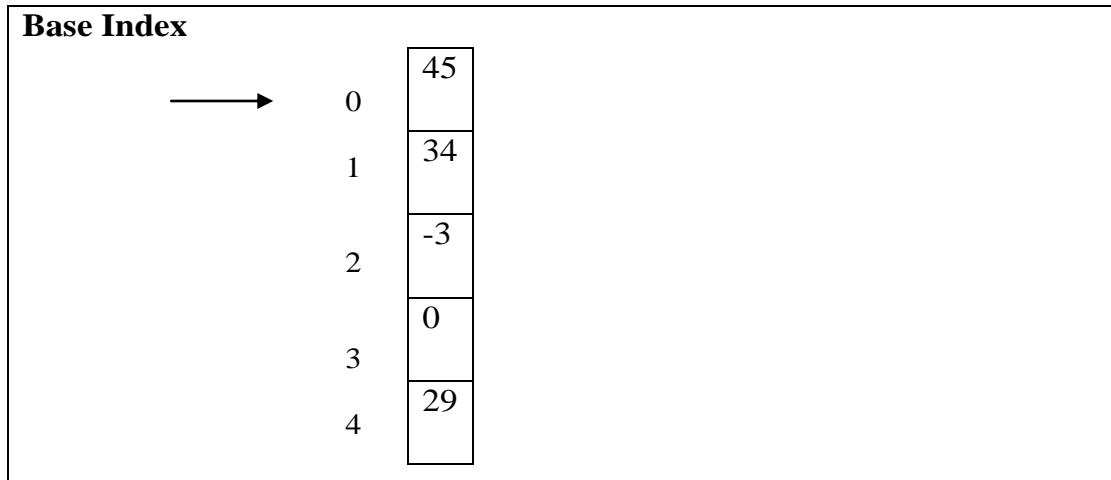


Figure 4.1 Linear array representation

2.7.1.3 IMPLEMENTATION OF ARRAY OPERATIONS

2.7.1.3.1 TRAVERSING ONE DIMENSIONAL ARRAY

Traversing operation is nothing but visiting each element exactly once and performing some processing. Here we have to access *all* elements of the array. For example, printing all the elements of an array is the traversing operation.

Algorithm 2.7: *Let A be a linear array with LB and UB as lower bound and upper bound. This algorithm traverses A by applying an operation PROCESS to each element of A.*

Step 1: FOR $I \leftarrow LB$ to UB DO

Step 2: Apply PROCESS to A[I]

[EndFor]

Step 3: END

The processing can be defined to print all element of array and/or to compute the sum or the products of all elements,

Algorithm 2.7.1: <i>Algorithm for printing elements of an array of size N.</i>
Step 1: FOR $I \leftarrow 0$ to N-1
Step 2: PRINT A[I] [EndFor]
Step 3: END

Program 2.7.1: Program to traverse an array (To find the largest of n numbers).

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

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

```
void main()
```

```
{
```

```
    int n,i,big,A[100];
```

```
    print("Enter number of elements");
```

```
    scanf("%d", &n);
```

```
    printf("Enter array elements\n");
```

```
    for(i=0; i<=n; i++)
```

```
    {
```

```
        scanf("%d", &A[i]);
```

```
    }
```

```
    /*     Traverse operation     */
```

```
    big=A[0];
```

```
    /*     Compare all the elements of array a with big exactly once     */
```

```
    for (i=0;i<n;i++)
```

```
    {
```

```
        if(big<A[i])
```

```

{
big=A[i];
}
}

printf("\n Largest element in the list is %d", big);

getch();
}

```

OUTPUT

```

Enter number of elements 5
Enter array elements
45
443
23
34
56
Largest element in the list is 443

```

Algorithm 2.7.2: *Algorithm for computing the sum of all elements of an array of size N.*

Step 1: FOR I 0 to N-1

Step 2: SUM ← SUM + A[I]

[EndFor]

Step 3: END

Program 2.7.2: Program to traverse an array (To find the largest of n numbers).

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

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

```
void main()
```

```
{
```

```

int n,i,sum,A[100];

print("Enter number of elements");

scanf("%d", &n);

printf("Enter array elements\n");

for(i=0; i<=n; i++)

{

scanf("%d", &A[i]);

}

/*    Traverse operation    */

sum=0;

for (i=0;i<n;i++)

{

Sum +=A[i];

}

printf("\n The Sum of elements is %d", sum);

getch();

}

```

OUTPUT

```

Enter number of elements 5
Enter array elements
45
44
23
34
56
The Sum of elements is 202

```

2.7.1.3.2 INSERTION AND DELETION

One can insert an ITEM at the end of the array provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array, then on the average half of the elements must be moved downward to new locations to accommodate the new element and keep order of the other elements.

Example 2.7.3: Consider an array A of eight elements. To add an element 35, then if it is be inserted at A[3], i.e., all the elements from A[3] has to be moved downward.

A[0]	10	A[0]	10
A[1]	20	A[1]	20
A[2]	30	A[2]	30
A[3]	40	A[3]	35
A[4]	50	A[4]	40
A[5]	60	A[5]	50
A[6]	70	A[6]	60
A[7]	80	A[7]	70
		A[8]	80

Figure 2.7.3 (a) Before Insertion

(b) After Insertion

Note: The position of an element in the array is equal to index plus one since array indexing start from zero.

Algorithm 2.7.2: Consider an array A of N elements. This algorithm insert an element $ITEM$ into K^{th} position in A .

Step 1: FOR $I \leftarrow N$ DOWN TO $K-1$ DO [Decrements I by one]

Step 2: $A[I] \leftarrow A[I-1]$

[EndFor]

Step 3: $A[K-1] \leftarrow ITEM$ [Insert $ITEM$]

Step 4: $N \leftarrow N+1$ [Increase number of elements by 1]

Step 5: END

Program 2.7.2: Program to insert an item into a list.

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

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

```
void main()
```

```
{
```

```
    int n,i,item,pos,A[100];
```

```
    print("Enter number of elements");
```

```
    scanf("%d", &n);
```

```
    printf("Enter array elements\n");
```

```
    for(i=0; i<=n; i++)
```

```
        scanf("%d", &A[i]);
```

```
    print("Enter an item to be inserted");
```

```
    scanf("%d", &item);
```

```
    printf("Enter position of the item");
```

```
    scanf("%d", &pos);
```

```

if(pos-1>n || pos-1<0)
printf("invalid position");
else
{
    /*   Push all the elements downword   */
    for(i=n;i>=pos-1;i--)
    {
        A[i]=A[i-1];
    }
}
A[pos-1]=item;

/*   Increment number of elements by one   */
n=n+1;

printf("\n List elements after insertion\n");

for(i=0; i<n; i++)
    printf("%d\t", A[i]);
}

```

OUTPUT

```

Enter number of elements 8
Enter array elements
10  20  30  40  50  60  70  80
Enter an item to be inserted 35
Enter position of the item  4
List element after insertion
10  20  30  35  40  50  60  70  80

```

Similarly deleting an element at the end of an array presents no difficulties, but deleting an element somewhere in the middle of an array would require that each subsequent element be moved one location upward in order to fill up the array.

Example 2.7.4: DELETION: Consider an array A of example 2.7.4. Suppose 40 is to be deleted, then move all the elements below 40 upwards.

A[0]	10	A[0]	10
A[1]	20	A[1]	20
A[2]	30	A[2]	30
A[3]	40	A[3]	50
A[4]	50	A[4]	60
A[5]	60	A[5]	70
A[6]	70	A[6]	80
A[7]	80	A[7]	

Figure 2.7.4 (a) Before deletion

(b) After deleting A[3]

Algorithm 4.3: Consider an array A of N elements. This algorithm delete an element at the K^{th} position and it is stored in variable ITEM for future use.

Step 1: ITEM \leftarrow A[K]

Step 2: FOR I \leftarrow K-1 TO N-1 DO

Step 3: A[I] \leftarrow A [I+1]

[EndFor]

Step 4: N \leftarrow N-1 [Decrement number of elements by 1]

Step 5: END

```

#include <stdio.h>

void main()
{
    Int A[100],n,i,pos;
    print("Enter number of elements in the array");

    scanf("%d", &n);

    printf("Enter array elements\n");

    for(i=0; i<n; i++)

        scanf("%d", &A[i]);

    printf("Enter position of the item to be deleted");

    scanf("%d", &pos);

    if(pos-1>n || pos-1<0)

        printf("invalid position");
    else
    {
        for(i=pos-1; i<n-1; i++)
            A[i]=A[i+1];

        n=n-1;

        printf("\n List elements after deletion\n");

        for(i=0; i<n; i++)

            printf("%d\t", A[i]);

    }

}

```

OUTPUT

Enter number of elements in the array 5

Enter array elements

1 2 3 4 5

Enter position of the item to be deleted 3

List after deletion

1 2 4 5

2.7.1.3.3 SEARCHING AND SORTING

Sorting and Searching are fundamental operations in computer science. Sorting refers to the operation of arranging data in some given order. Searching refers to the operation of searching the particular record from the existing information. Normally, the information retrieval involves searching, sorting and merging. In this chapter we will discuss the searching and sorting techniques in detail.

2.7.1.3.3.1 SEARCHING

All our lives we sometime get stagger looking for somebody or the other or for something or the other in the area. Thus searching forms an important activity of our lives. In computers we also observe that this operation is very important, thus it is necessary for us to learn an efficient method of performing this operation.

In this section we shall consider some of searching Techniques of organizing data make the search process more efficient.

Definition

Searching: Searching refers to the operation of finding the location of a given item in a collection of items structure.

The search is said to be successful if ITEM does appear in DATA and unsuccessful otherwise.

Any algorithm, which performs a search operation, accepts an argument we call as "ele" and tries to find an occurrence of the key "ele" in some structure. It is possible that the search for a particular element in a structure is unsuccessful, i.e., there is no occurrence in the structure with the argument "ele" as its key. In such a case the algorithm should return a special message to indicate an unsuccessful search.

Clearly, Searches in which the entire array is constantly in main memory are called internal searches, whereas those in which most of the table is kept in auxiliary storage are called external searches.

A number of methods are available to perform this operation, two important methods here are **Linear Search** and **Binary search**.

2.7.1.3.2.1.1 LINEAR SEARCH

This is the most natural searching method. The most intuitive way to search for a given ITEM in Data Structure is to compare ITEM with each element of Data structure one by one. For example we may use this method when we are searching for a student in a campus and we have no idea about the class in which he is studying.

Let us assume that **A** is an array (say it represents the campus) of **N** elements from **A[0]** through **A[N-1]** (this represents the classrooms). Let us also assume that **ele** is the search element (this represents the student we are searching). The search starts by sequentially comparing the elements of the array one after the other from the beginning to the end with the element to be searched i.e., we search for the student one classroom at a time from the beginning to the end. If the element is found its position is identified otherwise an appropriate message is displayed.

Algorithm 5.1: Linear_search (A, ele, N) –A is an array of N elements and ele is the element being searched in the array.
Step 1: LOC = -1
Step 2: For I = 0 TO N-1 Do

```

Step 3: If(ele = A[I] Then

Step 4:      LOC = I

Step 5:      Goto Step 6

            [EndIf]

            [End of For loop]

Step 6:  If(LOC >= 0) Then

Step 7:      Print ele, "Found in location", LOC

Step 8:  Else

            Print ele, "Not found"

            [EndIf]

Step 10: Exit

```

This above algorithm starts the search from the first location of the array i.e., I=0. If the search element is found in a particular location its position is noted in the variable LOC. Finally at the end the appropriate message is displayed.

Example 1: Consider the following elements stored in array

12	23	9	17	7
A[0]	A[1]	A[2]	A[3]	A[4]

And we are searching for the element 17 i.e., ele = 17. The trace of the algorithm is given below.

ARRAY	VARIABLES	COMPARISON										
<table><tr><td>12</td><td>23</td><td>9</td><td>17</td><td>7</td></tr><tr><td>A[0]</td><td>A[1]</td><td>A[2]</td><td>A[3]</td><td>A[4]</td></tr></table> <div><div></div><div>I</div></div>	12	23	9	17	7	A[0]	A[1]	A[2]	A[3]	A[4]	<div>I = 0</div> <div>LOC = -1</div>	17 = 12
12	23	9	17	7								
A[0]	A[1]	A[2]	A[3]	A[4]								
<table><tr><td>12</td><td>23</td><td>9</td><td>17</td><td>7</td></tr><tr><td>A[0]</td><td>A[1]</td><td>A[2]</td><td>A[3]</td><td>A[4]</td></tr></table> <div><div></div><div>I</div></div>	12	23	9	17	7	A[0]	A[1]	A[2]	A[3]	A[4]	<div>I = 1</div> <div>LOC = -1</div>	17 = 23
12	23	9	17	7								
A[0]	A[1]	A[2]	A[3]	A[4]								
<table><tr><td>12</td><td>23</td><td>9</td><td>17</td><td>7</td></tr><tr><td>A[0]</td><td>A[1]</td><td>A[2]</td><td>A[3]</td><td>A[4]</td></tr></table> <div><div></div><div>I</div></div>	12	23	9	17	7	A[0]	A[1]	A[2]	A[3]	A[4]	<div>I = 2</div> <div>LOC = -1</div>	17 = 9
12	23	9	17	7								
A[0]	A[1]	A[2]	A[3]	A[4]								
<table><tr><td>12</td><td>23</td><td>9</td><td>17</td><td>7</td></tr><tr><td>A[0]</td><td>A[1]</td><td>A[2]</td><td>A[3]</td><td>A[4]</td></tr></table> <div><div></div><div>I</div></div>	12	23	9	17	7	A[0]	A[1]	A[2]	A[3]	A[4]	<div>I = 3</div> <div>LOC = 3</div>	17 = 17
12	23	9	17	7								
A[0]	A[1]	A[2]	A[3]	A[4]								

Since the value of LOC is greater than 0, we get the output as

Ele Found in location 3

And if we are searching for the element 1 i.e., ele = 1. The trace of the algorithm is given below.

ARRAY	VARIABLES	COMPARISON								
<table><tr><td>12</td><td>23</td><td>9</td><td>17</td><td>7</td></tr><tr><td>A[0]</td><td>A[1]</td><td>A[2]</td><td>A[3]</td><td>A[4]</td></tr></table> <div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></</div></div>	12	23	9	17	7	A[0]	A[1]	A[2]	A[3]	A[4]
12	23	9	17	7						
A[0]	A[1]	A[2]	A[3]	A[4]						

Since the value of LOC is not greater than 0, we get the output as **Ele not found**

Efficiency of linear Search

The efficiency of linear search can be understood by examining the number of comparisons made by a linear search in searching for a given element. We assume that no insertions or deletions are performed, so that the searching is performed through an array of a constant size N . The number of comparisons depends on where the element appears in the array. If the element is in the first position of the array, only one comparison is to be performed, if the element is in the last position of the array then N comparisons are necessary to find the element. If it is equally likely for the argument to appear at any given array position, a successful search will take $(n + 1)/2$ comparisons, and an unsuccessful search will take n comparisons. On an average the number of comparisons is $O(n)$.

Program 1: Program to perform linear search.

```
#include <stdio.h>

#include <conio.h>

int linear_search (int*, int, int);

void main()
{
    int A[100],ele,pos,n,i;

    clrscr();

    printf("Enter the size of the array:");

    scanf("%d", &n);

    printf("\n Enter %d elements\n", n);

    for(i=0; i<n; i++)

        scanf("%d", &A[i]);
```

```

printf("\n Enter element to search:");

scanf("%d", &ele);

pos = Linear_search(A,ele,n);

if(pos==-1)

    printf("\n Element not present");

else

    printf("\n Element found in position: %d", pos);

}

/*    Function to search an item using linear search technique    */

int linear_search(int a[ ], int e, int m)

{

    int i;

    for(i=0; i<m; i++)

        if(a[i]==e) return (i+1);

    return(-1);

}

```

OUTPUT

Enter the size of the array: 5

Enter 5 elements

12 45 34 67 23

Enter element to search: 67

Element found in position: 4

Enter the size of the array: 5

Enter 5 elements

15 47 39 65 28

Enter the element to search: 67

Element not present

2.7.1.3.2.1.2 BINARY SEARCH

The main disadvantage of linear search is the time taken to decide. The decision of an element not being present can be made only after comparing N element. This problem can be overcome when the elements are in sorted order with the technique called binary search. Binary search is considered the most efficient, method of searching a linear array without the use of auxiliary indices. The element to be compared is compared with the element of the middle element of the array. If the elements are the same, the search ends successfully; otherwise, the search should continue either to the left or to the right of the array in a similar manner. It may be noted that the binary search can be applied if and only if the array is sorted in some form of order.

The steps of binary search may be summarized as follows:

1. Find the position of the middle element of the array.
2. Compare the element in the middle position with the search element.
3. One of the following actions may be performed after the comparison.
 - a. If the search element is the same as the middle element then note its position.

- b. Otherwise if the search element is less than the element in the middle position then continue the search to the left portion of the middle element.
- c. Otherwise if the search element is greater than the element in the middle position then continue the search to the right portion of the mid.

Algorithm 5.2: Binary_search (*A, ele, N*) – *A* is a sorted array of *N* elements and *ele* is the element being searched in the array. **LOW** and **HIGH** identify the positions of the first and last elements in a range and **MID** identifies the position of the middle element.

Step 1: LOW = 0

Step 2: HIGH = N-1

Step 3: LOC = -1

Step 4: While (LOW <= HIGH) Do

Step 5: MID = (LOW+HIGH)/2

Step 6: If(ele = A[MID]) Then

LOC = MID

GO TO Step 9

[EndIf]

Step 7: If(ele < A[MID]) Then

HIGH = MID-1

Step 8: Else

LOW = MID+1

[EndIf]

[End of while loop]	
Step 9:	If(LOC >= 0) Then
	Print ele, Found in location", LOC
Step 10:	Else
Step 11:	Print ele, "Not found"
	[EndIf]
Step 12:	Exit

Example 2: Consider the following elements stored in an array

12	23	29	37	45
A[0]	A[1]	A[2]	A[3]	A[4]

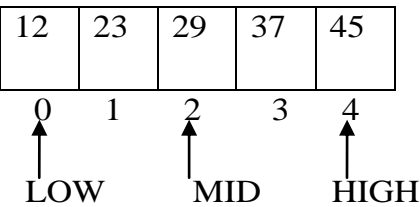
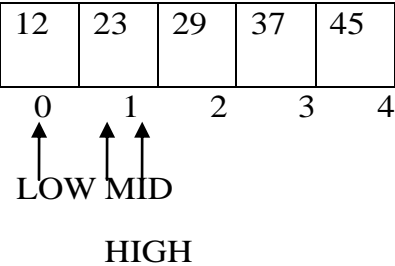
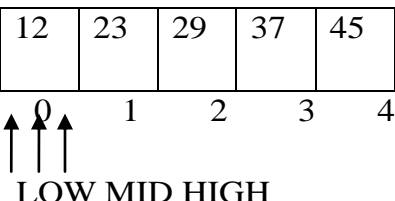
And we are searching for the element 37 i.e., ele = 37. The trace of the algorithm is given below.

ARRAY	VARIABLES	COMPARISON																									
<table><tr><td>12</td><td>23</td><td>29</td><td>37</td><td>45</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>↑</td><td></td><td>↑</td><td></td><td>↑</td></tr><tr><td>LOW</td><td></td><td>MID</td><td></td><td>HIGH</td></tr></table>	12	23	29	37	45	0	1	2	3	4	↑		↑		↑	LOW		MID		HIGH	LOW = 0 HIGH = 4 MID = 2 LOC = -1	37 = 29					
12	23	29	37	45																							
0	1	2	3	4																							
↑		↑		↑																							
LOW		MID		HIGH																							
<table><tr><td>12</td><td>23</td><td>29</td><td>37</td><td>45</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td></td><td></td><td>↑</td><td>↑</td><td>↑</td></tr><tr><td></td><td></td><td>LOW</td><td></td><td></td></tr><tr><td></td><td></td><td>MID</td><td></td><td>HIGH</td></tr></table>	12	23	29	37	45	0	1	2	3	4			↑	↑	↑			LOW					MID		HIGH	LOW = 3 HIGH = 4 MID = 3 LOC = 3	37 = 37
12	23	29	37	45																							
0	1	2	3	4																							
		↑	↑	↑																							
		LOW																									
		MID		HIGH																							

Since the element in the middle position is the same as the search element LOC gets the value 3. Since the value of LOC is greater than 0, we get the output as

37 Found in location 3

If we are searching for the element 7 i.e., $ele = 7$. The trace of the algorithm is given

ARRAY	VARIABLES	COMPARISON
	LOW = 0 HIGH = 4 MID = 2 LOC = -1	$7 = 29$
	LOW = 0 HIGH = 1 MID = 1 LOC = -1	$7 = 23$
	LOW = 0 HIGH = 0 MID = 0 LOC = -1	$7 = 12$

Since the element in the middle position is not the same as the search element and the search element is less than the element in the middle position HIGH gets the value

MID - 1 i.e., -1. Since the value of HIGH is greater than LOW the search is stopped. And since the value of LOC is not greater than 0, we get the output as

7 Not found

Efficiency of Binary search

Every comparison in the binary search reduces the number of possible elements remaining for comparisons by a factor of 2. Thus, the maximum number of element comparisons is approximately $\log 2^n$. We may say that the binary search algorithm is $O(\log n)$. If the size of the array is small then binary search is not more advantageous than a linear search. Unfortunately, the binary search algorithm can only be used if the array is stored. This is because it makes use of the fact that the indices of array elements are consecutive integers.

Program 2: Program to perform binary search

```
#include <stdio.h>

#include <conio.h>

int Binary_search (int*, int, int);

void main()
{
    int A[100],ele,loc,n,i;

    clrscr();

    printf("Enter the size array size:");

    scanf("%d", &n);

    printf("\n Enter %d elements in sorted order\n", n);

    for(i=0; i<n; i++)
```



```

        scanf("%d", &A[i]);

printf("\n Enter the element to search:");

scanf("%d", &ele);

loc = Binary_search(A,ele,n);

if(loc==-1)

    printf(" Element not present\n");

else

    printf("Element found in %d position \n", loc+1);

}

/*    Function to perform Binary search    */

int Binary_search(int A[ ], int e, int m)

{

    int  low, high, mid;

    low = 0;

    high = m-1;

    while (low <= high)

    {

        mid = (low + high)/2;

        if(e == a[mid])

            return (mid)

        if(e < a[mid])

```

```

        high = mid-1;

    else

        low = mid+1;

    }

    return(-1);

}

```

OUTPUT

```

Enter the array size: 5
Enter 5 elements in sorted order
12  23  34  45  67
Enter element to search: 67
Element found in 5 position

Enter the array size: 5
Enter 5 elements in sorted order
15  28  39  47  65
Enter the element to search: 67
Element not present

```

2.7.1.3.3.2 SORTING

This is another important activity of our lives. Whenever our lives are in a untidiness, we will have to arrange it in order, if we generalize we observe that every activity of life requires us to be organized for the effective execution of the Job. The librarian of a library has to keep the books sorted for effective use of the library. The Job of data processing requires us to have all the data in some order or the other, this minimizes the operation of searching for the correct data.

