NEW SYLLABUS

SECOND YEAR B. Sc.
COMPUTER SCIENCE

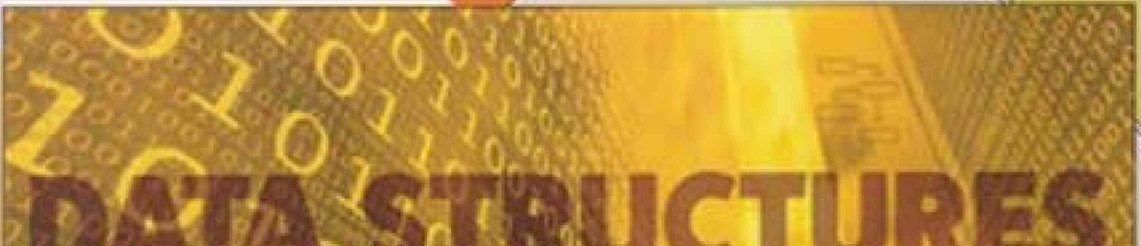
SEMESTER-III

DATA STRUCTURES AND ALGORITHMS-I

COMPUTER SCIENCE (CS 231) : Paper-I

CBCS
2 CREDITS

Dr. Ms. MANISHA BHARAMBE



NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

SPPU New Syllabus

A Book Of

DATA STRUCTURES AND ALGORITHMS-I

For S.Y.B.Sc. Computer Science : Semester - III (Paper - I)

[Course Code CS 231 : Credits - 2]

CBCS Pattern

As Per New Syllabus, Effective from June 2020

Dr. Ms. Manisha Bharambe

M.Sc. (Comp. Sci.), M.Phil. Ph.D. (Comp. Sci.)

Vice Principal, Associate Professor, Department of Computer Science

MES's Abasaheb Garware College

Pune

Price ₹ 340.00

NIRALI PRAKASHAN

ADVANCEMENT OF KNOWLEDGE

N5401

Data Structures & Algorithms - I **ISBN 978-93-90225-44-6**

First Edition : July 2020

© : Author

Published By :
NIRALI PRAKASHAN
Abhyudaya Pragati, 1312, Shivaji Nagar,
Off J.M. Road, Pune – 411005
Tel - (020) 25512336/37/39, Fax - (020) 25511379
Email : niralipune@pragationline.com

Polyplate

Printed By :
YOGIRAJ PRINTERS AND BINDERS
Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune - 411041
Mobile No. 9404233041/9850046517

Preface ...

I take an opportunity to present this Text Book on **"Data Structures and Algorithms-I"** to the students of Second Year B. Sc. (Computer Science) Semester-III as per the New Syllabus, June 2020.

The book has its own unique features. It brings out the subject in a very simple and lucid manner for easy and comprehensive understanding of the basic concepts. The book covers theory of Introduction to Data Structures and Algorithm Analysis, Array as a Data Structure, Linked List, Stack and Queue.

A special word of thank to Shri. Dineshbhai Furia, and Mr. Jignesh Furia for showing full faith in me to write this text book. I also thank to Mr. Amar Salunkhe and Mr. Akbar Shaikh of M/s Nirali Prakashan for their excellent co-operation.

I also thank Ms. Chaitali Takle, Mr. Ravindra Walodare, Mr. Sachin Shinde, Mr. Ashok Bodke, Mr. Moshin Sayyed and Mr. Nitin Thorat.

Although every care has been taken to check mistakes and misprints, any errors, omission and suggestions from teachers and students for the improvement of this text book shall be most welcome.

Author

■■■

Syllabus ...

1. Introduction to Data Structures and Algorithm Analysis (4 Hrs.)

1.1 Introduction

1.1.1 Need of Data Structure

1.1.2 Definitions - Data and Information, Data Type, Data Object, ADT, Data Structure

1.1.3 Types of Data Structures

1.2 Algorithm Analysis

1.2.1 Space and Time Complexity, Graphical understanding of the Relation between different Functions of n, Examples of Linear Loop, Logarithmic, Quadratic Loop etc.