Definition
Sorting: Sorting refers to arranging of data elements in some type of order.

Ordering or sorting of data with some relationship is of fundamental importance. Certain factors however should be considered before designing a sort algorithm. Algorithms are designed with the following objectives.

1. The movement of data should be as minimum as possible. If the size of the data Item is large and if there is excessive movement of data, this would result in a large amount of processing time.
2. The movement of data items between the secondary storage and the main memory should be in large blocks. The larger the data block the more efficient is the process.
3. As much as possible the data should be retained in the main memory, so that it can be effectively used.

Careful consideration should be made before considering an algorithm. It is of no meaning if we write a complex algorithm to sort a small amount of data. In such situations any sorting algorithm would perform the job effectively. As the size of the data becomes larger minimizing processing time becomes most necessary; thus an efficient algorithm should be considered. While choosing an algorithm it is very much necessary to have thorough knowledge of the data, this helps in assuming the “**worst case**” scenario and the “**best case**” scenario.

There is a number of sorting techniques:

1. Bubble sort
2. Selection sort
3. Insertion sort

4. Merge sort
5. Heap sort
6. Quick sort

In this section we discuss Bubble sort, Selection sort, Insertion sort. Merge sort and Quick sort.

2.7.1.3.3.2.1 BUBBLE SORT

The algorithm achieves its name from the fact that with, each iteration a number moves like a bubble to its appropriate position. However the algorithm is not efficient for large arrays. The method of bubble sort relies heavily on an exchange mechanism to achieve its goals. The method is also called as “**sorting by exchange**”.

In this sorting algorithm, multiple swapping take place in one iteration. Smaller elements move or ‘bubble’ up to the top of the list. In this method ,we compare the adjacent members of the list to be sorted , if the item on top is greater than the item immediately below it, they are swapped.

During the next pass the same steps are repeated from the beginning of the array, however this time the comparisons are only for $n-1$ elements. The process is repeated again and again until only two elements are left for comparison. The last iteration ensures that the first two elements of the array are placed in the correct order.

Suppose the list of numbers $A[0]$, $A[1]$, $A[2]$ and $A[3]$ is in memory.

33	44	22	11
$A[0]$	$A[1]$	$A[2]$	$A[3]$


The bubble sort algorithm works as follows:

PASS 1			
1. Compare A[0] and A[1] and arrange them In the desired order. so that A[1] > A[0]. 2. Then compare A[1] and A[2] and arrange them so that A[2] > A[1] 3. Then compare A[2] and A[3] and arrange them so that A[3] > A[2]			
COMPARISON	ELEMENTS	RESULT	ARRAY
A[0] with A[1]	33 and 44	No interchange	33 44 22 11
A[1] with A[2]	44 and 22	Interchange	33 22 44 11
A[2] with A[3]	44 and 11	Interchange	33 22 11 44

Observe that Pass 1 involves N-1 comparisons. During Pass 1 the largest element is bubbled up to (N-1)th position. When Pass 1 is completed, A[N-1] will contain the largest element.

PASS 2			
Repeat pass 1 with one less comparison i.e., now we stop after we compare and possible rearrange A[N-3] and A[N-2].			
COMPARISON	ELEMENTS	RESULT	ARRAY
A[0] with A[1]	33 and 22	Interchange	22 33 11 44
A[1] with A[2]	33 and 11	Interchange	22 11 33 44

When pass 2 is completed, A[N-2] will contain the second largest element.

PASS 3			
Repeat pass 1 with one lesser comparison i.e., now we stop after we compare and possible rearrange A[N-4] and A[N-3].			
COMPARISON	ELEMENTS	RESULT	ARRAY
A[0] with A[1]	22 and 11	Interchange	 11 22 33 44

i.e., we finally compare A[0] and A[1] and arrange them so that $A[0] < A[1]$.

After n-1 passes, the list will be sorted in increasing order. The following algorithm describes bubble sort for sorting the array of "N" elements in the ascending order:

Algorithm 5.3: Bubble_sort (A,N) Given an array **A** of **N** elements, this procedure sorts the elements in the ascending order using the method described above. The variables **I** and **J** are used to index the array elements.

Step 1: For I = 1 to N – 1 Do

Step 2: For J = 0 to N – 2 Do

Step 3: [Compare adjacent elements]

 If ($A[J] > A[J + 1]$) Then

 [Exchange the values]

Step 4: Temp = A[J]

 A[J] = A[J + 1]

 A[J + 1] = Temp

Step 5: [End If]

 [End of Step 2 For loop]

 [End of Step 1 For loop]

Step 6: Exit

Analysis of Bubble sort

In analyzing the procedure of bubble sort, we shall count only the number of element comparisons. It is easy to see that the frequency count of all other operations is the same as that of the comparisons. In the algorithm we assume that all the elements are distinct.

The first pass of the algorithm results in $N-1$ comparisons and in the worst case may result in $N-1$ exchanges also. The second pass results in $N-2$ comparisons and in the worst case may result in $N-2$ exchanges. Continuing the analysis we observe that as the iterations or passes increase the comparisons and exchanges decreases. Finally the total number of comparisons will be equal to

$$= (N - 1) + (N - 2) + (N - 3) + \dots + 2 + 1$$

$$= (N) * (N - 1) / 2$$

$$= O(N^2)$$

Program 5.3: Program to arrange **N** numbers in the ascending and descending order using bubble sort method.

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

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

```
void main()
```

```
{
```

```
int i,j,n,A[100],temp;
```

```

clrscr();

printf("Enter number of elements:");

scanf("%d",&n);

printf("Enter those element:\n");

for(i=0;i<n;i++)

{

    scanf("%d",&A[i]);

}

for(i=0;i<n;i++)

{

    for(j=0;j<n-1;j++)

    {

        if(A[j]>A[j+1])

        {

            temp=A[j];

            A[j]=A[j+1];

            A[j+1]=temp;

        }

    }

}

printf("After sorting here is the elements in ascending order:\n");

for(i=0;i<n;i++)

{

    printf("%d\t",A[i]);

}

getch();

}

```


OUTPUT

Enter number of elements 6

Enter those element:

18 4 56 78 45 -13

After sorting here is the elements in ascending order:

-13 4 18 45 56 78

2.7.2 SELECTION SORT

This is a sorting algorithm, which is very simple to understand and implement. The algorithm achieves its name from the fact that with each iteration the smallest element for a key position is selected from the list of remaining elements and put in the required position of the array i.e., In this sorting we find the smallest element in this list and put it in the first position. Then find the second smallest element in the list and put it in the second position. And so on.

The method of selection sort relies heavily on a comparison and swapping mechanism to achieve its goals.

ALGORITHM DESCRIPTION

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
K = 1, LOC = 3	77	33	44	11	88	22	66	55
K = 2, LOC = 5	11	33	44	77	88	22	66	55
K = 3, LOC = 5	11	22	44	77	88	33	66	55
K = 4, LOC = 5	11	22	33	77	88	44	66	55
K = 5, LOC = 7	11	22	33	44	88	77	66	55
K = 6, LCjC = 6	11	22	33	44	55	77	66	88
K = 7, LOC = 6	11	22	33	44	55	66	77	88
Sorted:	11	22	33	44	55	66	77	88

The following algorithm describes selection sort for sorting the array of “N” elements in the ascending order.

Algorithm 5.4: Selection_sort (A,N) –Given an array **A** of **N** elements, this procedure sorts the elements in the ascending order using the method described above. The variables **I** and **J** are used to index the array elements.

Step 1: For I = 0 TO N – 2 Do

Step 2: [Assume i^{th} elements as smallest]

Small = A[I]

Step 3: POS = I

Step 4: [Find the smallest element in the array and its position]

For J = I + 1 TO N – 1 Do

Step 5: If(A[J] < small) Then

Step 6: small = A[J]

Step 7: POS = J

[EndIf]

[End of Step 4 For loop]

Step 8: [Exchange i^{th} element with smallest element]

A[POS] = A[I]

Step 9: A[I] = small

[End of Step 1 For loop]

Step 10: Exit

2.7.2.3 Analysis of Selection sort

In analyzing the procedure of selection sort, we shall count only the number of element comparisons. It is easy to see that the number of exchanges is the same as that of the number of elements. In the algorithm we assume that all the elements are distinct.

The first pass of the algorithm results in $N-1$ comparisons. The second pass results in $N-2$ comparisons. Continuing the analysis we observe that as the iterations or passes increases the comparisons decreases. Finally the total number of comparisons will be equal to

$$\begin{aligned} & (N-1) + (N-2) + (N-3) + \dots + 2 + 1 \\ &= (N) * (N-1) / 2 \\ &= O(N^2) \end{aligned}$$

We observe that in spite of the superiority of the selection sort algorithm over the bubble sort algorithm, there is no significant gain in run time. The efficiency of the algorithm still remains as $O(N^2)$ for “N” data items.

Program 5.4: Program to perform selection sort.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,k,n,A[100],temp,pos;
clrscr();
printf("Enter number of elements:");
scanf("%d",&n);
printf("Enter those element:\n");
```

```

for(i=0;i<n;i++)
{
    scanf("%d",&A[i]);
}
for(k=0;k<n-1;k++)
{
    pos=k;
    for(j=k+1;j<n;j++)
    {
        if(A[pos]>A[j])
        {
            pos=j;
        }
    }
    temp=A[k];
    A[k]=A[pos];
    A[pos]=temp;
}
printf("After sorting here is the elements in ascending order\n");
for(i=0;i<n;i++)
{
    printf("%d\t",A[i]);
}
getch();
}

```

OUTPUT

Enter number of elements:5

Enter those element:

67 34 12 54 89

After sorting here is the elements in ascending order

12 34 54 67 89

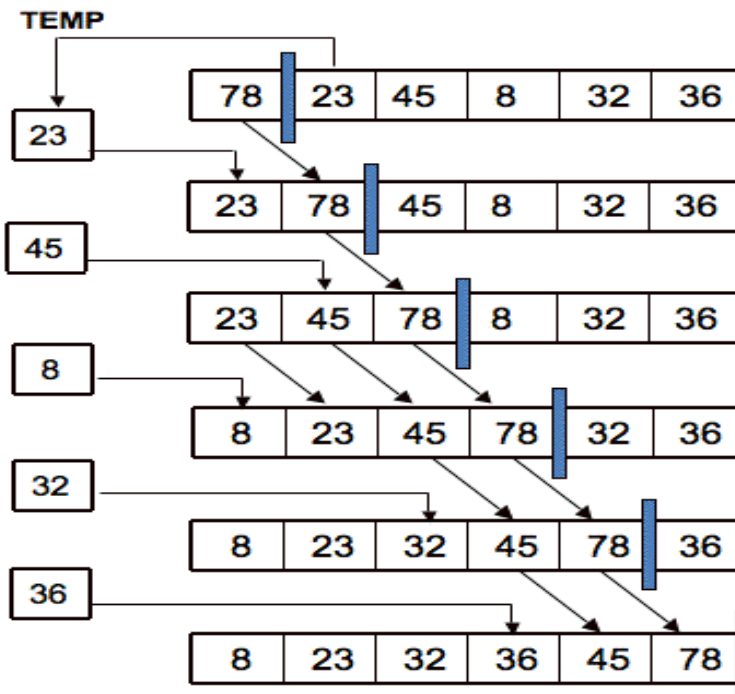
2.7.3 INSERTION SORT

This is one of the natural ways using which information can be sorted. It follows the same procedure that is used by card game players. The main idea of the algorithm is to build a complete solution by inserting a new element from the unsorted portion of a list into the appropriate position in the partially ordered portion of the list. The procedure is extended one element at a time for the entire list. The method employed is similar to the one used in selection sort where we select the smallest element from the unordered part of the list and place it at the end of the sorted part.

The algorithm of insertion sort functions as follows:

Initially, the first element is considered to be in the ordered part. The second element is then inserted either in the first or the second position as appropriate. This process extends the ordered part to two elements and the remaining elements are considered to be unordered. Inserting the third element, then the fourth element and so on slowly extends the ordered part. The outline for the insertion sort algorithm is as follows

LOGICAL DESCRIPTION OF INSERTION ALGORITHM



Algorithm 5.5: Insertion_sort (A,N) Given an array **A** of **N** elements, this procedure sorts the elements in the ascending order using the method described above. The variables **I** and **J** are used to index the array elements.

Step 1: For $I = 1$ to $N - 1$ Do

Step 2: $J = I$

Step 3: While $((J \geq 1) \text{ AND } (A[J] < A[J-1]))$

Step 4: If $(A[J] < A[J - 1])$ Then

Step 5: $\text{temp} = A[J]$

$A[J] = A[J - 1]$

$A[J - 1] = \text{temp}$

 [End If]

Step 6: J = J-1

[End of While loop]

[End of Step 1 For loop]

Step 7: Exit

2.7.3.3 Analysis of Insertion sort

In analyzing the procedure of insertion sort, we will have to count the number of element comparisons and the number of array elements that need to be shifted or moved. It is easy to see that the number of exchanges is the same as that we have seen in bubble sort. In the algorithm we assume that all the elements are distinct.

The first pass of the algorithm results in one comparison and in the worst case may result in one exchange also. The second pass results in two comparisons and in the worst case may result in two exchanges. Continuing the analysis we observe that as the iterations or passes increases the comparisons and exchanges increases. Finally the total number of comparisons will be equal to

$$\begin{aligned} & 1 + 2 + 3 + \dots + (N - 3) + (N - 2) + (N - 1) \\ = & (N) * (N - 1) / 2 \\ = & O(N^2) \end{aligned}$$

Program 5.5: Program to perform insertion sort

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

void main()
{
    int  a[100],n;
    int  i,j,temp;
```

```

clrscr();

printf("\nEnter number of elements");

scanf("%d", &n);

printf("\nEnter array elements\n");

for(i=0; i<n; i++)

scanf("%d", &a[i]);

printf("\n The unsorted list is\n");

for(i=0; i<n; i++)

    printf("\t%d", a[i]);

    printf("\n");

for(i=1; i<n; i++)

{

j = i;

while((j>=1) && (a[j] < a[j-1]))

{

    if(a[j] < a[j-1])

    {

        temp = a[j];

        a[j] = a[j-1];

        a[j-1] = temp;

    }

    j=j-1;

}

}

printf("\n The sorted list is \n");

for(i=0; i<n; i++)

printf("\t%d", a[i]);

```



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

OUTPUT

Enter number of elements 5

Enter array elements

67 34 12 54 89

The unsorted list is

67 34 12 54 89

The sorted list is

12 34 54 67 89

2.7.4 Quick sort

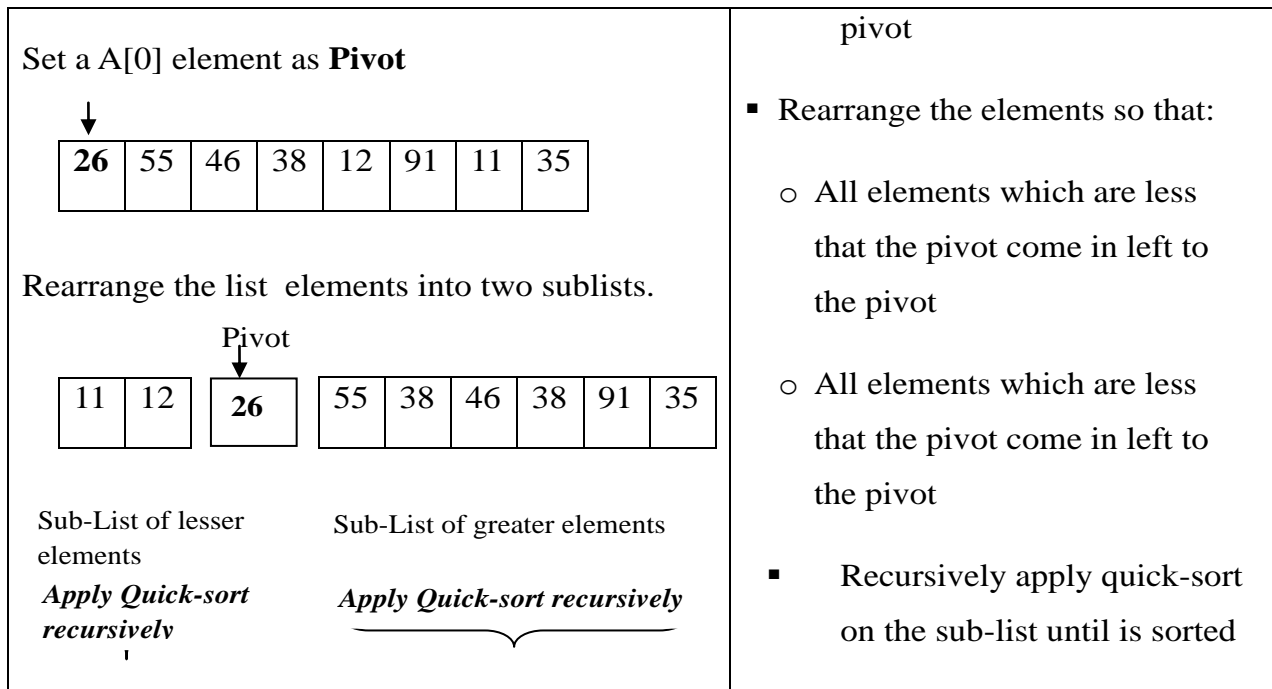
This is one of the best techniques for a large set of data. This technique also works on the method of partitioning. The process of divide and conquer is again recursively applied here. However in this method of sorting the partition is created at a position such that the elements to the left of the partition is less than the elements to the right of the partition.

The purpose of quick sort is to move the data item in the correct direction just enough for it to reach its final place. Thus the amount of swapping is greatly reduced and the required item moves a greater distance in a shorter duration of time. Let us now trace the entire sequence for the following set of data.

26	55	46	38	12	91	11	35
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

The entire process can be performed recursively.

Original-list of 8 elements	Description of Algorithm:																
<table><tr><td>26</td><td>55</td><td>46</td><td>38</td><td>12</td><td>91</td><td>11</td><td>35</td></tr><tr><td>A[0]</td><td>A[1]</td><td>A[2]</td><td>A[3]</td><td>A[4]</td><td>A[5]</td><td>A[6]</td><td>A[7]</td></tr></table>	26	55	46	38	12	91	11	35	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	<ul style="list-style-type: none">Set the first element in the list as
26	55	46	38	12	91	11	35										
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]										



1.4.4.1. Analysis of Quick sort

In analyzing quick sort we shall count the number of element comparisons $C(n)$. It is observed that the frequency count of all other operations are of the same order $C(n)$. The following assumptions are also made during the analysis

1. The elements of the array are distinct.
2. The partition element is chosen using a random selection process.
3. The proper position for the pivot element always turns out to be the middle or the sub array.
4. The array size is assumed to be a power of 2 , say $n = 2^m$ or $m = \log_2 n$.

In that case there will be approximately n comparisons in the first pass, after which the array is split into two sub arrays each approximately of the size $n/2$. For each of the sub arrays there are approximately about $n/2$ comparisons and a total of four sub arrays each of size $n/4$ are formed. Each of the sub arrays then requires $n/4$ comparisons and yielding $n/8$ sub arrays. After repeating the process "m" times then there will be "n" sub arrays each of size 1. Thus the total number of comparisons is approximately equal to

$$C(n) = n + 2 * (n / 2) + 4 * (n / 4) + + n * (n / n)$$

$$= n + n + n + + n \text{ (m times)}$$

$$= O(n * m)$$

$$= O(n \log n)$$

Program 5.6: Program to perform quick sort

```
#include<stdio.h>
#include<conio.h>
int partition(int a[], int beg, int end)
{
    int left,right,loc,flag=0,pivot;
    loc=beg;
    left=beg;
    right=end;
    pivot=a[loc];
    while(flag==0)
    {
        while((pivot<=a[right])&&(loc!=right))
        {
            right--;
        }
        if(loc==right)
        {
            flag=1;
        }
        else
        {
```

```

    a[loc]=a[right];
    left=loc+1;
    loc=right;
}
while((pivot>=a[left])&&(loc!=left))
{
    left++;
}
if(loc==left)
{
    flag=1;
}
else
{
    a[loc]=a[left];
    right=loc-1;
    loc=left;
}
}
a[loc]=pivot;
return loc;
}

void quick_sort(int a[], int beg,int end)
{
    int loc;
    if(beg<end)
    {

```

```

loc=partition(a,beg,end);
quick_sort(a,beg,loc-1);
quick_sort(a,loc+1,end);
}
}

void print_array(int a[],int n)
{
int i;
for(i=0;i<n;i++)
{
printf("%d \t",a[i]);
}
}

void main()
{
int count,num[50],i;
clrscr();
printf("\n Enter the number of elements:\n ");
scanf("%d",&count);
printf("\nEnter those elements:\n");
for(i=0;i<count;i++)
{
scanf("%d",&num[i]);
}

printf("\n The unsorted elements are :\n");
print_array(num,count);
quick_sort(num,0,count-1);

```

```
printf("\n\n The sorted elements in ascending order are :\n\n");  
print_array(num,count);  
getch();  
}
```

OUTPUT

Enter the number of elements

5

Enter those elements:

67 34 12 54 89

The unsorted elements are

67 34 12 54 89

The sorted elements in ascending order are

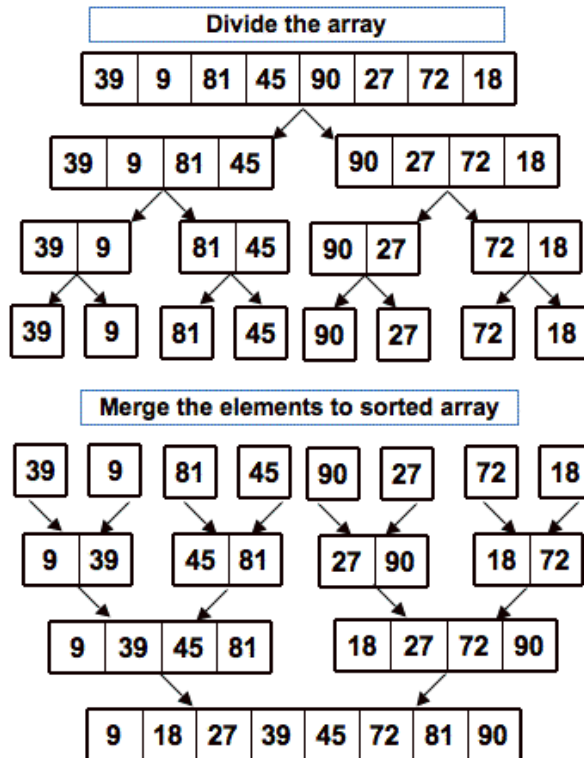
12 34 54 67 89

1.4.5. Merge Sort

This sorting method also follows the technique of divide-and-conquer. The technique works on a principle where a given set of inputs are split into distinct subsets and the required method is applied on each subset separately and then the sorted subsets are combined into one set. This is normally achieved through a recursive practice.