1.2.2 Best, Worst, Average Case Analysis, Asymptotic Notations (Big O, Omega Ω, Theta θ), Problems on Time Complexity Calculation

2. Array as a Data Structure (10 Hrs.)

2.1 ADT of Array, Operations

2.2 Array Applications - Searching

2.2.1 Sequential Search, Variations - Sentinel Search, Probability Search, Ordered List Search

2.2.2 Binary Search

2.2.3 Comparison of Searching Methods

2.3 Sorting Terminology - Internal, External, Stable, In-Place Sorting

2.3.1 Comparison Based Sorting - Lower Bound on Comparison Based Sorting, Methods- Bubble Sort, Insertion Sort, Selection Sort, Algorithm Design Strategies - Divide and Conquer Strategy, Merge Sort, Quick Sort, Complexity Analysis of Sorting Methods

2.3.2 Non Comparison Based Sorting - Counting Sort, Radix Sort, Complexity Analysis

2.3.3 Comparison of Sorting Methods

3. Linked List (10 Hrs.)

3.1 List as a Data Structure, Differences with Array

3.2 Dynamic Implementation of Linked List, Internal and External Pointers

3.3 Types of Linked List - Singly, Doubly, Circular

3.4 Operations on Linked List - create, traverse, insert, delete, search, sort, reverse, concatenate, merge, Time Complexity of Operations

3.5 Applications of Linked List - Polynomial Representation, Addition of Two Polynomials

3.6 Generalized Linked List - Concept, Representation, Multiple-Variable Polynomial Representation Using Generalized List

4. Stack **(6 Hrs.)**

4.1 Introduction

4.2 Operations - init(), push(), pop(), isEmpty(), isFull(), peek(), Time Complexity of Operations

4.3 Implementation - Static and Dynamic with Comparison

4.4 Applications of Stack

4.4.1 Function Call and Recursion, String Reversal, Palindrome Checking

4.4.2 Expression Types - Infix, Prefix and Postfix, Expression Conversion and Evaluation (Implementation of Infix to Postfix, Evaluation of Postfix)

4.4.3 Backtracking Strategy - 4 Queens Problem (Implementation using Stack)

5. Queue **(6 Hrs.)**

5.1 Introduction

5.2 Operations - init(), enqueue(), dequeue(), isEmpty(), isFull(), peek(), Time Complexity of Operations, Differences with Stack

5.3 Implementation - Static and Dynamic with Comparison

5.4 Types of Queue - Linear Queue, Circular Queue, Priority Queue, Double Ended Queue (with Implementation)

5.5 Applications - CPU Scheduling in Multiprogramming Environment, Round Robin Algorithm

■■■

Contents ...

■■■

CHAPTER 1

Introduction to Data Structures and Algorithm Analysis

Objectives ...

- To know the Basic Concepts of Data Structures and Algorithm Analysis
- To study Data, Data Types, Data Objects etc.
- To study different Types of Data Structures
- To know the Concept of ADT
- To study Time and Space Complexities
- To learn Asymptotic Notations with Examples

1.0 INTRODUCTION

- The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed.
- Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm we might use, and choosing an algorithm affects the data structures we use.
- To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, program is represented as:
  Program = **Algorithm** + **Data Structure**
- While, the data structure is represented as:
  Data Structure = **Organized Data** + **Allowed Operations**
- In simple terms data structure can be defined as, a representation of data along with its associated operations.
- Data structure is the way of organizing and storing data in a computer system so that it can be used efficiently.
- Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem.
- An algorithm is a step-by-step procedure that provides solution to given problem.
- The analysis of algorithms is the process of finding the computational complexity of algorithms – the amount of time, storage, or other resources needed to execute them.
- In this chapter we are going to study the fundamental concepts related to data structures and algorithm analysis.

1.1 OVERVIEW OF DATA STRUCTURE

- Data structure contains the data object along with the set of operations which will be performed on them. Data structure also contains the information about the manner in which these data objects are related.
- A data structure is a mathematical or logical way of organizing data in memory where not only the data item stored inside them but also the relationship among each data is also considered.
- Data structures are a method of representing of logical relationships between individual data elements related to the solution of a given problem.