The technique of merge sort works as follows: Given a sequence of "n" elements the idea is to split them into two sets. Each set is individually sorted and the resulting sequence is then combined to produce a single sorted sequence of "n" elements. If each subset is big in size then the subset is further recursively divided into smaller subsets until each subset is small enough to be solved independently without splitting. The sorted subsets are then combined to obtain a single solution to the entire problem.

The algorithm of merge sort may have the following steps with example:



Algorithm's Steps:

Step 1: If the list is of length 0 or 1, then it is already sorted

Step 2: Divide the Unsorted list into two sublist of about half the size

Step 3: Sort each sublist recursively by re-applying merge-sort

Step 4: Merge the two sublists back into one sorted list.

Implementational Algorithm : Merge_sort (A,low,high) Given an array **A** of **high-low + 1** elements, this procedure sorts the elements in the ascending order using the method described above. The variables **low** and **high** are used to identify the positions of the first and last elements in each partition.

Step 1: If (low < high)

Step 2: mid = (low + high) / 2

Step 3: Call Merge_sort (A, low, mid)

Step 4: Call Merge_sort (A, mid + 1, high)

Step 5: Call Merge(A, low, mid, high)

[EndIf]

Step 6: Exit

The algorithm of merging may have the following steps

Implementational Algorithm: Merge(a,low,mid,high) Given an array **A** of **high-low + 1** elements, this procedure sorts the elements in the ascending order using the method described above. The variables **low**, **mid** and **high** is used to identify the positions of the elements in each partition. The first partition is from the position low to the position mid and the next partition is from the position mid + 1 to the position high.

Step 1: $I = \text{low}$

Step 2: $J = \text{mid} + 1$

Step 3: $K = \text{low}$

Step 4: While ($I \leq \text{mid}$) and ($J \leq \text{high}$) Do

Step 5: If ($a[I] < a[J]$)

Step 6: $c[K] = a[I]$

$K = K + 1$

$I = I + 1$

Else

Step 9: $c[K] = a[J]$

Step 10: $K = K + 1$

Step 11: $J = J + 1$

[EndIf]

[End of Step 4 While loop]	
Step 12:	While ($I \leq \text{mid}$) Do
Step 13:	$c[K] = a[I]$
Step 14:	$K = K + 1$
Step 15:	$I = I + 1$
[End of Step 12 While loop]	
Step 16:	While ($J \leq \text{high}$)
Step 17:	$c[K] = a[J]$
Step 18:	$K = K + 1$
Step 19:	$J = J + 1$
[End of Step 16 While loop]	
Step 20:	For $I = \text{low}$ to high
Step 21:	$a[I] = c[I]$
[End of Step 20 For loop]	
Step 22:	Exit

1.4.5.1. Analysis of Merge sort

The following assumptions are also made during the analysis.

1. The elements of the array are distinct.
2. The partition element is chosen using a random selection process.
3. The proper position for the pivot element always turns out to be the middle of the sub array.

4. The array size is assumed to be a power of 2, say $n = 2^m$ or $m = \log_2 n$.

In that case there will be approximately n comparisons in the first pass, after which the array is split into two sub arrays each approximately of the size $n / 2$. For each of the sub arrays there are approximately about $n / 2$ comparisons and a total of four sub arrays each of size $n / 4$ are formed. After repeating the process " n " times then there will be " n " sub arrays each of size 1. Thus the total number of comparisons is approximately equal to

$$\begin{aligned} C(n) &= n + 2 * (n / 2) + 4 * (n / 4) + \dots + n * (n / n) \\ &= n + n + n + \dots + n \text{ (m times)} \\ &= O(n * m) \\ &= O(n \log n) \end{aligned}$$

Program 5.7: Program to perform merge sort.

```
# include <stdio.h>
#include <conio.h>
void mergesort(int*, int, int);
void merge (int*, int, int, int);
void main()
{
    int  a[20],n,i;
    clrscr();
    printf("\n Enter the number of elements");
    scanf("%d", &n);
    printf("\nEnter %d elements\n", n);
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("\n The unsorted elements are \n");
```

```

        for(i=0; i<n; i++)
            printf("\t%d", a[i]);
        printf("\n");
    mergesort(a, 0, n-1);
    printf("\n The sorted elements are \n");
    for(i=0; i<n; i++)
        printf("\t%d", a[i]);
        printf("\n");
}

/*    Merge_sort function    */
void mergesort(int a[], int low, int high)
{
    int mid;
    if(low < high)
    {
        Mid = (low + high) / 2;
        mergesort(a, low, mid);
        mergesort(a, mid + 1, high);
        merge(a, low, mid, high);
    }
}

/*    Merge function    */
void merge(int a[], int low, int mid, int high)
{
    int i, j, k, c[20];
    i = low;
    k = low;

```

```

j = mid + 1;
while ((i <= mid) && (j <= high))
{
    if(a[i] < a[j])
    {
        c[k] = a[j];
        k++;
        i++;
    }
    else
    {
        c[k] = a[j];
        k++;
        j++;
    }
}
while( i <= mid);
{
    c[k] = a[i];
    k++;
    i++;
}
while( j <= high);
{
    c[k] = a[j];
    k++;
    j++;
}

```

```

    }
    for (i = low; i <= k-1; i++)
        a[i] = c[i];
}

```

OUTPUT

Enter the number of elements 9

Enter 9 elements

66 22 10 30 41 51 82 77 2

The unsorted elements are

66 22 10 30 41 51 82 77 2

The sorted elements are

2 10 22 30 41 51 66 77 82

SELF ASSESSMENT QUESTIONS

1. What is an array? Explain how a two- dimensional array can be represented in memory.
2. What are the advantages and disadvantages of an array?
3. Write an algorithm to delete and insert an item into or from a linear array.
4. Explain the following operations with examples:
 - i. Traversing
 - ii. Searching
5. Define searching.
6. Explain the process of searching with the characteristics.
7. Explain linear search technique with an example.
8. Write a C program to perform Linear search.

9. Explain binary search technique with an example.
10. Write a C program to perform binary search.
11. What do you mean by sorting
12. Explain Selection sort technique with an example.
13. Write an algorithm to arrange N numbers in descending order using Quick sort method.
14. Write a C program to arrange N numbers in descending order using Merge Sort method.
15. Explain Quick sort technique with an example.
16. Write a C program to arrange N numbers in ascending order using merge sort method.
17. Which is the fastest sorting technique? Justify.

PART 3

LINKED LIST BASED DATA STRUCTURES

3.1 INTRODUCTION

The list usually contains a list of Items arranged in order indicating the items of our requirement. Many problems in computing also involve lists of items. A list is a data type, which contains elements of the same type arranged in a set of continuous memory locations. With lists it is possible for us to perform certain operations such as searching the list, sorting it, printing it, and so on. The structure we have used as the concrete data representation of a list in part two is the array , which is a sequential structure.

Operations such as searching can be very efficiently performed, However, inserting and deleting items cannot be easily performed with an array representation. To insert a new item into a location in a list, we must shift the array elements down to make room for the new item. Similarly, deleting an item from the list requires that we shift up all the array elements following the one to be deleted.

Another drawback with arrays is that they are not very adaptable since the size of the array must be fixed in advance. The size of the array cannot be increased or decreased during the execution of the program. These problems can be overcome by considering dynamic memory allocation and/or with another data structure called - the **linked list**.

Definition
Linked List: Is a List in which the order of the elements is determined by an explicit link field in each element rather than sequential order in memory.

3.2. REPRESENTATIONAL OF LINKED LIST

A linked list represents a linear collection of items, called as nodes. Nodes of a linked list can be scattered about in the memory, they need not necessarily represent a set of consecutive memory locations. Each node in a linked list has two parts:

1. **Information part**, which contains the data for the node.
2. **Link part**, which gives the memory address of the next node in the list.

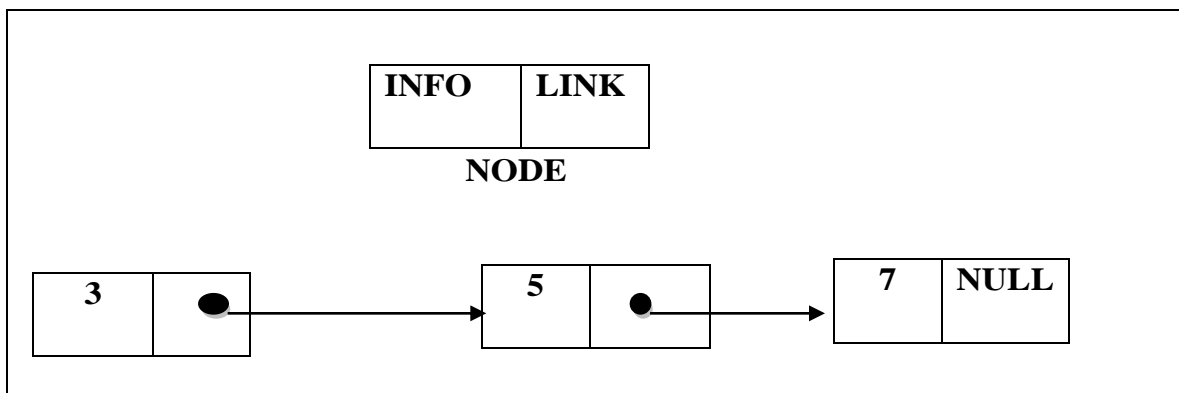


Figure 3.1. Linked List with three nodes

The Figure 3.1 shows an abstract representation of a Linked List. Information part may consist of one or more fields. A linked list thus consists of a series of structures, which are not necessarily contiguous. Each structure contains an information field and a link, which is a pointer to the next structure, which is its successor. An arrow is used in the link member of each node to indicate the location of the next node. The **NULL** in the link member of the last node signifies the end of the list.

In this scope we discuss the following different types of Linked Lists:

1. Singly linked list
2. Circular linked list
3. Doubly linked list

3.3. SINGLY LINKED LIST

A linked list in its simplest form is a collection of nodes that together form a linear ordering. The ordering is determined as in the children's game **"Follow the Leader,"** in this line, each node usually consists of a structure that includes information fields and a link fields.

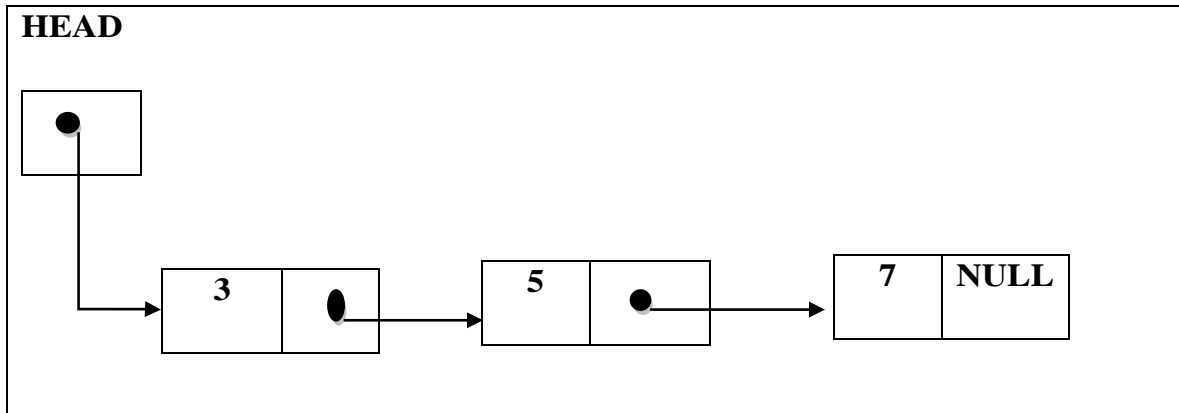


Figure 3.2 Single Linked list representation

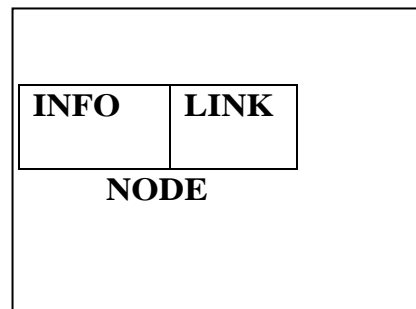
The Figure 3.2 shows a chaining list of numbers, which is represented in the form of a Linked List. **HEAD** is a pointer, which stores the address of the first node in the linked list. Link field of the last node contains the **NULL**, which indicates it does not hold any address. The first and last node of a linked list usually is called the **HEAD** and **TAIL** of the list, respectively. We identify the tail as the node having a **NULL** next reference, which indicates the termination of the list. A linked list defined in this way is known as a **singly linked list**.

Definition of a **NODE** of a Linked list in C

The Single Linked List it can be defined as follows:

```

struct Node
{
    data type    Information;
    struct Node  *Link;
};
  
```



Once the structure of a node is defined we may want to allocate memory space for the nodes of the linked list.

Memory space is always allocated with the help of a declaration statement. For example If we want to create a linked list that contains three nodes called **Node 1**, **Node 2** and **Node 3** the declaration may be as follows.

```
struct Node * Node1, *Node2, *Node3;
```

Program 3.1: A program to create a linked list which contains three nodes.

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

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

```
Struct Node
```

```
{
```

```
    int info;
```

```
    struct Node link;
```

```
};
```

```
void main()
```

```
{
```

```
    struct Node *Node1, *Node2, *Node3;
```

```
    Node1 -> info = 123;
```

```
    Node1 -> link = Node2;
```

```
    Node2 -> info = 456;
```

```
    Node2 -> link = Node3;
```

```
    Node3 -> info = 789;
```

```
    Node1 -> link = NULL;
```

```
    Printf("\n Linked list is \n");
```

```
    Printf("%d → %d → %d → NULL", Node1 -> info, Node2 -> info, Node3 -> info);
```

```
}
```

OUTPUT

123 → 456 → 789 → NULL

3.4 DIFFERENT OPERATIONS POSSIBLE ON A LINKED LIST

1. Creating a linked list
2. Traversing a linked list.
3. Inserting an item into a linked list.
4. Deleting an item from the linked list.
5. Searching an item in a linked list.

3.4.1 Creating a Linked List

To create a linked list, which contains nodes, where we do not know how many elements are to be inserted in a linked list. we will have to use a technique in which the list components are dynamically created as they are needed using pointers and dynamic memory allocation function such as **malloc()**. We start accessing the list with a pointer variable that holds the address of the first node in the list. This pointer variable is named as **HEAD**. Every node except the first node can be accessed by using the link member of its previous node.

To create a dynamic linked list, we begin by initializing the header **HEAD** to **NULL**. We then create the first node and then assign its address to the header **HEAD**. We then allocate a second node and store its address in the link part of the first node. We continue this process of allocating a new node and storing the pointer to it into the link part of the previous node until the user decides to stop the process.

Algorithm 3.1: Creating a Linked List

Step 1: [Initialize the header]

HEAD = NULL

Step 2: [Create a new node and store its address in the pointer NewNode]

NewNode = Address of new node

Step 3: [Copy the information for the new node to the information part]

INFO [NewNode] = element

Step 4: [Set the contents of the link part as NULL]

LINK[NewNode] = NULL

Step 5: [Is the new node the first node?]

If answer is yes Then

[Connect the new node to the header]

HEAD = NewNode

Else

(a) [Take the CurrPtr to the last node of the linked list]

CurrPtr = Address of the last node.

(b) [Connect the new node to the link of the last node]

LINK [CurrPtr] = NewNode

(c) Exit

[Endif]

Step 6: Return

In the above algorithm, we used three pointers:

1. **HEAD** is used to hold the address of the first node of the linked list.
2. **NewNode** is used to hold the address of the new node.
3. **CurrPtr**, which is updated to always point to the last node in the linked list.

Program 3.2: Program to create a linked list.

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <conio.h>

struct node

{

    int info;

    struct node *link;

};

typedef struct node NODE;

NODE *HEAD;

void Create_link_list (int);

void display (NODE*);

void main()

{

    int ele;

    char ch;

    clrscr();

    HEAD = NULL;

    printf("\n Linked list creation \n \n");

    do{

        printf("Enter information field of the node \n");

        scanf("%d", &ele);

        create_link_list(ele);
```

```

        printf("Add another node \n");

        fflush();

        scanf("%c", &ch);

    } while (toupper (ch)!='y');

    printf("\n The linked list is \n");

    display (HEAD);

    getch();

}

/*    Function to add a node to the linked list    */

void create_link_list(int ele)

{

    NODE *NewNode, *CurrPtr;

    NewNode = (NODE*) malloc(sizeof(NODE));

    NewNode -> info = ele;

    NewNode -> link = NULL;

    if (HEAD == NULL)

        HEAD = NewNode;

    else

    {

        CurrPtr = HEAD;

        while(CurrPtr -> link != NULL)

            CurrPtr = CurrPtr -> link;

        CurrPtr -> link = NewNode;

    }

}

```

```
/* Function to display linked list element */
```

```
void Display (NODE *CurrPtr)
```

```
{  
    while(CurrPtr != NULL)  
    {  
        printf("%d →", CurrPtr -> info);  
        CurrPtr = CurrPtr -> link;  
    }  
    printf ("NULL\n");  
}
```

OUTPUT

Linked list creation

Enter information field of the node 56

Do you wish to enter one more node? y

Enter information field of the node 89

Do you wish to enter one more node? y

Enter information field of the node 23

Do you wish to enter one more node? n

Linked list elements

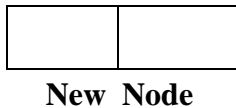
56 → 89→ 23→ NULL

Going through some of the statements, describing in words what is happening and showing the linked list as it appears after the execution of the statement.

HEAD = NULL;

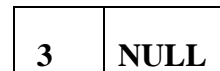
The statement is used to initialize the header to NULL, which indicates that the linked list has not yet been created.

NewNode = (NODE*) malloc (sizeof(NODE));



A dynamic variable of type NODE is created. This statement obtains a piece of memory to store a node and assigns its address to the pointer variable **NewNode**.

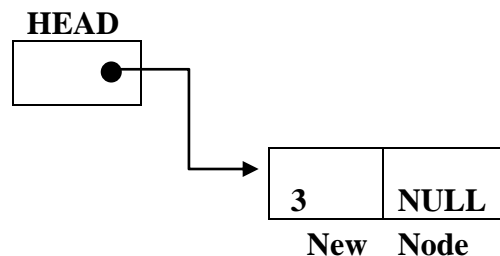
NewNode -> info = ele;



NewNode -> link = NULL;

Using the above statements, the content of the new node is recorded and its link part is set to be Null.

HEAD = NewNode;

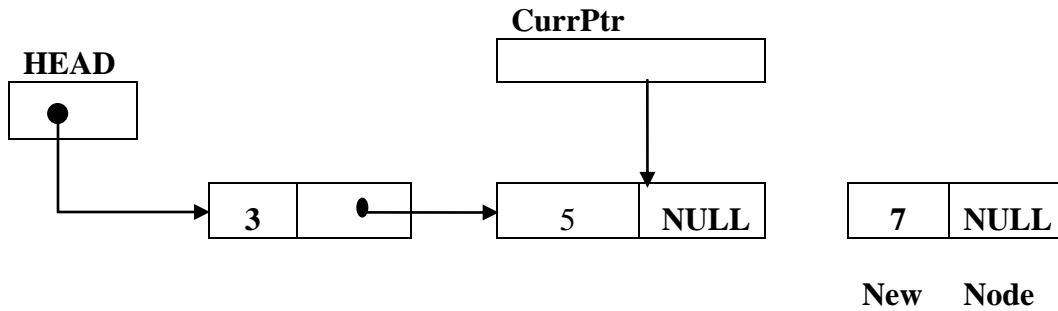


Connects the first node to the head pointer.

CurrPtr = HEAD;

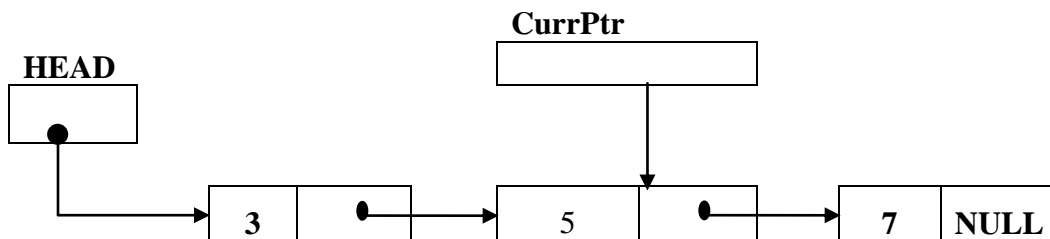
While (CurrPtr -> link != NULL)

CurrPtr = CurrPtr -> link;



The looping statement takes the CurrPtr from the first node to the last node of the linked list.

CurrPtr -> link = NewNode;



The pointer to the new node is copied into the link member of CurrPtr, which connects the new node to the existing Linked List.

6.4.2. Traversing a linked list

Traversing is the process of visiting each node of the linked list exactly once to perform some operation. Our aim is to traverse the list starting from the first node to the end of the list. Let LIST be a linked list with nodes containing two fields INFO and LINK. HEAD is a pointer variable, which contains the address of the first node in the linked list. CurrPtr is a pointer variable, which points to the node currently being processed. The fields of the node is referenced by INFO[CurrPtr] indicates information field of the node which has the address stored in CurrPtr variable. LINK[CurrPtr) is the link field of the node which has the address stored in CurrPtr variable. The statement CurrPtr = LINK[CurrPtr] Moves the pointer to the next node in the list as illustrated in the below algorithm.

Algorithm 6.2: *This algorithm traverses a linked list by applying a process to each element of the list. The variable **CurrPtr** points to the node currently being processed.*

Step 1: [Initialize pointer variable CurrPtr]

CurrPtr = HEAD

Step 2: [Perform the traversing Operation]

While CurrPtr != NULL Do

Step 3: Apply PROCESS to INFO[CurrPtr]

Step 4: [Move pointer to next node]

CurrPtr = LINK [CurrPtr]

[End of While Operation]

Step 5: Exit

Example 3.1: Printing all the elements of the linked list is an example of traversing a linked list. Here 3,5,7 is printed.

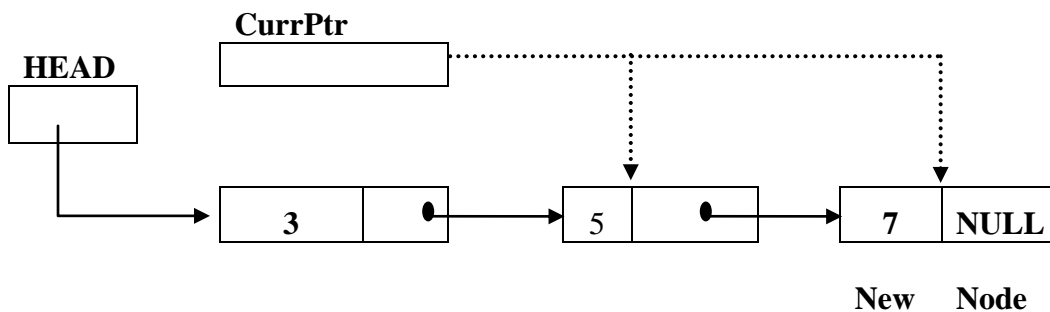


Figure 3.3 Traversing a linked list

Function to perform traversing