1.1.1 Need of Data Structure

- Data structure is a fundamental in computer science. Computer deals with manipulation of various kinds of data, wherefrom the concept of data structure comes.
  1. Data structures shows how data are stored in a computer so that operations can be implemented efficiently.
  2. Data structures are especially important when we have a large amount of information/data.
  3. Data structures gives conceptual and concrete ways to organize data for efficient storage and manipulation.
  4. A data structure helps us to understand the relationship of one data element with the other.
  5. With help of data structure data can be stored in such a manner that the program can easily access (retrieval) it.
  6. Various operations (insertion, deletion, traversing, searching, sorting etc.) can be performed easily as data is well organized and accessible with help of data structure.
  7. Effective use of principles of data structures increases efficiency of algorithms to solve problems like searching, sorting and so on.
- **Example:** Let us consider two ways of the data (phone numbers and names) organization's in the directory.
  1. The data is organized randomly. Then to search a person by name one has to linearly start from first name till the last name in the directory.
  2. If the data is organized by sorting on names alphabetically then the search is much easier.
- Instead of sequential (linear) search, we may search, in particular area. So because of organizing data in sorted order, search is faster.
- Hence, we can conclude that there is a strong relationship between the structuring of data (data structure) and operations to process the data. Data structure specification comes before programming language application.

Advantages of Data Structures:

1. Data structure allows easier processing of data.
2. It allows information stored on disk very efficiently.
3. Data structures are necessary for designing an efficient algorithm.
4. Data can be maintained more easily by encouraging a better design or implementation of real life problems.
5. Data structure is a secure way of storage of data.
6. Data structures allows processing of data on software system.

1.1.2 Definitions

- In this section we will study common definitions used in data structures.

1.1.2.1 Data

- The data plays an important role in programming and all computer programs involve applying operations on the data. Data refers to value or set of values. Example: Marks obtained by the student.
- Data is defined as, raw facts and figures before they have been processed. **OR** Data is defined as, a known fact that can be recorded and that have implicit meaning.
- Data is divided into atomic or non-atomic (composite) data as explained below:
  1. **Atomic Data:** It is a single or non-decomposable entity. For example, an integer 3241 is consider as a single integer value. Even though 3241 is decomposed into digits, the decomposed digits will not have the same characteristics of the original integer.
  2. **Non-atomic Data or Composite Data:** A data that we can be broken into subfields that have meaning is called composite data. For example, Date (dd/mm/yy) which can be divided into three subfields day, month and year.
- In short, a collection of raw facts and figures is called data. The word raw means that the facts have not yet been processed to get their exact meaning.
- Data is collected from different sources. It is collected for different purposes. Data may consist of numbers, characters, symbols or pictures etc.

1.1.2.2 Information

- When data is processed it becomes an information. In short, meaningful, logical and processed data is called as information.
- Information is a set of data which is processed in a meaningful way according to the given requirement.
- Information can be defined as, meaningful data or processed data. **OR** “the processed data is known as information.”
- The words data and information are often used interchangeably. However, they have different meanings.
- Data are raw facts from which the required information is produced. Information is an organized and processed form of data.

- Fig, 1.1 shows interrelation between data and information.

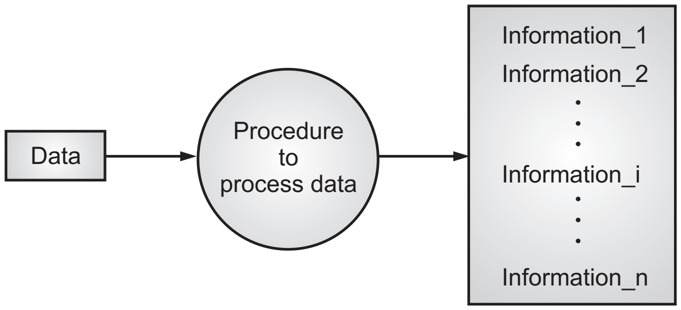


Fig. 1.1: Relation between Data and Information

1.1.2.3 Data Types

- The term data type is used to describe the type of data that variable may hold in the programming language.
- A data type is a collection of values along with a set of operations defined on those values.
- A data type determines the value that a variable can take and the operations that can be performed on that variable.
- A data type is defined as, "the set or collection of values and the set of operations that operate on those values". **OR** A data type is defined as, the data storage format that a variable can store a data to perform a specific operation.
- In simple terms, data type is represented as:

  Data Type = **Permitted Data Values** + **Operations**
- The Fig. 1.2 shows the data types in 'C' language.

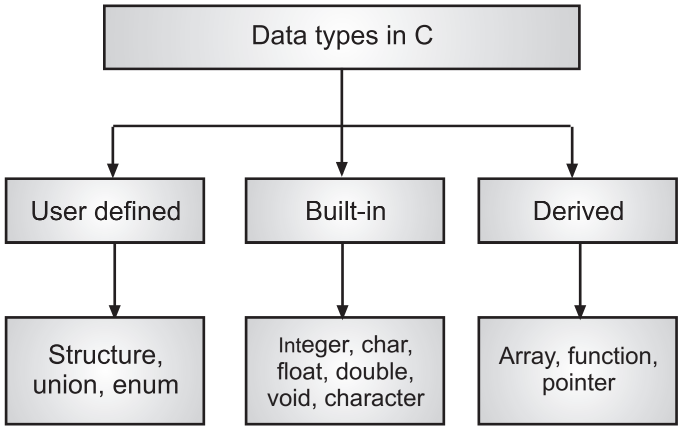


Fig. 1.2: Data Types in 'C'

Built-in Data Types:

- The data types provided by programming language are known as built-in data types.
- The primitive/basic data types in C language are the in-built data types provided by the C language itself. Thus, all C compilers provide support for these data types.

- The Table 1.1 shows basic or built-in data types in C and their sizes.

Table 1.1: Size and Range of Basic Data Types

Type	Bytes	Range
Char	1	– 128 to 127
unsigned char	1	0 to 255
int	2	– 32,768 to 32,767
unsigned int	2	0 to 65535
signed int	2	– 32768 to 32767
short int	2	– 32768 to 32767
long int	4	– 2147483648 to 2147483647
float	4	3.4 E – 3.8 to 3.4 E + 38
double	8	1.7 E – 308 to 1.7 E + 308

- Examples of int data type are, 324, – 34, 40 etc.
- Examples of char data types are, 'A', 'z' '1' etc.
- Examples of float data types are, 1.2, – 2.30, 123.45E6 etc.
- The void type specifies an empty set of values. It is used as the types returned by functions that generate no value.

User-Defined Data Types:

- Those data types which are defined by the user as per his/her requirements are called as user-defined data types.
- Some common used defined data type are enum, structure, union and so on.

1. Enumeration (enum):

- An enumeration type (also called enum) is a data type that consists of integral constants. To define enums, the enum keyword is used.
```
enum demo{ const1,const2, ……, constN };
```
- By default, const1 is 0, const2 is 1 and so on. You can change default values of enum elements during declaration (if necessary).
- For example, enum colors {black, green, red, blue, grey}
  In this example, colors are the name of the enumeration and black, green, red, blue, grey are the members of the enumeration which are called enumeration constants.
- Enumeration data type help in making the program listings are more readable and also help you to reduce programming errors.

2. Structure:

- A structure is a user defined data type in C. A structure is a tool for packing together logically related data items of different types (heterogeneous).

  For example:
```
struct employee
{
   char empname[20];
   int empcode;
   float basic;
   char address[20];
};
```

- Once, the structure is defined, we can declare structure variables and access its members by using either dot (.) operator or arrow (→) operator.

3. **Union:**

- Unions are very much similar to structures where both are used to group number of different elements.
- Unlike structures, the members of a union share the same storage area, even though the individual members may differ in types.

  For example:

```
union id
{
   char color[12];
   int size;
}
```

Derived Data Types:

- Derived data types are the data types that are derived from primitive data types. The C language provides derived data types such as arrays, pointers and so on.

1. **Arrays:**
   - Assign a single name to whole group (homogeneous).
   - For example, `int a[10];`
   - Above example defines an array of a size 10.
2. **Pointers:**
   - A pointer is a memory location that holds a memory address.
   - Pointers are more efficient in handling complex data structures and data tables.
   - Pointers increase the execution speed.
   - There are two special pointers operators * and &.

     For example:

     ```
     int *p;
     means p is a pointer to an integer.
     ```
   - We can initialize the pointer after declaring a pointer variable. Initializing the pointer using an assignment statement such as:

     ```
     P = &a; which causes p to point a.
     ```

1.1.2.4 Data Object [Oct. 16]

- A data object represents a container for data values – a place where data values are stored and later retrieved.
- A data object is characterized by a set of attributes. One of the most important of which is the data type.
- Data objects are the entities that actually contain the information. Data objects are the implementations of the data structure y the programmer/user.