```
Void Display (NODE * CurrPtr)
{
    While (CurrPtr != NULL)
    {
        Printf (“ %d\t”, CurrPtr -> info);
        CurrPtr = CurrPtr -> link;
    }
    Printf(“NULL \n”);
}
```

To print the components of a linked list, we need to access the nodes one at a time.

Program 3.3: Program to illustrate traversing operation in which we find the largest element in a linked list.

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

struct node
{
    int info;
    struct node *link;
};

typedef struct node NODE;
NODE *HEAD;
void Create_link_list (int);
```

```

void Display (NODE*);
void Largest (NODE*);
void main()
{
    int ele;
    char ch;
    clrscr();
    HEAD = NULL;
    printf("\n Linked list creation \n \n");
    do{
        printf("Enter information field of the node \n");
        scanf("%d", &ele);
        create_link_list(ele);
        printf("Add another node \n");
        flushall();
        scanf("%c", &ch);
    }while (toupper (ch)!='y');
    printf("\n The linked list is \n");
    display (HEAD);
    printf("\n Largest element in the list is %d", Largest (HEAD));
    getch();
}

/*    Function to add a node to the linked list    */
void Create_link_list(int ele)
{
    NODE *NewNode, *CurrPtr;
    NewNode = (NODE*) malloc(sizeof(NODE));

```

```

    NewNode -> info = ele;

    NewNode -> link = NULL;

    if (HEAD == NULL)

        HEAD = NewNode;

    else

    {

        CurrPtr = HEAD;

        while(CurrPtr -> link != NULL)

            CurrPtr = CurrPtr -> link;

        CurrPtr -> link = NewNode;

    }

}

/* Function to display linked list element */
void Display (NODE *CurrPtr)
{
    while(CurrPtr != NULL)
    {
        printf("%d→", CurrPtr -> info);

        CurrPtr = CurrPtr -> link;

    }

    printf ("NULL\n");
}

/* Function to find the largest element in a linked list */
int Largest (NODE * CurrPtr)
{
    int big = CurrPtr -> info    /* First node information */

    CurrPtr = CurrPtr -> link; /* Move pointer to next node */
}

```

```

while (CurrPtr != NULL)
{
    if (big < CurrPtr -> info) big = CurrPtr -> info;
    CurrPtr = CurrPtr -> link;
}
return big;
}

```

OUTPUT

Linked list creation

Enter information field of the node 56

Do you wish to enter one more node? y

Enter information field of the node 89

Do you wish to enter one more node? y

Enter information field of the node 23

Do you wish to enter one more node? n

Linked list elements

56 → 89 → 23 → NULL

Largest element in the list is 89

3.4.3 Inserting an item into a linked list

The ordering may be in increasing order or decreasing order of information field. Before inserting a new node, first we have to decide where to insert the new Node. There are different Options in at which the insertions can take place:

1. Insert a Node at the beginning of the linked list.
2. Insert a Node at the end of the linked list.
3. Insert a Node at a given position in the list

3.4.3.1 Insert a Node at the beginning of the linked list

Let the **HEAD** pointer hold the address of the first node of the linked list. The following algorithm will create a new node and insert this node at the beginning of the linked list.

Algorithm 3.3: Let **LIST** be a list stored in memory, **HEAD** gives the address of the first node. This algorithm inserts **ITEM** into the list as the first node.

Step 1: [Create a new node and store its address in the pointer NewNode]

NewNode = Address of new node

Step 2: [Copy the information for the new node to the information part]

INFO[NewNode] = element

Step 3: [Set the contents of the link part to contain the address of the first node]

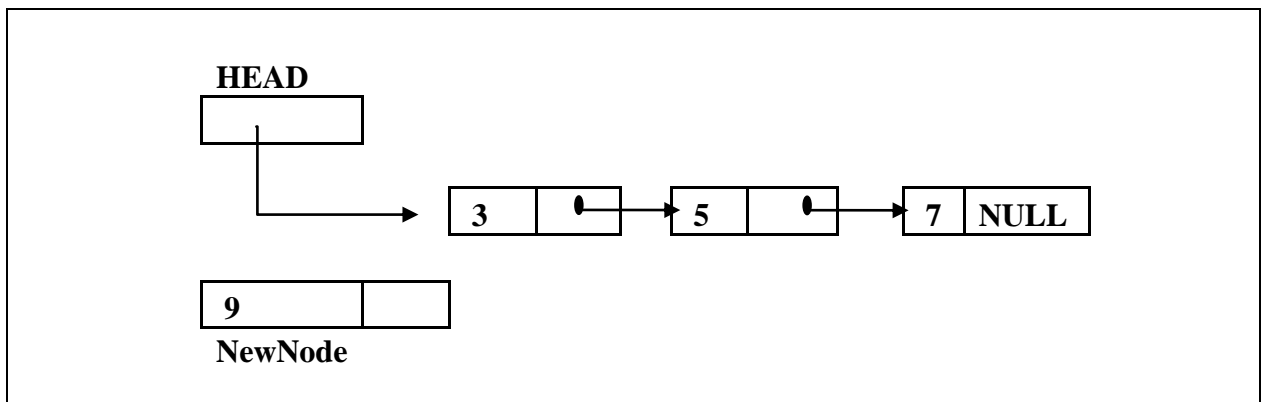
LINK [NewNode] = HEAD

Step 4: [Make the new node as the first node of the linked list]

HEAD = NewNode

Step 5: Exit

The schematic diagram of Step 2, Step 3 and Step 4 are shown below.



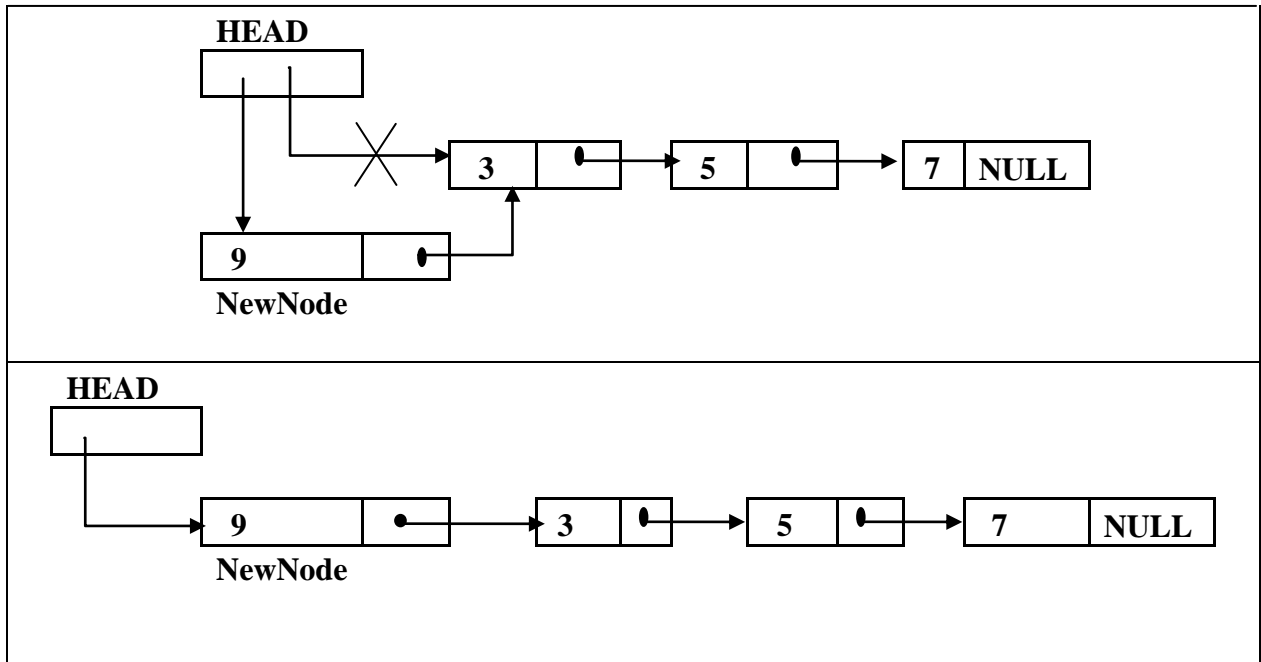


Figure 3.5 Insert an ITEM values 9 at the beginning of the list

Function to insert a node at the beginning of the linked list

```

Void Insert_Beginning (int ITEM)
{
    NODE * NewNode;

    NewNode = (NODE *) malloc (sizeof(NODE));

    NewNode -> info = ITEM;

    NewNode -> link = HEAD;

    HEAD = NewNode;
}

```


3.4.3.2 Insert a Node at the End of the linked list

In some case we might be requested to write an application that appends an item at the end of the list. To append an item to a linked list, we loop through the list until find the element whose link member points to NULL. The following algorithm is used to insert a node into a linked list as a last node.

Algorithm 3.4: *Let **LIST** be a list stored in memory, **HEAD** gives the address of the first node. This algorithm inserts **ITEM** into the list as the last node. **CurrPTR** is a pointer, which is pointing to the current node of the list.*

Step 1: [Create a new node and store its address in the pointer NewNode]

 NewNode = Address of new node

Step 2: [Copy the information for the new node to the information part]

 INFO[NewNode] = element

Step 3: [Set the contents of the link part to NULL]

 LINK [NewNode] = NULL

Step 4: [Initialize HEAD value to CURRPTR]

 CurrPtr = HEAD;

Step 5: [Find the last node of the list]

 While LINK(CurrPtr) != NULL Do

Step 6: [Move CURRPTR to next node]

 CurrPtr = LINK [CurrPtr]

 [WhileEnd]

Step 7: [Connect the new node to the link of the last node]

 LINK[CurrPtr] = NewNode

Step 8: Exit

The schematic diagram of Step 6 and 7 are shown in Figure below:

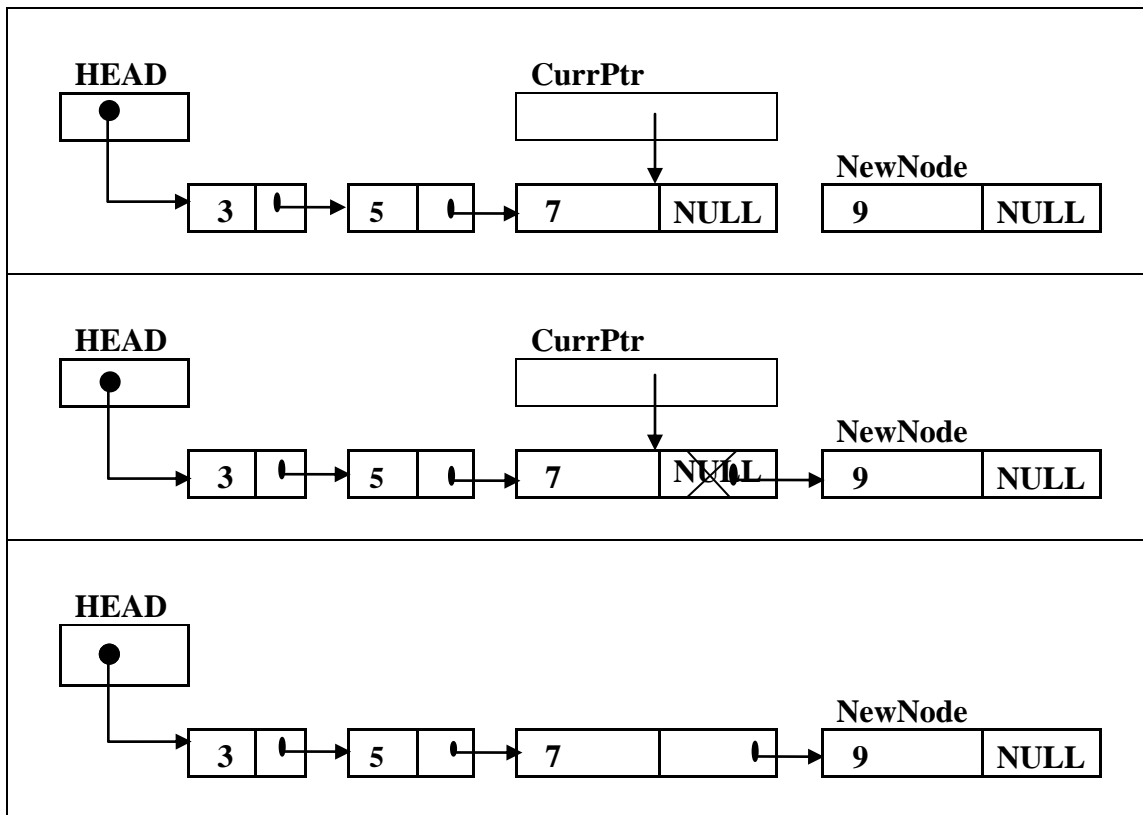


Figure 3.6 Insert an ITEM value 9 at the end of the list

Function to insert a node at the end of the linked list

```
Void Insert_end(int ITEM)
{
    NODE * NewNode, *CurrPtr;

    NewNode = (NODE *) malloc (sizeof(NODE));

    NewNode -> info = ITEM;

    NewNode -> link = NULL;

    CurrPtr = HEAD;

    While (CurrPtr -> link != NULL)
```

```

    CurrPtr = CurrPtr -> link;

    CurrPtr -> link = NewNode;

}

```

3.4.3.3 Insert a Node at a given position

Suppose we are given the ITEM, which has to be placed at a given position in the linked list. The following is an algorithm, which inserts ITEM into LIST so that ITEM is placed at the required position.

Algorithm 6.5: Let *LIST* be a list stored in memory, *HEAD* gives the address of the first node. This algorithm inserts *ITEM* into the list at the position *POS*. *CurrPtr* is a pointer, which is pointing to the current node of the list.

Step 1: [Create a new node and store its address in the pointer NewNode]

NewNode = Address of new node

Step 2: [Copy the information for the new node to the information part]

INFO[NewNode] = element

Step 3: [Initialize HEAD value to CURRPTR]

CurrPtr = HEAD

Step 4: [Move to the required position]

For I = 1 to POS – 1 and CurrPtr != NULL Do

Step 5: [Move CURRPTR to next node]

CurrPtr = LINK [CurrPtr]

[ForEnd]

Step 6: [Check whether the position is found or not]

If(CurrPtr = NULL)

Printf " Position Out Of Range"

Exit

Step 7: Else

[Insert the new node at the required position]

LINK [NewNode] = LINK [CurrPtr]

LINK[CurrPtr] = NewNode

[EndIf]

Step 8: Exit

The schematic diagram of Step 4 and Step 7 are shown in Figure

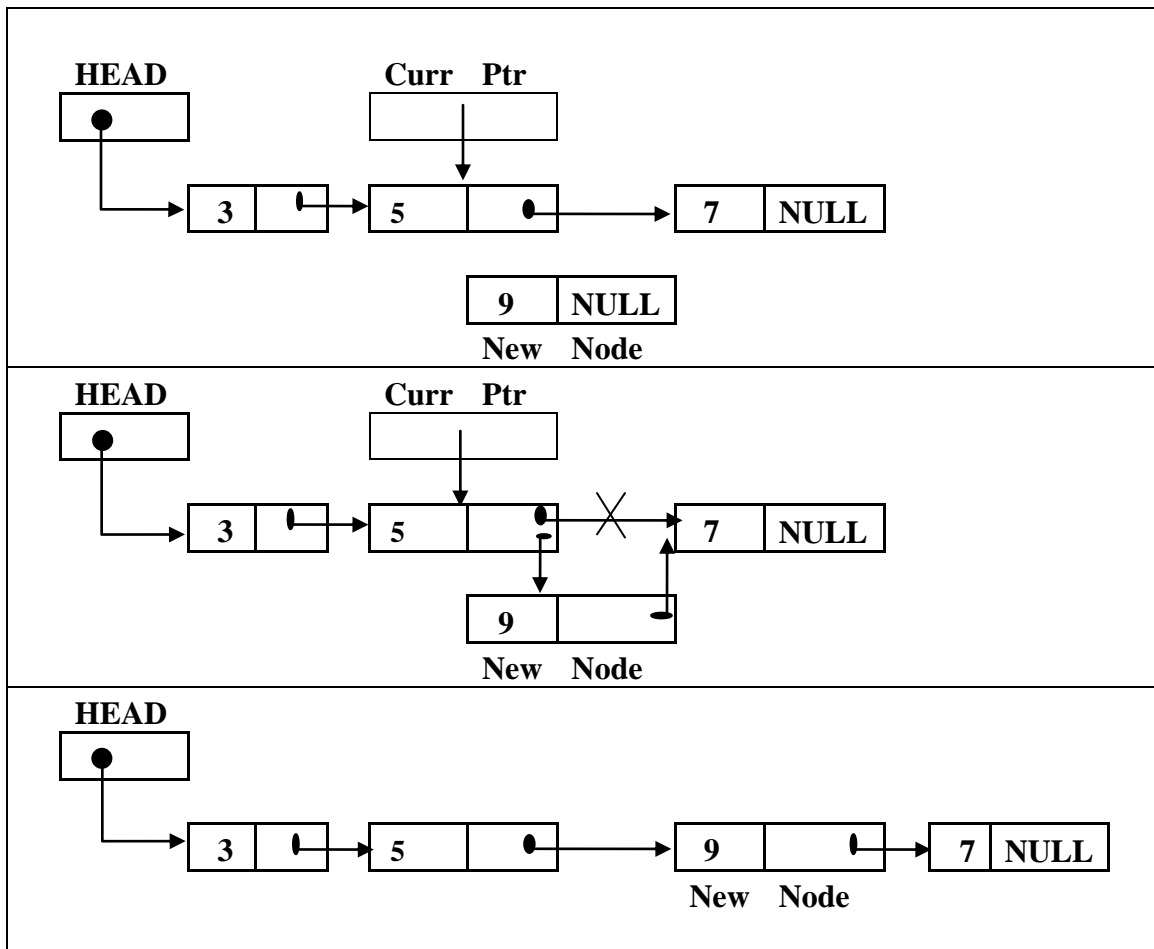


Figure 3.7 Insert an ITEM value 9 at the third position of the list.

Function to insert a node at a given position

```
Void Insert_Position (int ITEM, int ITEM)
{
    NODE * NewNode, *CurrPtr;

    int i;

    NewNode = (NODE *) malloc (sizeof(NODE));

    NewNode -> info = ITEM;

    CurrPtr = HEAD;

    for(i = 1; i < POS - 1 && CurrPtr != NULL; i++)

    CurrPtr = CurrPtr -> link;

    if(CurrPtr == NULL)
    {
        printf ("Position Out Of Range \n");
        exit(0);
    }

    NewNode -> link = CurrPtr -> link;

    CurrPtr -> link = NewNode;
}
```

Program 3.4: To create a linked list and insert an element into it.