- Data object refers to the set of elements say D, which may be finite or infinite. For example: The data object "alphabet" is define as:
  D = {A, B, … Z}
  The data object "integer" is defined as:
  D = { …–2, –1, 0, 1, 2, 3 …}
- Data object is runtime instance of data structure. Some of the data objects that exist during program execution are programmer defined such as variables, constants, arrays, files etc.
- System defined data objects are ordinarily during program execution without explicit specification by the programmer.
- Fig. 1.3 shows the simple variable data object. In Fig. 1.2 the data object x containing the data value 9.

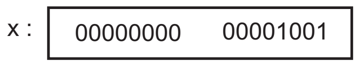


Fig. 1.3: Data Object

- A data object may be elementary if it contains a single value. If data object contains number of heterogeneous or homogeneous elements, then they referred as "data structures".

1.1.2.5 Abstract Data Type (ADT) [April 16, 17, 18; Oct. 17]

- Each programming language provides a set of built-in data types. However, these data types are not enough, since present day programming problems are more complex.
- Thus, there is a need for a structured data type which may be a combination or collection of basic data types with a set of properties and legal operations that may be performed on it by programmer. Programmers own data type is termed as Abstract Data Type (ADT).
- The data type made by user or programmer for performing any specific task of the program, called Abstract Data Type or ADT. Therefore, ADT is represented as:
  ADT = Type + Function Names + Behavior of Each Function
- With an ADT, users are not concerned with how the task is done but rather with what it can do. In other words, it allows user to abstract away from implementation detail.
- The generalization of operations with hidden implementation is known as abstraction. We abstract the essence of the process and leave the implementation details hidden.
- To know the abstraction concept, we consider the statement: "I put my lunch in my bag and went to school".
- What is meant by the term bag in this context? Most likely it is a backpack or satchel, but it could also be a hand bag, shopping bag, sleeping bag, body bag… (but probably not a bean bag).
- It does not actually matter. To parse the statement above, we simply understand that a bag is something that we can,
  1. put things in,
  2. carry places, and
  3. take things out.

  Such a specification is an Abstract Data Type.

Definition of ADT:

- An ADT (Abstract Data Type) is defined as, "a mathematical model of the data objects that make up a data type as well as the functions that operates on these objects". An ADT is the specification of logical and mathematical properties of a data type or structure.
- An ADT is a data declaration packaged together with the operations that are meaningful for the data type.
- In other words, we encapsulate the data and the operations on data and we hide them from a user. They are not concerned with the implementation details like space and time efficiency.
- An abstract data type includes:
  1. **Domain:** A collection of data.
  2. **Functions:** A set of operations on the data or subsets of the data.
  3. **Axioms:** A set of axioms, or rules of behavior governing the interaction of operations.
- Abstract Data Type (ADT) is a mathematical model with a set of operations defined on that model. For example, consider a rational number that can be expressed as the quotient of two integers.
- The operations on rational numbers are addition, multiplication and testing for equality. We can say that rational number together with addition, multiplication, equal operations is an example of ADT of rational number.

Example of ADT in 'C' Language:

- In 'C', we define data type named FILE, which is a abstract model of a file. We can perform operations on file such as, fopen() to open a file, fclose() to close file, fgetc() to read character from file, fputc() to write a character to the file.
- Definition of FILE is in stdio.h, the file input-output routines in 'C' runtime library enable us to view the file as a stream of bytes and to perform various operations on this stream by calling input-output routines. So, FILE definition together with functions operate on it, becomes a new data type.
- We do not know 'C' structure internally, instead, we rely on functions and macros that essentially hide inner details of FILE. This is called hiding.

Advantages of ADT: **[April 17, 18]**

1. ADT is reusable and ensures robust data structure.
2. Encapsulation ensures that data cannot be corrupted.
3. It reduces coding efforts.
4. The ADT is a useful guideline to implementers and a useful tool to programmers who wish to use the data type correctly.
5. Implementation details hiding.

1.1.2.6 Definition of Data Structure **[April 15, Oct. 17]**

- Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way.