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

struct node
```

```

{
    int info;
    struct node *link;
};

typedef struct node NODE;
NODE *HEAD;

void Create_link_list (int);
void Display (NODE*);
void Insert_Beginning (int);
void Insert_End (int);
void Insert_Position (int,int);
void main()
{
    int ele, pos;
    char ch;
    clrscr();
    HEAD = NULL;
    printf("\n Linked list creation \n \n");
    do{
        printf("Enter information field of the node \n");
        scanf("%d", &ele);
        create_link_list(ele);
        printf("Add another node \n");
        fflush();
        scanf("%c", &ch);
    }while (toupper (ch)=='y');
    printf("\n The linked list is \n");

```

```

display (HEAD);

printf("\n Insert Operation\n\n");

do
{
    printf("\n Choose Position To Insert: \n");
    printf("\n 1. AT THE BEGINNING");
    printf("\n 2. AT THE END");
    printf("\n 3. AT A GIVEN POSITION");
    printf("\n 4. EXIT");
    printf("\n Enter your choice: ");
    scanf("%d",&ch);

    switch(ch)
    {
    case 1: {
        printf("Enter element to add: ");
        scanf("%d", &ele);
        Insert_Beginning (ele);
        printf("\n The linked list is \n");
        Display (HEAD);
        break;
    }

    Case 2: {
        printf("Enter element to add: ");
        scanf("%d", &ele);
        Insert_End (ele);
        printf("\n The linked list is \n");
        Display (HEAD);
    }
    }
}

```

```

        break;
    }
    Case 3: {
        printf("Enter element to add: ");
        scanf("%d", &ele);
        printf("Enter position to add: ");
        scanf("%d", &pos);
        Insert_Position (ele, pos);
        printf("\n The linked list is \n");
        Display (HEAD);
        break;
    }
}
}while (ch != 4);
getch();
}
/*    Function to add a node to the linked list    */
void Create_link_list(int ele)
{
    NODE *NewNode, *CurrPtr;
    NewNode = (NODE*) malloc(sizeof(NODE));
    NewNode -> info = ele;
    NewNode -> link = NULL;
    if (HEAD == NULL)
        HEAD = NewNode;
    else
    {

```



```

        CurrPtr = HEAD;

        while(CurrPtr -> link != NULL)

            CurrPtr = CurrPtr -> link;

        CurrPtr -> link = NewNode;

    }

}

/* Function to display linked list elements */
void Display (NODE *CurrPtr)
{
    while(CurrPtr != NULL)
    {
        printf("%d →", CurrPtr -> info);

        CurrPtr = CurrPtr -> link;
    }

    printf ("NULL\n");
}

/* Function to insert a node at the beginning of the linked list */
void Insert_Beginning (int ITEM)
{
    NODE *NewNode;

    NewNode = (NODE*) malloc(sizeof(NODE));

    NewNode -> info = ITEM;

    NewNode -> link = HEAD;

    HEAD = NewNode;
}

/* Function to insert a node at the end of the linked list */
void Insert_End (int ITEM)

```

```

{
    NODE *NewNode, *CurrPtr;

    NewNode = (NODE*) malloc(sizeof(NODE));

    NewNode -> info = ITEM;

    NewNode -> link = NULL;

    CurrPtr = HEAD;

    While(CurrPtr -> link != NULL)

        CurrPtr = CurrPtr -> link;

    CurrPtr -> link = NewNode;
} /* Function to insert a node at a given position in a linked list */

void Insert_Position(int ITEM, int POS)
{
    NODE *NewNode, *CurrPtr;

    int i;

    NewNode = (NODE*) malloc(sizeof(NODE));

    NewNode -> info = ITEM;

    CurrPtr = HEAD;

    for(i = 1; i < POS - 1; && CurrPtr != NULL; i++)

        CurrPtr = CurrPtr -> link;

    if(CurrPtr == NULL)
    {
        printf("Position Out Of Range\n");

        exit(0);
    }

    NewNode -> link = CurrPtr -> link;

    CurrPtr -> link = NewNode;
}

```

OUTPUT

Linked list creation

Enter information field of the node 56

Do you wish to enter one more node? y

Enter information field of the node 89

Do you wish to enter one more node? y

Enter information field of the node 23

Do you wish to enter one more node? n

Linked list elements

56 → 89 → 23 → NULL

Insert Operation

Choose Position To Insert:

1. AT THE BEGINNING
2. AT THE END
3. AT A GIVEN POSITION
4. EXIT

Enter your choice: 1

Enter element to add: 15

The linked list is

15 → 56 → 89 → 23 → NULL

Choose Position To Insert:

1. AT THE BEGINNING
2. AT THE END
3. AT A GIVEN POSITION
4. EXIT

Enter your choice: 2

Enter the element to add: 19

15 → 56 → 89 → 23 → 19 → NULL

Choose Position To Insert:

1. AT THE BEGINNING
2. AT THE END
3. AT A GIVEN POSITION
4. EXIT

Enter your choice: 3

Enter the element to add: 65

Enter Position to add: 2

The linked list is

15 → 65 → 56 → 89 → 23 → 19 → NULL

3.4.4 Deleting an item from the linked list

To delete an existing node from a linked list, we have to loop through the nodes until we find the node we want to delete. We should follow the steps below to delete a node from a linked list

1. If the linked list is empty, then deletion is not possible and this condition is called as **underflow** condition.
2. To delete a particular node, we have to loop through the nodes until we find the node we want to delete.

The deletion operation is classified into following types:

1. Deletion of the first node.
2. Deletion of the last node.
3. Deleting of the node at a given position.

3.4.4.1 Deletion of first node

To delete the first node, we just change the **HEAD** pointer to point to the second node or to contain NULL if we are deleting the only node in a one-node list. Suppose we have a linked list as shown in Figure and want to delete first node.

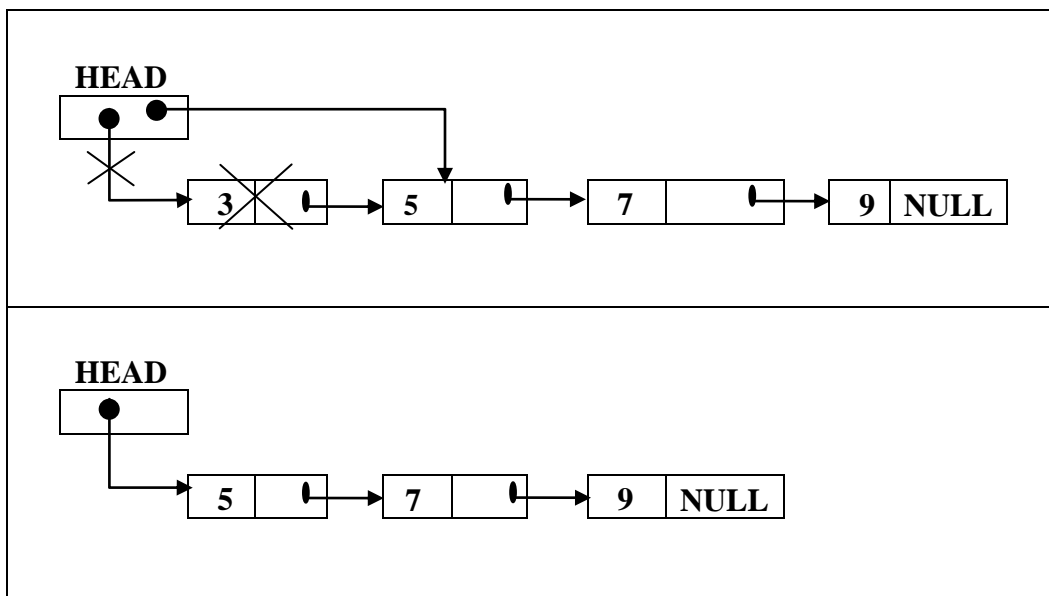


Figure 3.8 Delete last node from the linked list

The following algorithm deletes the first node from the linked list and insert-deleted node into availability list.

Algorithm 3.6: Let *LIST* be a list stored in memory. It deletes first node from the linked list. *HEAD* gives the address of the first node.

Step 1: [List Empty?]

 If HEAD = NULL

 Display “Under flow”

 Exit

 [EndIf]

Step 2: [Initialize pointer variable CurrPtr]

 CurrPtr = HEAD

Step 3: [Delete first node]

 HEAD = LINK [CurrPtr]

Step 4: [Return deleted node to the AVAIL list]

 FREE [CurrPtr]

Step 5: Exit

Function to delete a node at the beginning of the linked list

```
int Delete_Beginning (int ITEM)
```

```
{
```

```
    NODE *CurrPtr;
```

```
    int ele;
```

```
    if (HEAD == NULL)
```

```

{
    printf("\n Deletion not possible, the List is Empty");
    exit(0);
}

CurrPtr = HEAD;

ele = CurrPtr -> info;

HEAD = CurrPtr -> link;

free (CurrPtr);

return (ele);
}

```

3.4.4.2 Deletion of last node

To delete a last node of a linked list, we have to scan the list from the beginning and find the address of the next to last node and assign NULL to link field of the next to last node. The following algorithm deletes the last node from the linked list.

Algorithm 6.7: *Let **LIST** be a list stored in memory. It deletes first node from the linked list. **HEAD** gives the address of the first node. **CurrPtr** is a pointer, which is pointing the previous node of the current node.*

Step 1: [List Empty?]

If HEAD = NULL Then

Display “Under flow”

Exit

[EndIf]

Step 2: [Initialize pointer variable CurrPtr]

CurrPtr = HEAD

Step 3: [Traverse linked list and take the pointer to next node]

While (LINK (CurrPtr) != NULL) Do

Step 4: PrevPtr = CurrPtr

Step 5: CurrPtr = LINK (CurrPtr]

[End of While]

Step 6: LINK[PrevPtr] = NULL

Step 7: [Return deleted node to the AVAIL list]

FREE [CurrPtr]

Step 8: Exit

Function to delete the last node of the linked list

```
int Delete_End(int ITEM)
{
    NODE *CurrPtr, *PrevPtr;
    int ele;
    if (HEAD == NULL)
    {
        printf("\n Deletion not possible");
        exit(0);
    }
}
```



```

CurrPtr = HEAD;

while (CurrPtr -> link != NULL)
{
    PrevPtr = CurrPtr;
    CurrPtr = CurrPtr -> link;
}

ele = CurrPtr -> info;
PrevPtr -> link = NULL;
free (CurrPtr);
return(ele);
}

```

Following is the data structure after the execution of the function.

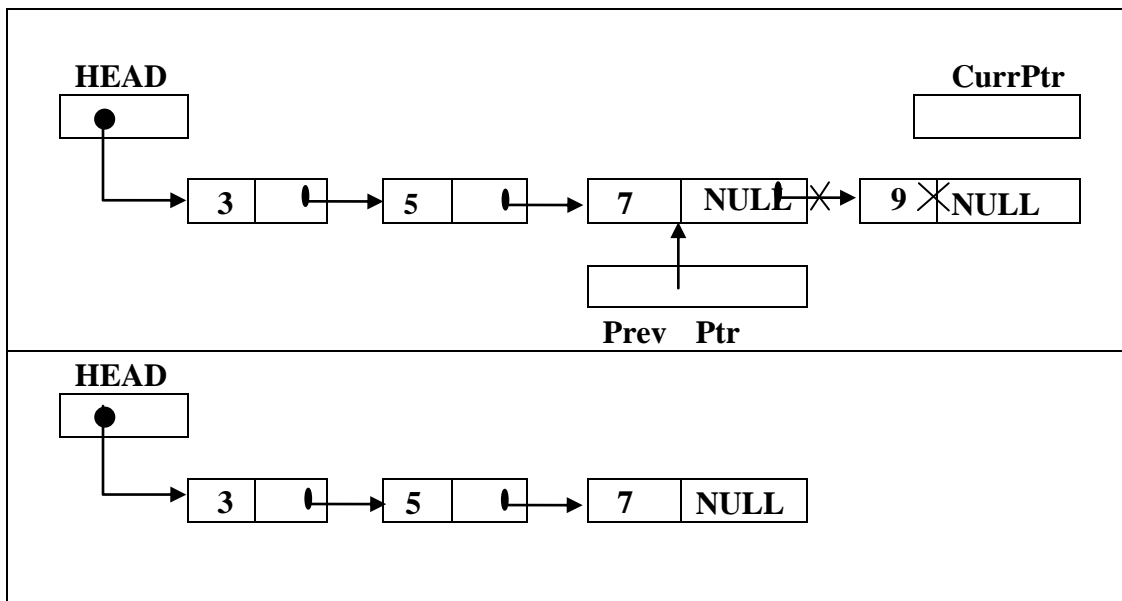


Figure 3.9 Delete last node from the linked list

3.4.4.3 Deleting the node at a given position

The function for deleting a node at a given position involves the following steps.

1. Find the location of the node with **CurrPtr**, and location of previous node with **prevptr**.
2. Delete the node (This is nothing but deleting the node following a given node).

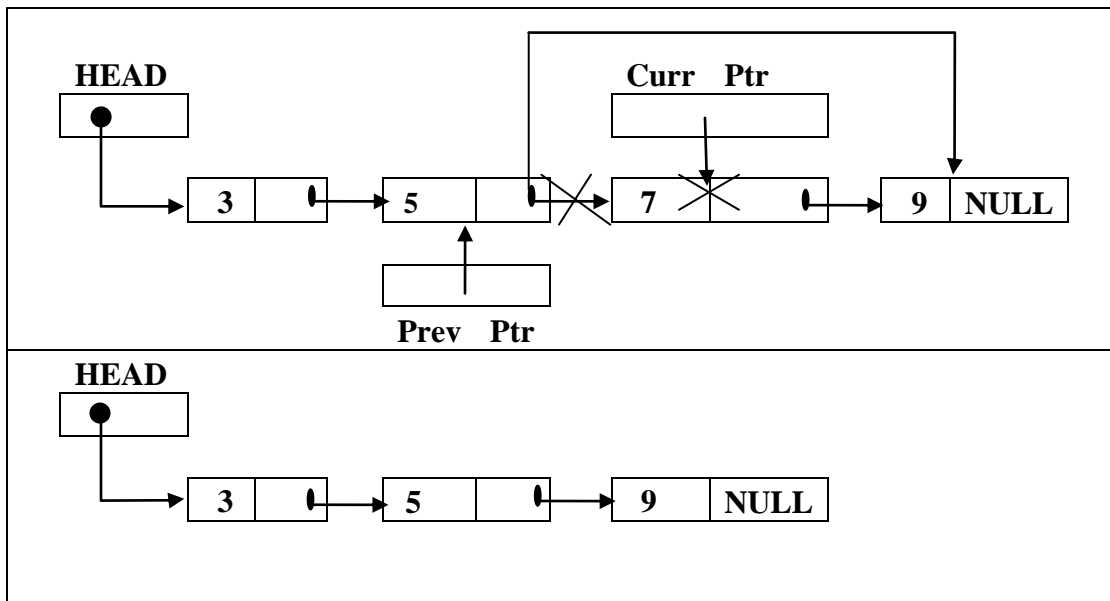


Figure 6.10 Delete the node at the third position of the list.

Algorithm 6.8: Let **LIST** be a list stored in memory. It deletes first node from the linked list. **HEAD** gives the address of the first node. **POS** is the position of the node, which is to be deleted. **CurrPtr** is a pointer, which is pointing to the node at a given position of the list. **PrevPtr** is a pointer, which is pointing the previous node of the current node.

Step 1: [List Empty?]

If **HEAD** = **NULL** Then

Display "Under flow"

Exit

[EndIf]

Step 2: [Initialize pointer variables CurrPtr]

CurrPtr = HEAD

Step 3: [Traverse linked list and take the pointer to the desired position]

For I = 1 to POS and CurrPtr != NULL Do

Step 4: PrevPtr = CurrPtr

Step 5: CurrPtr = LINK (CurrPtr)

[End of For]

Step 6: If CurrPtr = NULL Then

Display “Position out of range”

Exit

[EndIf]

Step 7: [Delete the node]

LINK (PrevPtr) = LINK (CurrPtr)

Step 8: [Return deleted node to the AVAIL list]

FREE (CurrPtr)

Step 9: Exit

Function to delete a node at a given position of the linked list

```

int Delete_Position(int POS)

{

    NODE *CurrPtr, *PrevPtr;

    int ele, i;

    if (HEAD == NULL)

    {

        printf("\n Deletion not possible");

        exit(0);

    }

    CurrPtr = HEAD;

    for (i = 1; i < POS && CurrPtr != NULL; i++)

    {

        PrevPtr = CurrPtr;

        CurrPtr = CurrPtr -> link;

    }

    if (CurrPtr == NULL)

    {

        printf("\n Position out of range");

        exit (0);

    }

    ele = CurrPtr -> info;

```

```
PrevPtr -> link = CurrPtr -> link;

free(CurrPtr);

return (ele);

}
```

Program 3.5: To create a linked list and delete an element from it.

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <conio.h>

struct node

{

    int info;

    struct node *link;

};

typedef struct node NODE;

NODE *HEAD;

void Create_link_list (int);

void Display (NODE*);

void Delete_Beginning ( );

void Delete_End ( );
```

```

void Delete_Position (int);

void main()

{

    int ele, pos;

    char ch;

    clrscr();

    HEAD = NULL;

    printf("\n Linked list creation \n \n");

    do{

        printf("Enter information field of the node \n");

        scanf("%d", &ele);

        Create_link_list(ele);

        printf("Add another node \n");

        flushall();

        scanf("%c", &ch);

    } while (toupper (ch)!='Y');

    printf("\n The linked list is \n");

    display (HEAD);

    printf("\n Deletion Operation\n\n");

    do

    {

```

```

printf("\n Choose Position To Delete: \n");

printf("\n 1. AT THE BEGINNING");

printf("\n 2. AT THE END");

printf("\n 3. AT A GIVEN POSITION");

printf("\n 4. EXIT");

printf("\n Enter your choice: ");

scanf("%d",&ch);

switch(ch)

{

case 1: {

        ele = Delete_Beginning ( );

        printf("Deleted element is: %d", ele);

        printf("\n The linked list is \n");

        Display (HEAD);

        break;

    }

Case 2: {

        ele = Delete_End ( );

        printf("Deleted element is: %d", ele);

        printf("\n The linked list is \n");

        Display (HEAD);

```

```

        break;
    }

    Case 3: {

        printf("\n Enter position to delete:");

        scanf("%d", &pos);

        ele = Delete_Position (pos);

        printf("Deleted element is: %d", ele);

        printf("\n The linked list is \n");

        Display (HEAD);

        break;

    }

}

}while (ch != 4);

getch();

}

/*    Function to add a node to the linked list    */

void Create_link_list(int ele)

{

    NODE *NewNode, *CurrPtr;

    NewNode = (NODE*) malloc(sizeof(NODE));

    NewNode -> info = ele;

```



```

    NewNode -> link = NULL;

    if (HEAD == NULL)

        HEAD = NewNode;

    else

    {

        CurrPtr = HEAD;

        while(CurrPtr -> link != NULL)

            CurrPtr = CurrPtr -> link;

        CurrPtr -> link = NewNode;

    }

}

/* Function to display linked list elements */

void Display (NODE *CurrPtr)

{

    while(CurrPtr != NULL)

    {

        printf("%d →", CurrPtr -> info);

        CurrPtr = CurrPtr -> link;

    }

    printf ("NULL\n");

}

```

/* Function to delete a node at the beginning of the linked list */

int Delete_Beginning ()

{

 NODE *CurrPtr;

 int ele;

 if(HEAD == NULL)

 {

 printf("\n Deletion not possible");

 exit(0);

 }

 CurrPtr = HEAD;

 ele = CurrPtr -> info;

 HEAD = CurrPtr -> link;

 free (CurrPtr);

 return(ele);

}

/* Function to delete a node at the end of the linked list */

int Delete_End ()

{

 NODE *CurrPtr, *PrevPtr;

 int ele;

```

if( HEAD == NULL)

{

    printf("\n Deletion not possible");

    exit(0);

}

CurrPtr = HEAD;

while (CurrPtr -> link != NULL)

{

    PrevPtr = CurrPtr;

    CurrPtr = CurrPtr -> link;

}

ele = CurrPtr -> info;

PrevPtr -> link = NULL;

free (CurrPtr);

return(ele);

}

/* Function to delete a node at a given position in a linked list */

int Delete_Position (int POS)

{

    NODE *CurrPtr, *PrevPtr;

    int ele, i;

```

```

if( HEAD == NULL)

{

    printf("\n Deletion not possible");

    exit(0);

}

CurrPtr = HEAD;

for (i = 1; I < POS && CurrPtr != NULL; i++)

{

    PrevPtr = CurrPtr;

    CurrPtr = CurrPtr -> link;

}

If( CurrPtr == NULL)

{

    Printf("\n Position out of range");

    Exit(0);

}

ele = CurrPtr -> info;

PrevPtr -> link = CurrPtr -> link;

free (CurrPtr);

return(ele);

}

```

OUTPUT

Linked list creation

Enter information field of the node 56

Do you wish to enter one more node? y

Enter information field of the node 89

Do you wish to enter one more node? y

Enter information field of the node 23

Do you wish to enter one more node? y

Enter information field of the node 35

Do you wish to enter one more node? n

Linked list elements

56 → 89 → 23 → 35 → NULL

Deletion Operation

Choose Position To Delete:

1. AT THE BEGINNING
2. AT THE END
3. AT A GIVEN POSITION
4. EXIT

Enter your choice: 1

Deleted element is: 56

The linked list is

89 → 23 → 35 → NULL

Choose Position To Delete:

1. AT THE BEGINNING
2. AT THE END
3. AT A GIVEN POSITION
4. EXIT

Enter your choice: 2

Deleted element is: 35

89 → 23 → NULL

Choose Position To Delete:

1. AT THE BEGINNING
2. AT THE END
3. AT A GIVEN POSITION
4. EXIT

Enter your choice: 3

Enter Position to delete: 2

Deleted element is: 23

The linked list is

89 → NULL

3.4.5 Searching an item in a linked list

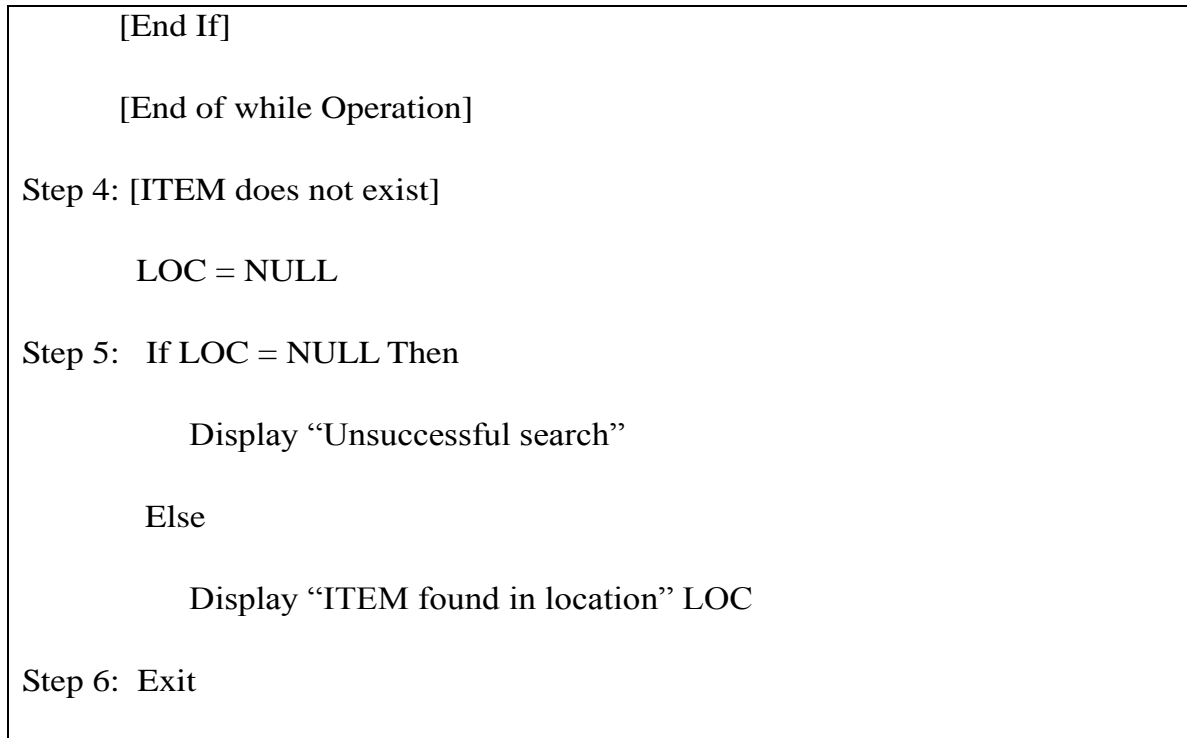
Finding the location of the node with a given key value, or finding the locations of all nodes which satisfy one or more conditions. Suppose a specific ITEM of information is given. Algorithm 3.9 and Algorithm 3.10 finding the location LOC, of the node where ITEM first appears in LIST. There are two cases of searching.

1. Searching an item in an unsorted list.
2. Searching an item in a sorted list

3.4.5.1 Searching an item in an unsorted list

Suppose the data in LIST are unsorted, Then one searches for ITEM in LIST by traversing through the list using a pointer variable CurrPtr and comparing ITEM with the contents INFO[CurrPtr] of each node, one by one, of LIST.

Algorithm 3.9: Searching an item in an unsorted list
Step 1: [Initialize pointer variable CurrPtr] CurrPtr = HEAD Step 2: [Perform the traversing Operation] While CurrPtr \neq NULL Do Step 3: If ITEM = INFO [CurrPtr] Then (a) LOC = CurrPtr (b) GOTO Step 4 Else (a) [Move pointer to next node] CurrPtr = LINK[CurrPtr]



Program 3.6: Program to search an ITEM in a linked list.

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <conio.h>

struct node

{

    int info;

    struct node *link;

};

typedef struct node NODE;
```



```

NODE *HEAD;

void Create_link_list (int);

void Display (NODE*);

NODE *Linear_Search(NODE*, int);

void main()

{

    int ele, pos;

    char ch;

    clrscr();

    HEAD = NULL;

    printf("\n Linked list creation \n \n");

    do{

        printf("Enter information field of the node \n");

        scanf("%d", &ele);

        Create_link_list(ele);

        printf("Add another node \n");

        flushall();

        scanf("%c", &ch);

    }while (toupper (ch)=='Y');

    printf("\n The linked list is \n");

    display (HEAD);

```

```

printf("\n Enter an item to be searched");

scanf("%d", &item);

if(Linear_Search(head, item) == NULL)

    printf("Item %d does not exist", item);

else

    printf("Item %d exist in the list", item);

getch();
}

/*    Function to add a node to the linked list    */

void Create_link_list(int ele)

{

    NODE *NewNode, *CurrPtr;

    NewNode = (NODE*) malloc(sizeof(NODE));

    NewNode -> info = ele;

    NewNode -> link = NULL;

    if (HEAD == NULL)

        HEAD = NewNode;

    else

    {

        CurrPtr = HEAD;

        while(CurrPtr -> link != NULL)

```

```

        CurrPtr = CurrPtr -> link;

        CurrPtr -> link = NewNode;

    }

}

/*  Function to display linked list elements  */

void Display (NODE *CurrPtr)

{

    while(CurrPtr != NULL)

    {

        printf("%d →", CurrPtr -> info);

        CurrPtr = CurrPtr -> link;

    }

    printf ("NULL\n");

}

/*  Function to search a node in a linked list  */

NODE *Linear_Search(NODE *CurrPtr, int ITEM)

{

    While(CurrPtr != NULL)

    {

        if( ITEM == CurrPtr -> info) return (CurrPtr);

        CurrPtr = CurrPtr -> link;

    }

}

```

```

    }

/* return NULL value only when ITEM does not exist in the list */

return NULL;

}

```

OUTPUT

Linked list creation

Enter information field of the node 26

Do you wish to enter one more node? y

Enter information field of the node 859

Do you wish to enter one more node? y

Enter information field of the node 243

Do you wish to enter one more node? n

Linked list elements are

26 → 859 → 243

Enter an item to be searched 859

Item 859 exist in the list

3.4.4.1 Searching an item in a sorted linked list

Suppose LIST is sorted according to ascending order, We therefore, search for ITEM in the list by traversing the list by using CurrPtr and comparing ITEM with the contents INFO[CurrPtr] of each node, one by one. Now, however, we can stop once ITEM exceeds INFO[CurrPtr].

Algorithm 3.10: *LIST is stored in memory. This algorithm finds LOC of the node where ITEM first appears in LIST.*

Step 1: [Initialize pointer variable CurrPtr]

CurrPtr = HEAD

Step 2: [Perform the traversing Operation]

While CurrPtr = NULL Do

Step 3: If ITEM = INFO [CurrPtr] Then

[ITEM is found in location LOC]

(c) LOC = CurrPtr

(d) GOTO Step 5

Else If ITEM > INFO[CurrPtr] Then

(b) [Move pointer to next node]

CurrPtr = LINK[CurrPtr]

Else [If ITEM is less than INFO [CurrPtr]]

GOTO Step 4

[End If]

[End of while Operation]

Step 4: [ITEM does not exist]

LOC = NULL

Step 5: If LOC = NULL Then

Display “Unsuccessful search”

Else

Display “ITEM found in location” LOC

Step 6: Exit

Program 3.8: Program to search an ITEM in a stored linked list

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <conio.h>

struct node

{

    int info;

    struct node *link;

};

typedef struct node NODE;

NODE *HEAD;

void Create_link_list (int);

void Display (NODE*);

NODE *Search_Sorted_List(NODE*, int);

void main()

{

    int ele;
```

```

char ch;

clrscr();

HEAD = NULL;

printf("\n Linked list creation \n \n");

do{

    printf("Enter information field of the node \n");

    scanf("%d", &ele);

    Create_link_list(ele);

    printf("Add another node \n");

    flushall();

    scanf("%c", &ch);

}while (toupper (ch)=='Y');

printf("\n The linked list is \n");

display (HEAD);

printf("\n Enter an item to be searched");

scanf("%d", &item);

if(Search_Sorted_List(head, item) == NULL)

    printf("Item %d does not exist in the list", item);

else

    printf("Item %d exist in the list", item);

getch();

```

```

}

/*    Function to add a node to the linked list    */

void Create_link_list(int ele)

{

    NODE *NewNode, *CurrPtr;

    NewNode = (NODE*) malloc(sizeof(NODE));

    NewNode -> info = ele;

    NewNode -> link = NULL;

    if (HEAD == NULL)

        HEAD = NewNode;

    else

    {

        CurrPtr = HEAD;

        while(CurrPtr -> link != NULL)

            CurrPtr = CurrPtr -> link;

        CurrPtr -> link = NewNode;

    }

}

/*    Function to display linked list elements    */

void Display (NODE *CurrPtr)

{

```



```

while(CurrPtr != NULL)

{

    printf(“%d →”, CurrPtr -> info);

    CurrPtr = CurrPtr -> link;

}

printf (“NULL\n”);

}

/* Function to search a node in a sorted linked list */

NODE *Search_Sorted_List(NODE *CurrPtr, int ITEM)

{

    While(CurrPtr != NULL)

    {

        if( ITEM == CurrPtr -> info) return (CurrPtr);

        CurrPtr = CurrPtr -> link;  /* Move pointer to next node */

        Return NULL;

    }

    return NULL;

}

```

OUTPUT

Linked list creation

Enter information field of the node 26

Do you wish to enter one more node? y

Enter information field of the node 59

Do you wish to enter one more node? y

Enter information field of the node 73

Do you wish to enter one more node? n

Linked list elements are

26 → 59 → 73

Enter an item to be searched 73

Item 73 exist in the list

Enter an item to be searched 45

Item 45 item does not exist in the list

3.5 CIRCULAR LINKED LINEAR LIST

In the normal linked list, the link field of the last element stores the value **NULL**. By instead storing a pointer to the first element in the list, a **circular list** is created. With this implementation, a **TAIL** pointer is no longer needed. However, the **HEAD** pointer can be used as a marker to determine when list processing has worked full circle through the list. Figure 3.11 illustrates a circular singly Linked List.

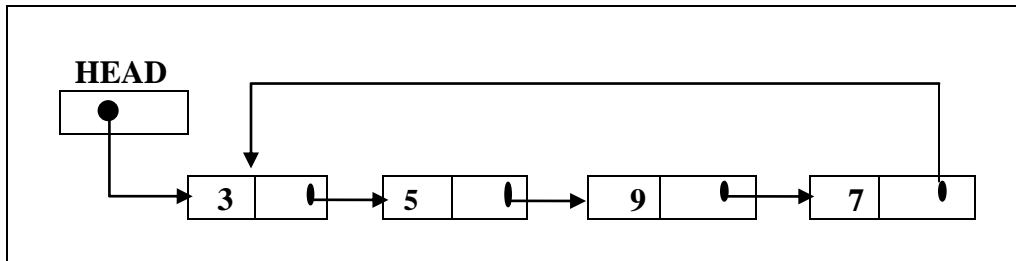


Figure 3.11 Circular Linked List

Algorithm 3.9: *HEAD* is a header node which contains three fields, *NodeNo*, *INFO* and *LINK*. The *LINK* field contains the address of the first node in a circular linked list. *CurrPtr* is a pointer which is being pointing current node of the list.

Step 1: [Create a **head** node]

HEAD = Create a node

Step 2: [Create a new node]

LINK[HEAD] = new node

Step 3: [Initialize node number]

Step 4: [CurrPtr is always point to the last node in the linked list]

CurrPtr = LINK[HEAD]

Step 5: Read INFO[CurrPtr]

Step 6: [Increment node number]

count = count + 1

Step 7: NodeNo[CurrPtr] = count

Step 8: [Create a New Node?]

If answer is Yes Then

(a) [Allocate space to newly created node]

NewNode = Create a node

(b) [NewNode address is assigned to LINK field of the previous Node]

LINK[CurrPtr]=NewNode

(c) [**CurrPtr** pointer should point next node in the list,, i.e.,
NewNode]

CurrPtr = LINK[CurrPtr]

(d) GOTO Step 8

Else

(a) [Last node of the linked list, last node link field should point head
node of the circular list]

LINK[CurrPtr] = HEAD

Step 9: Return

Program 3.9: To illustrate circular linked list operations (Creation and traversing)

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

```
#include <ctype.h>
```

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

```
#include <stdlib.h>
```

```
struct node
```

```
{
```

```
    int NodeNo;
```

```
    int info;
```

```

    struct node *link;

};

typedef struct node NODE;

int count = 0;

/*  Function to create a circular linked list  */

void Create_Circular_list (NODE *head)

{

    char ans;

    NODE *CurrPtr;

    head -> link = (NODE *) malloc(sizeof(NODE));

    CurrPtr = head -> link;

    while(1)

    {

        count ++;

        printf("Enter information field of the node");

        scanf("%d", &CurrPtr -> info);

        CurrPtr -> NodeNo = count;

        printf("\n Do you want enter next node?");

        scanf("%c", &ans);

        if(toupper(ans)=='N')

        {

```

```

        CurrPtr -> link = head;

        break;

    }

    CurrPtr -> link = (NODE *) malloc(sizeof(NODE));

    CurrPtr = CurrPtr -> link;

}

Head -> NodeNo = count;

}

/*   Function to display circular linked list elements   */

void Display (NODE *head)

{

    int i;

    head = head -> link;

    for(i = 1; i <= count; head = head -> link, i++)

        printf("\n Node %d information is %d, head -> NodeNo, head -> info);

}

/*   Main function   */

Void main ( )

{

    NODE *head;

    clrscr( );

```

```
    head = (NODE *) malloc(sizeof(NODE));

    Create_Circular_list(head);

    printf("Number of nodes in the circular linked list is %d", head -> NodeNo);

    Display (head);

}
```

OUTPUT

Enter information field of the node 25

Do you want enter next node? Y

Enter information field of the node 40

Do you want enter next node? Y

Enter information field of the node 60

Do you want enter next node? Y

Enter information field of the node 5

Do you want enter next node? n

Number of nodes in the circular linked list is 4

Node 1 information is 25

Node 2 information is 40

Node 3 information is 60

Node 4 information is 5

Advantages of circular linked list

1. In a circular linked list every node is accessible from a given node. That is from the given node, nodes can be reached by merely chaining of the list.
2. Concatenation and splitting the operations are more efficient.

3.6 DOUBLY LINKED LIST

The Singly linked list presented in previous section allows for direct access from a list node only to the next node in the list. This list is named as singly linked list because each list element contains a pointer to the next element. In singly linked list, traversing is possible only in one direction. Sometimes it is required to traverse a list in forward direction or backward direction. A doubly linked list is designed to allow convenient access from a list node to the next node and also to the preceding node on the list. This is also used to traverse the list in both the directions.

In a doubly linked list each node has not only pointer to the next node but also a pointer to the prior node. Each node of doubly linked list is divided into three parts (fields):

1. A pointer BACK (leftlink) which contains the location of the preceding node in the list.
2. An information field INFO which contains data of N.
3. A pointer FORWARD (rightlink) which contains location of the next node in the list.

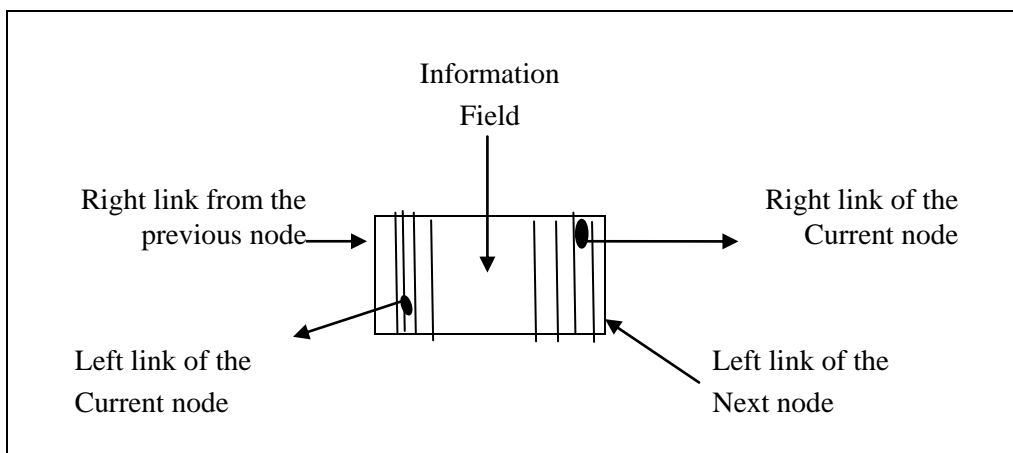


Figure 3.14 Doubly linked list

The doubly linked list is also called as **two-way** list because we can traverse the list in two directions: in the usual forward direction from the beginning of the list to end, or in the backward direction from the end of the list to the beginning.

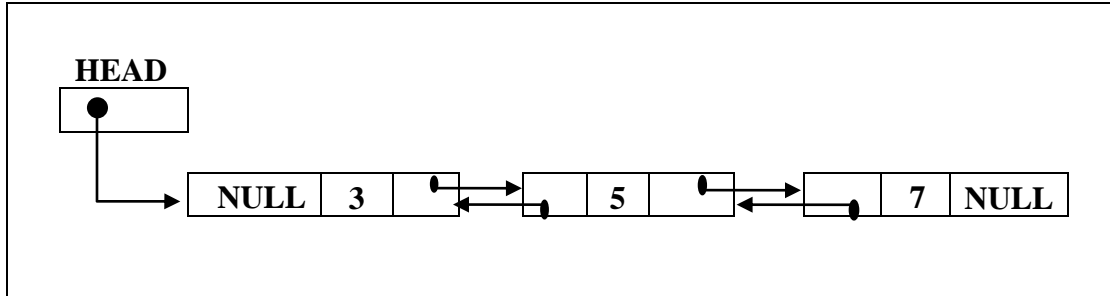


Figure 3.15 Doubly linked list

The doubly linked list is useful primarily because it is easier to implement operations than a singly linked list. We may consider the nodes on a doubly linked list to consist of three fields: an info field that contains the information stored in the node, and left and right fields that contain pointers to the nodes on either side. In the figure, **LEFT** and **RIGHT** are pointer variables denoting the left-most and right-most nodes in the list, respectively. The left link of the left-most node and the right link of the right-most node are both **NULL**, indicating the end of the list for each direction. The left and right links of a node are denoted by the variables **llink** and **rlink**, respectively. We may declare a set of such nodes using dynamic implementation, by

```

struct node

{

    int info;           //    Information field

    struct node *rlink; //    a pointer which is pointing right node

    struct node *llink; //    a pointer which is pointing left node

};

```

Algorithm 3.10: *This algorithm is used to create a doubly linked list whose left-most and right-most node addresses are given by the pointer variables **LEFT** and **RIGHT**, respectively.*

Step 1: [Initialize LEFT and RIGHT pointers]

LEFT = NULL and RIGHT = NULL

Step 2: [Create a doubly linked list]

Repeat Step 3 through Step 6 to create required number of nodes for
Linked list

Step 3: [Obtain new node from availability list]

NewNode = Address of new node

Step 4: [Copy information field]

INFO[NewNode] = ITEM

Step 5: RLINK[NewNode] = NULL and LLINK[NewNode] = NULL

Step 6: [Insertion into an empty list?]

If RIGHT = NULL Then

(a) RIGHT = NewNode and LEFT = NewNode

Else

(a) LLINK[NewNode] = RIGHT

(b) RLINK[RIGHT] = NewNode

(c) RIGHT = RLINK[RIGHT]

[EndIf]

[End of Step 2 loop]

Step 7: End

Program 3.10: Program to create a doubly linked list

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <conio.h>

struct node

{

    int info;

    struct node *link;

    struct node *link;

};

typedef struct node NODE;

/*  Function to display doubly linked list in forward direction  */

void Display_Forward (NODE *CurrPtr)

{

    While(CurrPtr)

    {

        printf("%d→", CurrPtr -> info);
```

```

        CurrPtr = CurrPtr -> link;

    }

}

/* Function to display doubly linked list in backward direction */

void Display_Backward (NODE *CurrPtr)

{

    While(CurrPtr)

    {

        printf("%d→", CurrPtr -> info);

        CurrPtr = CurrPtr -> link;

    }

}

/* Function to create a doubly linked list node */

NODE *GetNode( )

{

    NODE *NewNode = (NODE *) malloc(sizeof(NODE));

    NewNode -> llink = NULL;

    NewNode -> rlink = NULL;

    printf("Enter information field of the node");

    scanf("%d", &NewNode -> info);

    return NewNode;

```

```

}

/*  Main function  */

void main ( )

{

    NODE *RIGHT = NULL, *LEFT = NULL, *NewNode;

    char ans;

    while(1)

    {

        printf("Do you want to enter a new node?");

        scanf("%c", &ans);

        if(toupper(ans)=='N') break;

        NewNode = GetNode( );

        if(RIGHT == NULL)

        {

            LEFT = NewNode;

            RIGHT = NewNode;

        }

        else

        {

            NewNode -> llink = RIGHT;

            RIGHT -> link = NewNode;

```

```

        RGHT = RIGHT -> link;

    }

}

Printf("\n Doubly Linked list elements \n");

Display_Forward (LEFT)

Printf("\n Doubly linked list in the reverse direction \n");

Display_Backward (RIGHT);