- The logical or mathematical model of a particular organization of data in main memory is called as data structure.
- A Data Structure (DS) is defined as, a set of domains D, a designed domain $\in$ D, a set of functions F and a set of axioms A. The Triple (D, F, A) denotes the data structure d and it will usually have written as d, where,
  1. **Domain (D):** Denotes the data objects.
  2. **Function (F):** Denotes the set of operations that can be carried out on the data objects.
  3. **Axioms (A):** Denotes the properties and rules of the operations.
- Above definition is also called as Abstract Data Type (ADT). Implementation details of an ADT are hidden, that's why it is called abstract.
- To represent the mathematical model underlying an ADT, we use data structure, which is collection of variables, data types connected in different ways.
- The **basic operations that are performed on data structures** are as follows:
  1. **Insertion**: Insertion is used to adding a new data element in a data structure.
  2. **Deletion**: Deletion is used to remove a data element from a data structure if it is found.
  3. **Searching**: Searching is used to find out the location of the data element in a data structure.
  4. **Traversal**: Traversal of a data structure means processing all the data elements present in it.
  5. **Sorting**: It is used to arranging data elements of a data structure in a specified order is called sorting.
  6. **Merging**: Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

1.1.3 TYPES OF DATA STRUCTURES

- Data structures are divided into two types, Linear data structures and Non-Linear data structures as shown in Fig. 1.4.

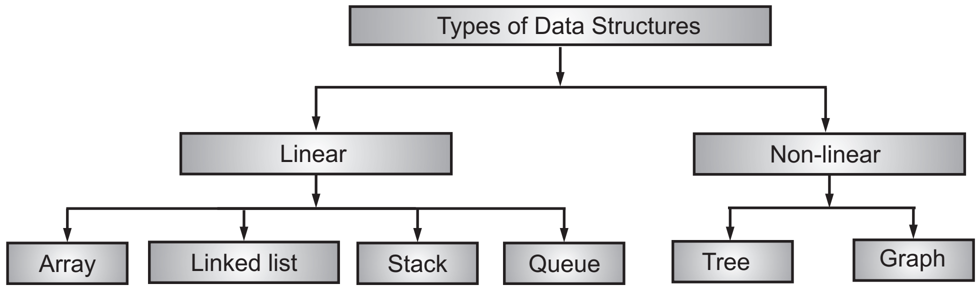


Fig. 1.4: Linear and Non-linear Data Structures

1. **Linear Data Structure**:

- A data structure is said to be linear if its elements form a sequence or a linear list. In this type of data structures, elements are accessed in sequential order but it is not compulsory to store all elements sequentially.

- Linear data structure traverses the data elements sequentially. In linear data structure, only one data element can directly be reached.
- Linear data structures include array, linked list, stack and queue as shown in Fig. 1.5.

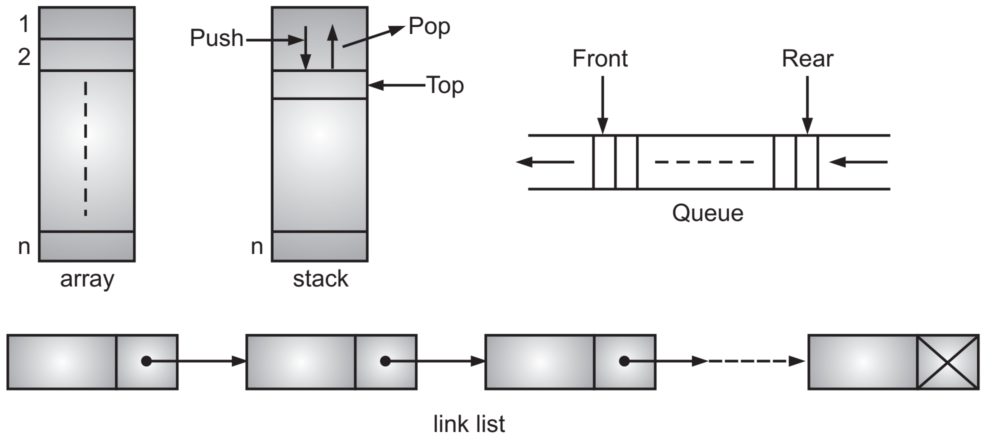


Fig. 1.5: Representation of Linear Data Structure

2. **Non-Linear Data Structure:**

- A data structure is said to be non-linear if its elements form a hierarchical classification where, data items appear at various levels.
- In non-linear data structures, every data element may have more than one predecessor as well as successor.
- In non-linear type of data structure, the elements do not form a sequence. Trees and Graphs are widely used non-linear data structures.