}

```

OUTPUT

Do you want to enter a new node? Y

Enter information field of the node 34

Do you want to enter a new node? Y

Enter information field of the node 56

Do you want to enter a new node? Y

Enter information field of the node 7

Doubly Linked list elements

34 → 56 → 7

Doubly linked list in the reverse direction

7 → 56 → 34

3.6.4 Advantages of Doubly linked list

1. One can traverse the list in both directions.
2. Insertions and deletions are easy.
3. It is used to present trees.

3.7 APPLICATION OF THE LINKED LIST

Linked list are used in a wide variety of applications. The following list gives the applications of linked list.

1. It is used to represent a polynomial.
2. It is used to find the sum of two long integers.
3. It is used to represent queues, stacks in memory resulting in efficient use of storage and computer time.
4. It is used in symbol table construction.
5. One interesting use of linked lists is a line editor. You can keep a linked list of line nodes, each containing a line number, a line of text, and a pointer to the next line information node. Since you can predict how many lines will be needed, this would be an appropriate application to implement with dynamically created nodes.
6. Operating systems use linked lists in many ways. The allocation of memory space may be managed using a doubly linked list of variably sized blocks of memory. Doubly linking the list facilitates removal of blocks from the middle of the list. In a multi-user system, the operating system may keep track of user jobs waiting to execute through linked queues of control blocks.

SELF ASSESSMENT QUESTIONS

1. What is a linked list?
2. What is the difference between linked list and array?
3. Mention different operations on a linked list.
4. What do you mean by dynamic allocation?
5. What do you mean by a head node?
6. What are the advantages of linked list?
7. Mention the applications of linked list.
8. Mention different types of linked list.
9. . Write an algorithm and a C program to traverse a linked list.
10. Wnte an alorhgm and a C program to count number of nodes in a linked list.
11. Write an algorithm to insert a node:
 - (a) At the beginning
 - (b) At the end
 - (c) Between two nodes
12. Write an algorithm and a C program to insert a node in a linked list.
13. Give an algorithm and a C function to insert an item after a given node.
14. Write an algorithm and a C program to search an item in a linked list.
15. Write an algorithm and a C program to delete a node from a linked lts.
16. Give an algorithm and a C function for deleting a node, given the location.
17. What is a doubly linked list?
18. Mention the applications of Linked List.

PART 4

STACKS BASED DATA STRUCTURES

4.1 INTRODUCTION

This chapter describes representation of the stack an important data structure often used to simulate real world situations. The concepts of the stack are then extended to a new structure, the list. Various forms of lists and their associated operations are examined and several applications are represented.

4.2 BLUE-PRINT OF STACKS

Consider the illustrations in Figure 4.1. Although the various pictures are very different, each illustrates a common concept: the stack.

Definition
Stack: A list in which elements are added and removed from only one end; a “Last In First Out (LIFO) structure.

At the logically level, a **stack** is an ordered collection of elements. The removal of elements from the stack and addition of new elements to it can take place only at the **top** of the stack. For example, if your favorite Data Structures book is in the stack of books underneath your C programming book, you must first remove the C programming book (the top element) from the stack. Only then can you remove the desired Data Structures book, which is now the top element in the stack. The C Programming book may then be replaced on the top of the stack or return to Library.

At any time, given any two elements in a stack, one element is higher than the other; that is, one is closer to the top of the stack (C++ Programming book was higher in the stack than the C Programming.) In this sense, the stack is an ordered collection of elements. Since the elements in a stack may constantly change, it is considered a dynamic structure.

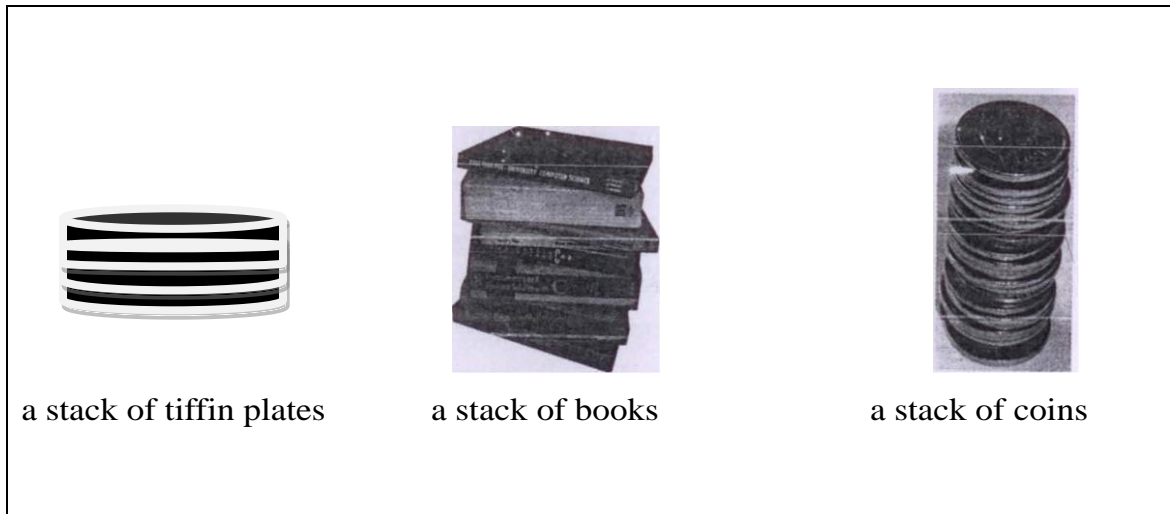


Figure 4.1 Real life Stacks

Consider another example, a stack of plates placed on the counter in a cafeteria. During the dinner time, the visitors take plates from the top of the stack and the home boy puts the washed plates on the top of the stack. The plate that is put most recently on the stack is the first one to be taken off. The plate at the bottom is the last one to be used.

Because elements are added and removed from the top of the stack, the last element added is the first to be removed. Accordingly, stacks are also called Last-In-First-Out (LIFO) lists.

4.3 STACK OPERATIONS

It is traditional to call the accessible element of the stack the **top** element. Elements are not said to be inserted; instead they are **pushed** onto the stack. When removed, an element is said to be **popped** from the stack. The fundamental operations, which are possible on a stacks are:

1. Push - Insertion of new element at the top of the stack
2. Pop - Deletion top element from the top of the stack
3. Traverse - Display current elements in the stack

The stack is a restricted list in which elements can be inserted or removed from only one end called top of the stack. Many applications require only the limited form of insert and remove operations that stack provide. Figure 4.2 illustrates stack as a building blocks with respect to PUSH and POP operation.

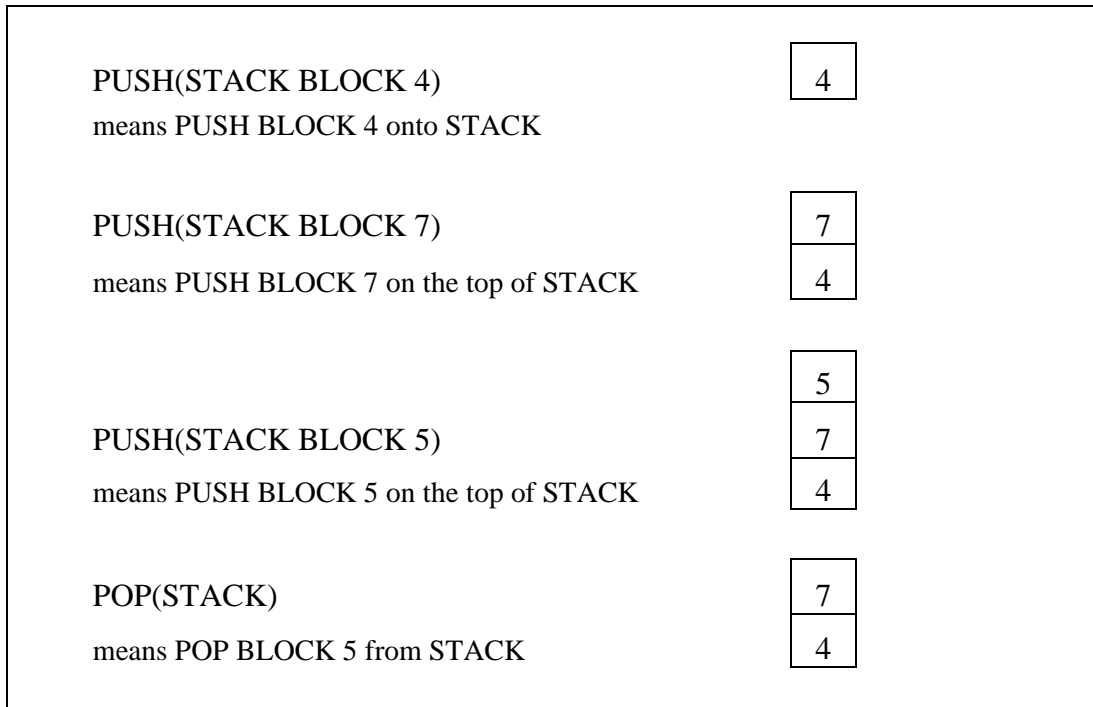


Figure 4.2 Stack operations (PUSH and POP)

4.4 IMPLEMENTATION OF STACK

The stack can be implementation following **array-based** and **linked List based** approaches. In this Edition we analogous implement stack based data structure using to array-based .

4.4.1 Memory representation of stack with array

Since all the elements of the stack are of the same type, an array seems like a resonable structure to contain a stack. We can place the elements in sequential order in the array; the first element in the first array position, the second element in the second array position, and so on.

Stack will be maintained by a linear array **STACK**, a pointer variable **TOP** which contains the location of the top element of the stack and variable **N** which gives maximum number of elements that can be accommodated in the stack. However, stack represent two conditions, $TOP = 0$ will indicate the stack is empty; $TOP = N-1$ will indicate the stack is Full. The Figure 4.3 illustrates an array representation of stack.

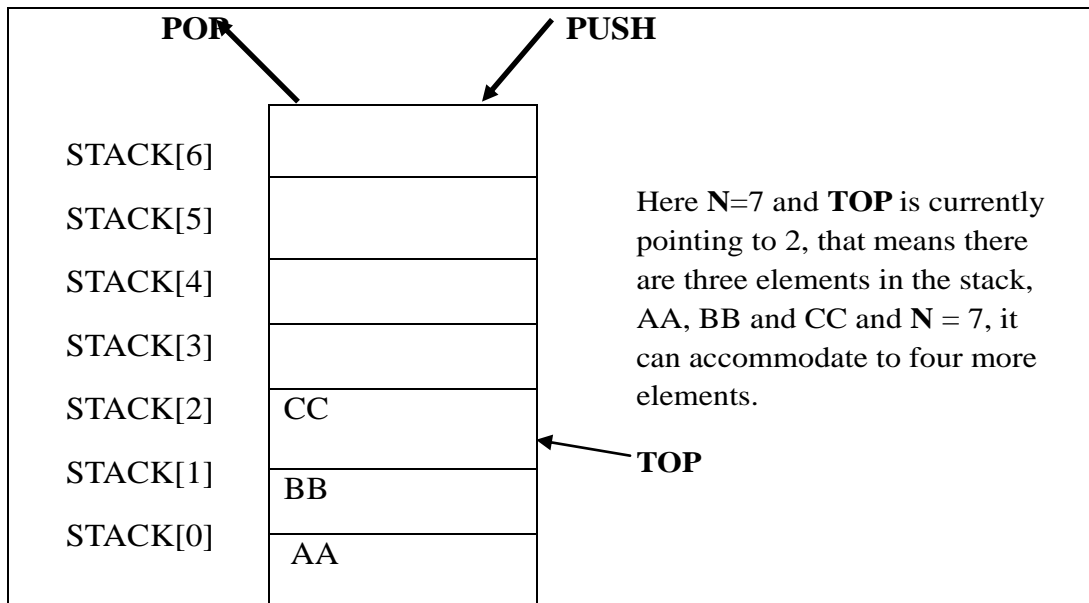


Figure 4.3 Memory representation of a stack

If the stack is already full, when we try to PUSH, the result is called stack **overflow condition**. Error checking for overflow may be handled in different ways. We could test for overflow inside the PUSH module to restrict and inform the user that there is not free space to allocate a new item into the stack..

4.5 ALGORITHM FOR PUSH OPERATION

Algorithm 7.1: PUSH(STACK, TOP, NEW_ITEM) *This algorithm inserts an element NEW_ITEM to the top of the stack represented by an array STACK containing N elements with a pointer TOP denoting the top element in the stack.*

Step 1: [Check for overflow]

If $TOP = N-1$ Then

<pre> Print "Sorry, Stack is full or overflow condition" Exit [EndIf] Step 2: [Increment TOP pointer] TOP = TOP + 1 Step 3: [Insert an NEW_ITEM] STACK[TOP] = NEW_ITEM Step 4: Return </pre>

When the stack is empty and whenever we try to pop top element, the resulting condition is called stack **underflow condition**. The test for underflow could be written into the POP module to alert the user that there is nothing to remove from the stack and that the stack is empty. Initially, top is set to -1 to demonstrate underflow condition.

4.6 ALGORITHM FOR POP OPERATION

<p>Algorithm 7.1: POP(STACK, TOP) <i>This algorithm removes the top element from a stack which is represented by an array STACK and return this element. TOP is a pointer to the top of the element.</i></p>
<pre> Step 1: [Check for underflow] If TOP = -1 Then Print "Stack is empty or Underflow" Exit [EndIf] </pre>

Step 2: [Return top element of the stack]

Return STACK [TOP]

Step 3: [Decrement pointer]

TOP = TOP - 1

Step 4: Return

4.7 REPRESENTATION OF STACK OPERATIONS IN C

There are several ways of representing a stack in C however, we must decide how to represent a stack using the data structures that exist in our programming language C.

A stack is an ordered collection of items, and in C an array is an ordered collection of items of the same data type. Whenever a program requires the use of a stack, it is better to begin a program by declaring a variable stack as an array even if stack and an array are two different things. The number of elements in an array is fixed and is assigned by the declaration for the array. In general, we cannot change this number. On the other hand, a stack is a dynamic data structure whose size is frequently changing as items are inserted (pushed) or deleted (popped).

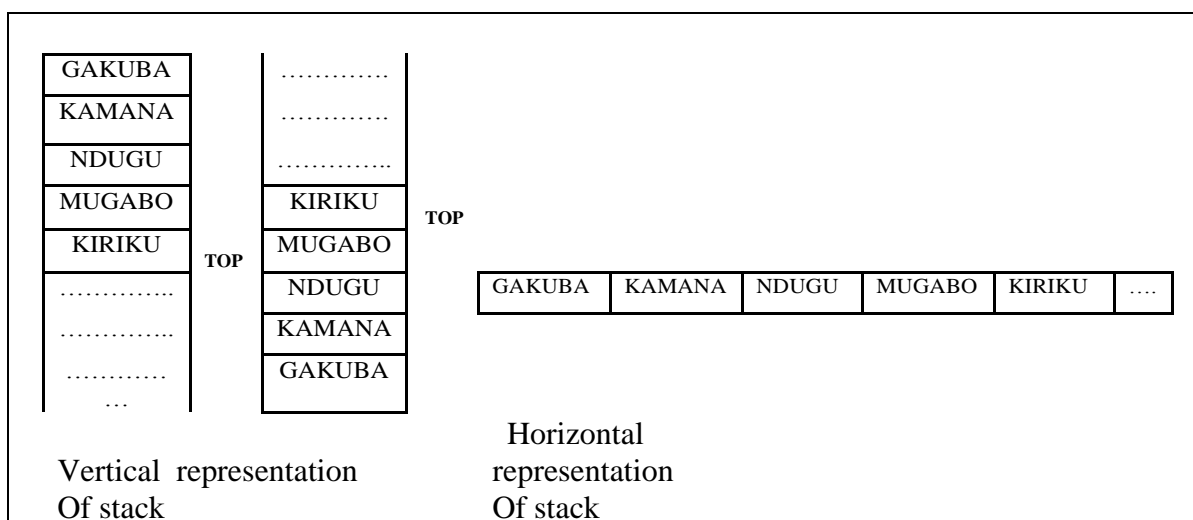


Figure 4.4 the three ways of representing a stack using arrays.

However, an array cannot be a stack; it can be the storage location of a stack. That is, an array can be declared large enough for the maximum size of the stack. During the program execution, the stack can grow and reduce in size within the space reserved for it. Thus, another field is needed at each point during program execution to keep track of the current position of the top of the stack. **Figure 4.4** Representation of stack

When working with stacks, the three basic operations – store, remove and retrieve are traditionally called **push**, **pop** and **Traverse** respectively. Therefore to implement a stack you need two functions: **PUSH()** which places a value on the stack, and **POP()** which removes top element from the stack and **Traverse** which retrieves value(s) from the stack. You also need a region of memory using C's dynamic allocation functions.

```
int STACK[N]
```

```
Static int TOP = -1; /* TOP of stack */
```

The following listing is the C functions for PUSH and POP operations.

```
/* Function to push an item onto a Stack */
```

```
void PUSH( )
```

```
{
    int item;
    if (TOP == N - 1)
    {
        printf("Stack is full or overflow");
        getch( );
    }
    else
    {
        printf("Enter an item to be pushed:");
        Scanf("%d", &item);
        TOP++;
        STACK[TOP] = item;
    }
}
```

```

/*  Function to pop an item from the Stack  */
void POP( )
{
    int item;
    if (TOP == - 1)
    {
        printf("Underflow");
    }
    else
    {
        item = STACK[TOP];
        printf("\n Deleted item is %d", &item);
        TOP--;
    }
}

```

The variable **TOP** is the index of the top of the stack. When implementing these functions, you must remember to prevent overflow and underflow conditions. In these routines, an empty stack is signalled by **TOP** being **-1** and a full stack by **TOP** attempting to be greater than the last storage location. To see how a stack works, see the following Figure 4.5

ACTION	CONTENTS OF STACK
PUSH (A)	A
PUSH (B)	BA
PUSH (C)	CBA
POP () retrieves C	BA
PUSH (F)	FBA
POP () retrieves F	BA
POP () retrieves B	A
POP () retrieves A	Empty

Figure 7.5 A stack in action

Program 4.1: Program to illustrate STACK operations (PUSH ,POP and tranverse operations).

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#define N 5
int STACK[N]
Static int TOP = -1;
/* Function to push an item onto a Stack */
void PUSH( )
{
    int item;
    if (TOP == N - 1)
    {
        printf("Stack is full or overflow");
        getch( );
    }
    else
    {
        printf("Enter an item to be pushed:");
        scanf("%d", &item);
        TOP++;
        STACK[TOP] = item;
    }
}
/* Function to pop an item from the Stack */
void POP( )
{
    int item;
    if (TOP == - 1)
    {
        printf("Stack is empty! Underflow");
    }
    else
    {
        item = STACK[TOP];
        printf("\n Deleted item is %d", &item);
        TOP--;
    }
    getch( );
}
```

```

}
/* Function to Display Stack elements */
void RETRIEVE( )
{
    int i;
    for(i=TOP;i>=0;i--)
    {
        printf("STACK[%d]=%s\n-----",i,STACK[i]);
    }
    getch();
}
/* Function wrapper */
Void main()
{
    int choice;
    while(1)
    {
        clrscr ( );
        printf("Welcome to Stack Operations:\n Press");
        printf("\n1. To Push new element");
        printf("\n2. To Pop top element");
        printf("\n3. To Display current value(s) ");
        printf("\n4. To Exit the application");
        printf("\n Enter your choice:");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:  PUSH( );
                     break;
            case 2:  POP( );
                     break;
            case 3:  RETRIEVE( );
                     break;
            case 4:  exit(0);
        }
    }
}

```

OUTPUT

Welcome to Stack Operations:
Press:
1. To Push new element
2. To Pop top element
3. To Display current value(s)
To Exit the application
Enter your choice: 1
Enter an item to be pushed: 3
Welcome to Stack Operations:
Press:
4. To Push new element
5. To Pop top element
6. To Display current value(s)
To Exit the application
Enter your choice: 1
Enter an item to be pushed: 4
Welcome to Stack Operations:
Press:
7. To Push new element
8. To Pop top element
9. To Display current value(s)
To Exit the application
Enter your choice: 1
Enter an item to be pushed: 7
Welcome to Stack Operations:
Press:
10. To Push new element
11. To Pop top element
12. To Display current value(s)
To Exit the application
Enter your choice: 3

STACK[2] = 7
STACK[1] = 4
STACK[0] = 3
Welcome to Stack Operations:
Press:
13. To Push new element
14. To Pop top element
15. To Display current value(s)
To Exit the application
Enter your choice: 2
Deleted item is 7
Welcome to Stack Operations:
Press:
16. To Push new element
17. To Pop top element
18. To Display current value(s)
To Exit the application
Enter your choice: 2
Deleted item is 4
Welcome to Stack Operations:
Press:
19. To Push new element
20. To Pop top element
21. To Display current value(s)
To Exit the application
Enter your choice: 2
Deleted item is 3
Welcome to Stack Operations:
Press:
22. To Push new element
23. To Pop top element
24. To Display current value(s)
To Exit the application
Enter your choice: 2
Stack is empty! Underflow

4.8 APPLICATIONS OF STACK

Application 1: An example of stack usage is four-function calculator. Most calculators today accept standard format of an expression called infix notation, which takes the general form operand-operator-operand. For example, to add 200 to 100, enter 100, then press the PLUS (+) key, then 200, and press the EQUAL(=) key. In contrast, many calculators (and some still made today) use postfix notation, in which both operands are entered first and then the operator is entered. For example, to add 200 to 100 by using postfix notation, you enter 100, then 200, and then press PLUS key. As operands are

entered, they are replaced on a stack. Each time an operator is entered, two operands are removed from the stack and the result is pushed back on the stack.

Application 2: RECURSION - One of the benefits of using a stack to implement method invocation is that it allows programs to use recursion. That is, it allows a method to call itself as subroutine. Recursion can be very powerful, as it often allows us to design simple and efficient programs for fairly difficult problems.

For example, we can use recursion to compute the factorial of a number, $n! = n * (n-1)!$, as shown in the following program segment:

```
long FACTORIAL(int N)
{
    if(N==0) return 1;

    return N* FACTORIAL (N-1);
}
```

SELF ASSESSMENT QUESTIONS

1. What is a stack?
2. Give an example for stack.
3. What is PUSH operation in stack?
4. What is POP operation in stack?
5. Define LIFO list.
6. Write an algorithm and a C program to perform Stack operations.
7. Mention the applications of stack.

PART 5

QUEUE BASED DATA STRUCTURES

1.0 INTRODUCTION

Another fundamental data structure is the **Queue**. It is a linear data structure represent a container of objects that are inserted and removed according to the **first-in-first-out (FIFO)** principle. That is, elements can be removed at any time, but only the element that has been in the queue the longest can be removed at any time; that is random access of any specific item is not allowed. Queue elements may be inserted at the back (called the REAR end) and removed from the front end (called FRONT end).

Definition
Queue: A linear list of elements in which deletion can take place only at one end, called the <i>front</i> , and insertion can take place only at the other end, called the rear.

Queues operate like standing in line at a Bank Teller Machine, then newcomers go to the back of the line. The person at the front of the line is the next to be served. Consider the automobile waiting to pass through Traffic right from a queue, in which the first car in line is the first car through. Getting a bus ticket at Nyabugogo bus stand is also an example of queue where the people collect their tickets on the **first-come-first-serve** basis. An important example of a queue in computer science occurs in a time sharing system. In this system many users share the system simultaneously. The procedure, which is used to design such type of system, is **Round Robin Technique**.

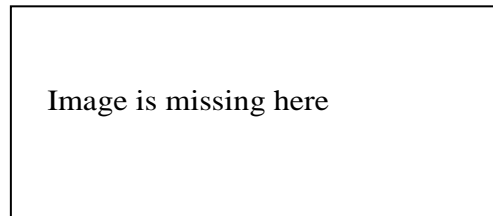
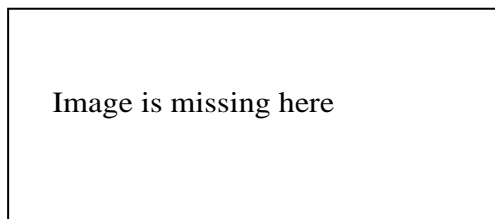


Figure 5.1 a) Queue waiting in Bank

b) Automobile waiting to pass Traffic right

5.1 REPRESENTATION OF QUEUES AND QUEUE OPERATIONS

Two common ways in which queues may be implemented are as follows:

- i. Array based queue and
- ii. Pointers (One way linear linked list)

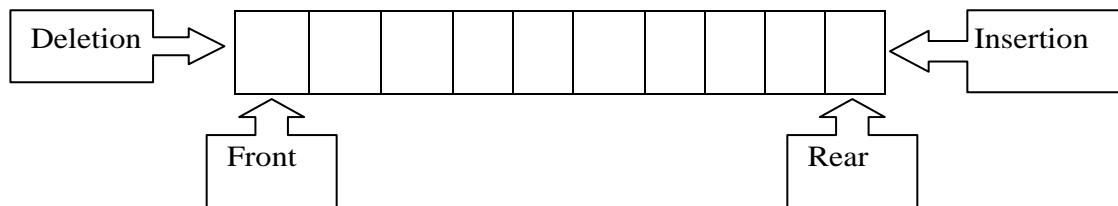


Figure 5.2 Queue

Formally, the queue defines a sequence-oriented object container where element access and deletion is restricted to the first element in the sequence, which is called the **FRONT** of the queue, and element insertion is restricted to the end of the sequence, which is called the **REAR** of the queue. This restriction enforces the rule that items are inserted and deleted in a queue to the first-in-first-out (FIFO) principle.

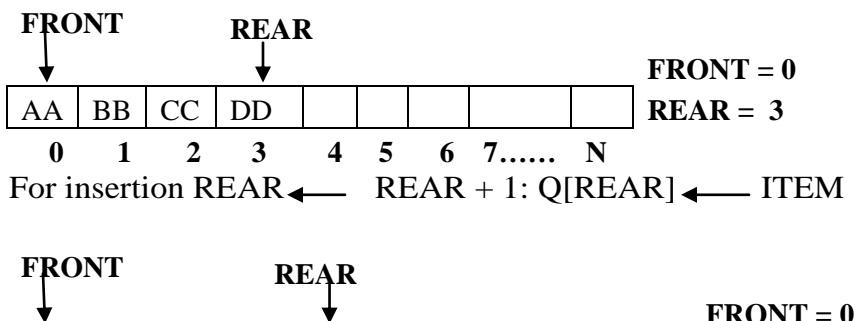
To visualize how a queue works, consider two functions: **Queue_Insert()** and **Queue_Delete()**. The **Queue_Insert()** function places a new element onto the end of the Queue, and **Queue_Delete()** removes the first element from the front of queue and returns its value. The following Figure 5.3 shows a series of queue operations and their effects on an initially empty queue Q.

ACTION	Content of the Queue						
Queue_Insert(A)	<table><tr><td>A</td><td></td><td></td><td></td><td></td></tr></table>	A					Insert A
A							
Queue_Insert(B)	<table><tr><td>A</td><td>B</td><td></td><td></td><td></td></tr></table>	A	B				Insert B
A	B						
Queue_Insert(C)	<table><tr><td>A</td><td>B</td><td>C</td><td></td><td></td></tr></table>	A	B	C			Insert C
A	B	C					
Queue_Delete()	<table><tr><td></td><td>B</td><td>C</td><td></td><td></td></tr></table>		B	C			Remove A
	B	C					
Queue_Insert(D)	<table><tr><td></td><td>B</td><td>C</td><td>D</td><td></td></tr></table>		B	C	D		Insert D
	B	C	D				
Queue_Delete()	<table><tr><td></td><td></td><td>C</td><td>D</td><td></td></tr></table>			C	D		Remove B
		C	D				
Queue_Delete()	<table><tr><td></td><td></td><td></td><td>D</td><td></td></tr></table>				D		Remove C
			D				

Figure 5.3 A Queue in action

5.2 Representation of queues using arrays

When a queue is created as an array, the number of elements are declared before processing. Let **Q** be a linear array of size **N**. Two pointer variables **FRONT** and **REAR** are used to keep track of the insertions and deletions. **FRONT** pointer contains the location of front element of the queue (element to be deleted) and a **REAR** pointer containing the location of the rear element of the queue (recently added element). The condition **FRONT = REAR = -1** indicates that the queue is empty. To add the elements, the value of the **REAR** is incremented by 1. Similarly to delete, the element value of **FRONT** is incremented by 1. If **REAR = N-1** then queue is full.



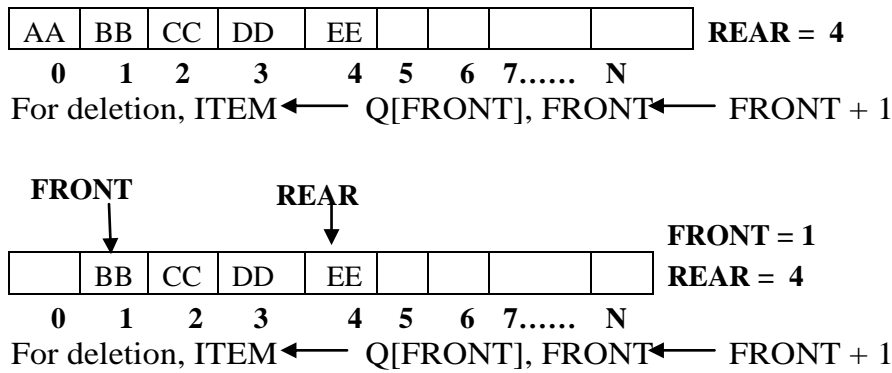


Figure 5.4 Insertion and deletion operation in a queue.

5.3 IMPLEMENTATION OF QUEUE

5.3.1 Algorithm for Insertion

Algorithm 5.1: QINSERT-Let Q be a Queue consisting of N elements. **FRONT** and **REAR** are pointers to the front and rear elements of a queue. This algorithm inserts **ITEM** at the rear end of the queue.

Step 1: [Check for overflow]

 If $REAR = N-1$ Then

 Display "Overflow!"

 Exit

 [EndIf]

Step 2: [Check for empty queue]

 If $FRONT = -1$ Then

$FRONT = 0$ and $REAR = 0$ [Insert ITEM as first element of the queue]

 Else

 [Increment REAR pointer]

$REAR = REAR + 1$

 [EndIf]

Step 3: Q[REAR] = ITEM

Step 4: Return

5.3.2 Algorithm for Deletion

Algorithm 5.2: QINSERT-*Let Q be a Queue consisting of N elements. **FRONT** and **REAR** are pointers to the front and rear elements of a queue. This algorithm deletes front element of the queue.*

Step 1: [Check for Underflow]

 If FRONT = -1 Then

 Display “Underflow /nothing to delete!”

 Exit

 [EndIf]

Step 2: [Delete front element]

Step 3: [Empty queue?]

 If FRONT = REAR Then

 [Queue has only one element]

 FRONT = -1 and REAR = -1

 Else

 [Increment FRONT pointer]

 FRONT = FRONT + 1

 [EndIf]

Step 4: Return

5.3.3 Demonstration of Queue operations using C

Program 5.1: Program to perform queue operations.

```
#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#define MAX 5

int REAR = -1, FRONT = -1;

int Q[MAX];

/*    Function to display interactive screen    */

void screen( )

{

    printf("\n1. Queue Insert");

    printf("\n2. Queue Delete");

    printf("\n3. Display");

    printf("\n4. Exit");

    printf("\n Enter your choice");

}

/*    Function to insert an item into queue    */

void QInsert( )

{

    int item;

    if(REAR == MAX-1)
```

```

    {

        printf("Queue is full. Overflow!");

        getch( );

    }

    else

    {

        printf("Enter an item to be inserted");

        scanf("%d", &item);

        if(FRONT == -1)

        {

            FRONT = 0;

            REAR = 0;

        }

        else

            REAR++;

        Q[REAR] = item;

    }

}

/*  Function to delete an item from the queue  */

void QDelete( )

{

```

```

int item;

if(FRONT == -1)

{
    printf("Queue is empty. Underflow!");
    getch( );
}

else

{
    item = Q[FRONT];

    printf("Deleted item = %d", item);

    getch( );

    if(REAR == FRONT)

    {
        FRONT = -1;

        REAR = -1;

    }

    else

        FRONT++;

}

}

```

```

/*   Function to display elements   */

void QDisplay( )

{

    int i;

    if(FRONT == -1)

        printf("Empty queue!\n");

    else

        for(i=FRONT; i<= REAR; i++) printf("%d->", Q[i]);

    getch( );

}

/*   Main function   */

void main( )

{

    int choice;

    while(1)

    {

        Screen( );

        scanf("%d", &choice);

        switch(choice)

        {

            case 1:    printf("\n Insert operation \n");

```

```

        QInsert( );

        break;

    case 2:    printf("\n Delete operation \n");

               QDelete( );

               break;

    case 3:    printf("\n Queue elements \n");

               QDisplay( );

               break;

    case 4:    printf("Good bye!!");

               Exit(0);

        }

    }

}

```

OUTPUT

1. Queue insert
2. Queue Delete
3. Display
4. Exit

Enter your choice 1

Insert operation

Enter an item to be inserted 2

1. Queue insert

2. Queue Delete

3. Display

4. Exit

Enter your choice 1

Insert operation

Enter an item to be inserted 3

1. Queue insert

2. Queue Delete

3. Display

4. Exit

Enter your choice 1

Insert operation

Enter an item to be inserted 1

1. Queue insert

2. Queue Delete

3. Display

4. Exit

Enter your choice 2

Queue deletion

Deleted item is = 1

1. Queue insert
2. Queue Delete
3. Display
4. Exit

Enter your choice 3

Queue elements

3 -> 1 ->

1. Queue insert
2. Queue Delete
3. Display
4. Exit

Enter your choice 4

Example 5.1: Consider a queue of size 4. Assume that queue is initially empty. It is required to insert BLUE, BLACK, GREEN, YELLOW, and followed by delete BLUE, BLACK. Figure 5.5 give a trace of the queue contents fo this sequence of operations.

				FRONT = -1
0	1	2	3	REAR = -1

BLUE				FRONT = 0
0	1	2	3	REAR = 0

BLUE	BLACK			FRONT = 0
0	1	2	3	REAR = 1

BLUE	BLACK	GREEN		FRONT = 0
0	1	2	3	REAR = 2

BLUE	BLACK	GREEN	YELLOW	FRONT = 0
------	-------	-------	--------	------------------

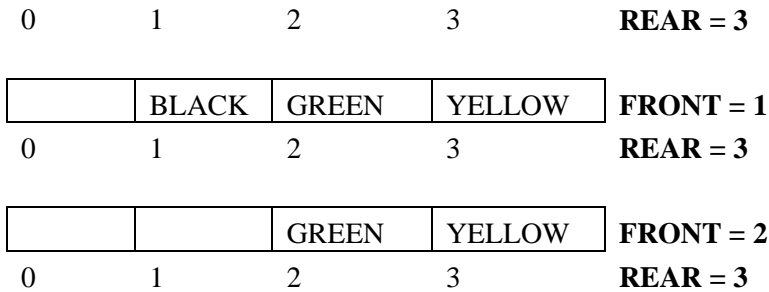


Figure 5.5 A queue in insertion and deletion actions.

If we try to insert ORANGE, overflow condition occurs even though first two cells are free. To avoid this drawback, we assume that array Q is circular, that is $Q[0]$ comes after $Q[N]$ in the array (We can arrange the elements in a circular fashion). Instead of increasing $REAR$ to $N+1$, we reset $REAR=0$ and then assign $ITEM, Q[REAR] = ITEM$.

5.5 APPLICATIONS OF QUEUES

1. In a computer network messages from one computer to another computer are generally created synchronously. These messages need to be buffered until the receiving computer is ready for it. These communication buffers make use of queues by storing these messages in a queue. Also the messages need to be sent to receiving computer in the same order in which they are created.
2. Queues are used in timesharing system in which programs form a queue while waiting to be executed.
3. Perhaps the most common use of circular queue is in operating systems, where a circular queue holds the information read from and written to disk files or the console. Circular queues are also used in real-time application, which must continue to process information while buffering I/O requests. Many word processor do this when they format a paragraph or justify a line. What is being typed is not displayed until the other process is complete. To accomplish this, the application program needs to check for keyboard entry during the other process's execution. If a key has been typed, it is quickly placed in the queue and the other process continues. Once the process is complete, the characters are retrieved from the queue.

SELF ASSESSMENT QUESTIONS

1. What is a queue?
2. Give an example for a queue.
3. Give the memory representation of a queues.
4. What is FIFO list?

PART 6

TREE BASED DATA STRUCTURES

1.0 INTRODUCTION

It is always believed that a great extent can be achieved in life by not thinking in one direction but by thinking "nonlinearly". One of the most important nonlinear concepts in computer science is a **tree**. Trees provide a natural organization for data, and this helps in implementing a lot of problems efficiently. When we say that trees represent a “**nonlinear**” relationship, we are referring to an organizational relationship that is richer than the simple "**before**" and "**after**" relationships that exist between objects in sequences. The relationships in a tree are hierarchical, with some objects being "**above**" and some "**below**" others. Actually, the main terminology for tree data structures comes from family trees, with the terms "**parent**," " **children**," "**ancestor**," and "**descendent**" being the most common words used to describe relationships.

Definition
Tree: non-linear data structures composed of a root node having children that are also nodes that can have children and so on.

Trees are encountered frequently in everyday life. In a linked list each node has a link which points to another node. In a tree structure, however, each node may point to several other nodes (which may then point to several other nodes, etc.). Thus a tree is a very flexible and powerful data structure that can be used for a wide variety of applications.

1.1 THE TREE ABSTRACT DATA TYPE

A tree is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more **children** elements. A tree is usually visualized by placing elements in inside ovals or rectangles, and by drawing the connections between parents and children with straight

lines. The top most element of tree is called as the **root** of the tree, but it is drawn as the highest element, with the other elements being connected below.

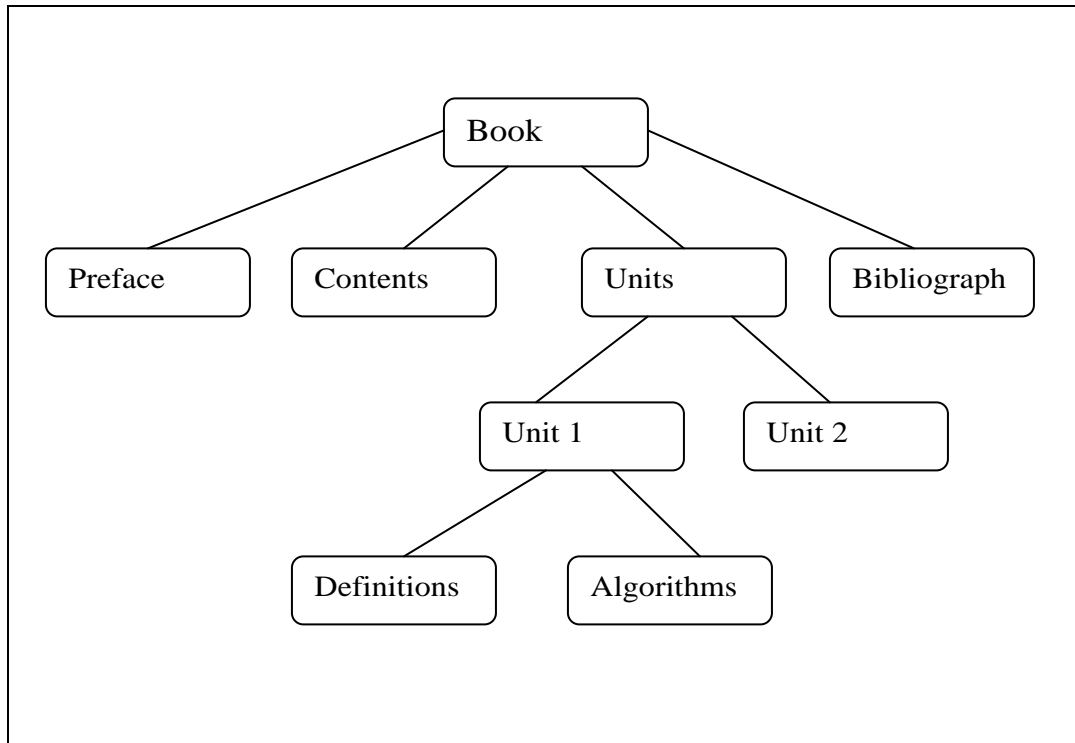


Figure 6.1 To represent a book Content Structure

1.2 TERMINOLOGIES AND BASIC PROPERTIES

A tree **T** represents a set of nodes storing elements in a parent-child relationship model with the following properties:

1. **T** has a distinguished node **R**, called the root of **T**, which has no parent.
2. Each node **V** of a distinct **T** from **R** has parent node **U**.
3. For every node **V** of **T**, the root **R** is the ancestor of **V**.

An ancestor of a node is either the node itself or an ancestor of the parent of the node. Conversely, we say that a node **V** is a **descendent** of a node **U** if **U** is an ancestor of **V**. For example, in the Figure 6.2 **B** is an ancestor of **D** and **D** is a descendent of **F**.

If node **u** is the parent of **V**, then we say that **V** is a **child** of **U**. In the Figure 6.2 node **A** is the parent of the nodes **B** and **C** and the node **B** is the child node of the node **A**.

Nodes that are children of the same parent are called as **siblings**. In the Figure 6.2 the nodes **F** and **G** are siblings.

A node is a **terminal node** if it has no children, and it is an **internal node** if it has one or more children. Terminal nodes are also called as **leaves**. In the Figure 6.2 nodes **E**, **F** and **G** are terminal nodes and the nodes **A**, **B**, **C** and **D** are internal nodes.

The **sub-tree** of **T** rooted at a node **v** is the tree consisting of all the descendent of **v** in **T** (including **v** itself) i.e., The nodes can be partitioned into **n** disjoint sets **T₁**, **T₂** **T_n** where each of these sets is a tree by itself. Then **T₁**, **T₂**.... **T_n** is called as **sub-trees**.

A tree is ordered if there is a linear ordering defined for the children of each node in such a way that we can identify children of a node as being the first, second, third, and so on. Such an ordering is determined by the use we wish to make for the tree, and it is usually indicated in drawing of a tree by arranging siblings left-to-right, corresponding to their linear relationship.

The number of sub-trees connected to a node is called the **degree** of the node. The degree of A is 2, B is 1, C is 1, D is 2 and F,G,E is 0.

The **degree of a tree** is the maximum degree of nodes in the tree. The tree in the Figure 6.2 has degree 2.

The level of node is defined as dept level at which a node is located. Levels a tree are set from the Root with Zero and incremented according to the dept of a given Tree . If a node is at level **p**, then its children are at level **p+1**.

The edges from A to B to D to G form a path of length 3.

The **depth** of a node **M** in the tree is the length of the path from the root of the tree to **M**. The **height** of a tree is one more than the depth of the deepest node in the tree.

The depth of G is 3; the height of this tree is 4.

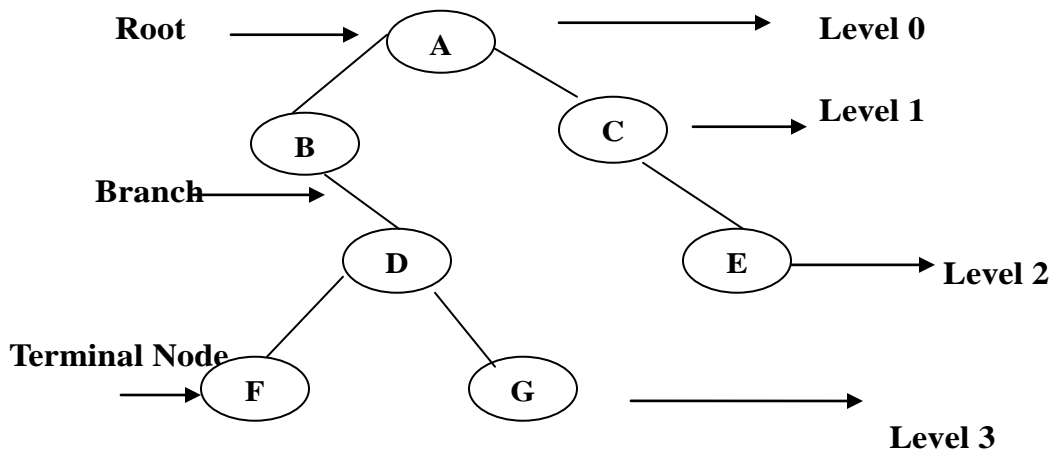


Figure 6.2 A General tree

1.3 BINARY TREE

A **binary tree** is an ordered tree in which each internal node can have a maximum of two children nodes connected to it. In a binary tree, the first child of an internal node is called the **left child**, and the second child is called the **right child**. The sub-tree rooted at the left and right of a child node are called the **left sub-tree** and the **right sub-tree**.

Definitions
Binary Tree: A tree data structure in which each node has at most two children.
Left Child: Node to the left of a given node in a binary tree; sometimes called left left son.
Left Subtree: All the nodes to the left of a given node in a binary tree.
Right Child: Node to the right of a given node in a binary tree, sometimes called right son.
Right Subtree: All the nodes to the right of a given node in a binary tree.

A binary tree has a number of applications. One of the applications is discussed in the example below.

Example 6.1: An arithmetic expression can be represented by a tree whose nodes are associated with variables or constants, and whose internal nodes are associated with one of the operators +, -, *, and /. Such an arithmetic expression tree is a binary tree, since each of the operators +, -, *, and / take exactly two operands. Also, note that the two children of a node associated with the - or / operator must be ordered.

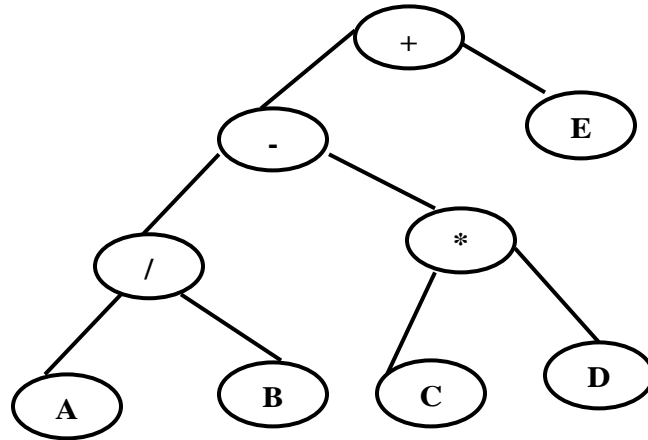


Figure 6.3 A Binary tree corresponding to the expression $A/B - C*D + E$

PROPERTIES OF A BINARY TREE

Binary trees have several remarkable properties, which include the following:

1. In a binary tree **T**, the number of external nodes is one more than the number of internal nodes.
2. The maximum number of nodes in a particular level **I** of a binary tree is given by the expression

$$2^I$$

3. The maximum number of nodes up to a particular level **I** of the binary tree is given by the expression

$$2^I - 1$$

The binary tree of depth K that has exactly $2^{K+1}-1$ nodes is called a **full binary tree** of depth K , this means that all internal nodes in the binary tree have exactly two child nodes and all the terminal nodes are at the same level.

Definition

Full binary tree: Binary tree in which all leaves are on the same level and every nonleaf node has two children.

1.4 COMPLETE BINARY TREES

Consider a binary tree of the Figure 6.4. Each node of tree has at most two children. Accordingly, one can show that level L of tree can have at most 2^{L-1} nodes.

Definition

Complete binary tree: Binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far left as possible.

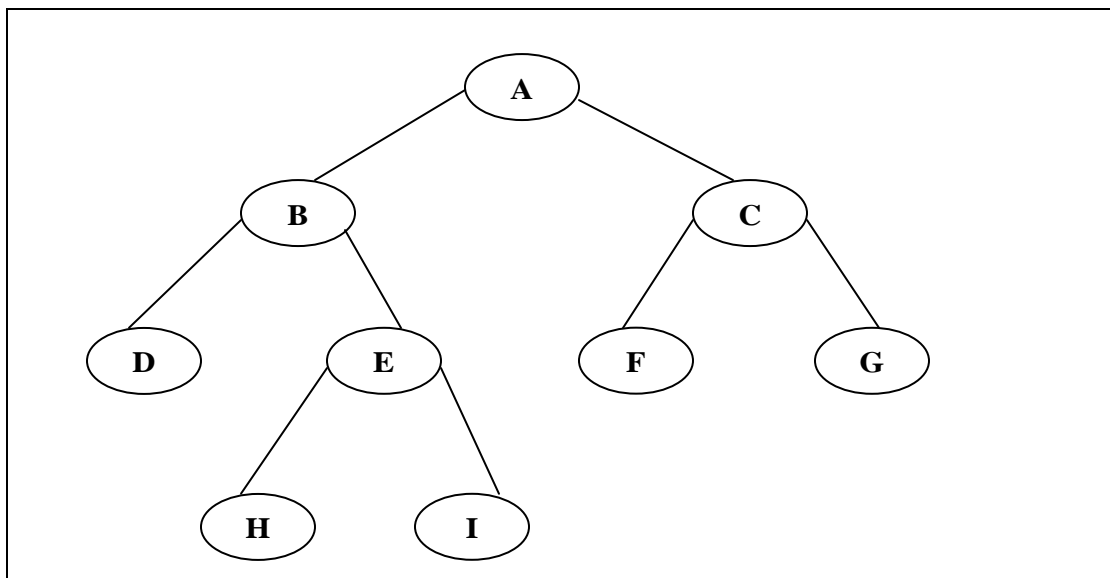


Figure 6.4 Complete binary tree with 9 nodes.

The tree is said to be complete if all its levels excepts possibly the last, have the maximum number of possible nodes. In a complete binary tree we observe that the terminal nodes need not be at the same level.

1.5 MEMORY REPRESENTATION OF BINARY TREE

Even though a binary tree represents a hierarchical structure it has to be converted to a suitable form so that it can be stored in the memory of the computer and all relationships are easily established. Two different representations are possible for a binary tree they are

1. Sequential representation using a single dimension array.
2. Linked list representation using a doubly linked list.

1.5.1 Sequential representation

In sequential representation arrays are used to store the binary tree. The nodes of a binary tree may be completely stored in one-dimensional array named **TREE** with the node number **I** being stored in **TREE (I)**. The following rules shows us how to easily determine the locations of the parent, left child and right child of any node **I** in the binary tree without explicitly storing any link information.

Dr.MBANZABUGABO Jean Baptiste(BE,MCA,MSc.SE,PhD.)

Dean, Faculty of Applied Science and Technology/Leading Technologies

@University Of Tourism, Technology and Business Studies(UTB)

E-mail: i_engineer@netzero.com, ict@utb.ac.rw /dean.it@utb.ac.rw

Skpe: j_engineer

WebSite: [www. utb.ac.rw](http://www.utb.ac.rw)

Cell: + 250 788 44 55 98 (Rwanda)