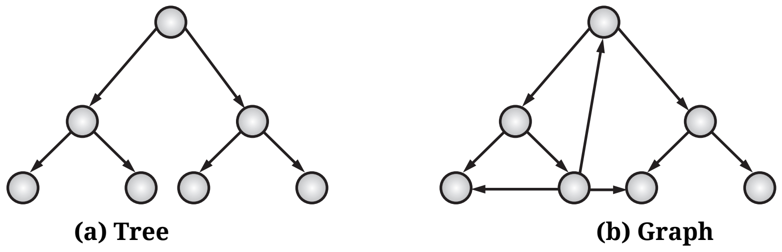


Fig. 1.6: Representation of Non-Linear Data Structure

Difference between Linear and Non-linear Data Structures:

Sr. No.	Linear Data Structures	Non-linear Data Structures
1.	A data structure is said to be linear if its elements form a sequence or a linear list.	A data structure is said to be non-linear if its every data item is attached to several other data items in a way of reflecting relationships.
2.	In linear data structures, data elements are organized sequentially.	In non-linear data structures, a data element are not organized sequentially.

contd. …

3.	They are easy to implement in the computer's memory.	They are difficult to implement in the computer's memory as compared to implementing linear data structures.
4.	Linear data structures are organized in a way similar to how the computer's memory is organized.	Non-linear data structures are organized in a different way than the computer's memory.
5.	Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other.	Non-linear data structures are constructed by attaching a data element to several other data elements in such a way that it reflects a specific relationship among them.
6.	Examples: Array, Linked List, Stack and Queue.	Examples: Tree, Graph.

1.2 ALGORITHM ANALYSIS

- Analysis of algorithms is the determination of the amount of time and space resources required to execute it.
- An algorithm analysis is a technique that is used to measure the performance of the algorithms.

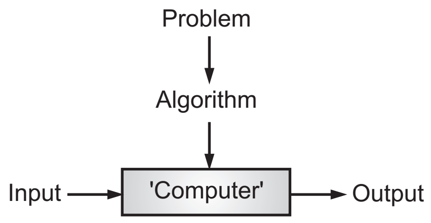


Fig. 1.7: Notion of Algorithm

- An algorithm is a finite sequence of instructions, each of which has a clear meaning and can be executed with a finite amount of effort in finite time.
- An algorithm refers to a finite set of steps, which when followed solves a particular problem.
- An algorithm can be defined as, a step-by-step procedure that provides solution to a given problem.
- **Example:** Algorithm for making a cup of tea:
  1. Put the teabag in a cup
  2. Fill the kettle with water
  3. Boil the water in the kettle.
  4. Pour some of the boiled water into the cup.
  5. Add milk to the cup.
  6. Add sugar to the cup.
  7. Stir the tea.
  8. Drink the tea.

- What we are doing is, for a given problem (preparing a tea), giving step-by-step procedure for solving it.
- Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

Characteristics of Algorithm:

- Following are the characteristics of algorithm:
  1. **Input:** An algorithm should accept zero or more inputs. Random number generation is an example of zero input.
  2. **Output:** An algorithm must produce at least one result.
  3. **Definiteness:** Each step is an algorithm must be clear and unambiguous. This helps the one who is following the steps to take a definite action.
  4. **Finiteness:** An algorithm must halt. Hence, it must terminate after a finite number of steps.
  5. **Effectiveness:** Each instruction must be sufficiently basic, then it can be done exactly and in finite time by a person using only paper and pencil.

Example: To find the largest of three numbers.

```
Step 1 : Start
Step 2 : Read three positive numbers, say x, y, z
Step 3 : If (x > y) and (x > z)
         then "x is a largest number" go to Step 5
Step 4 : If (y > z)
         then "y is largest number"
         else "z is largest number"
Step 5 : Stop
```

- We can find that above algorithm satisfies all the following criteria:
  1. **Input:** Here x, y and z are three positive number are input.
  2. **Output:** One output with largest number.
  3. **Definiteness:** All steps are clear and unambiguous.
  4. **Finiteness:** The algorithm terminates at step 5. Also at step 3 and step 4, if conditions are satisfied then algorithm terminates.
  5. **Effectiveness:** The operator ">" is sufficiently basic and steps are effective.
- Analysis of algorithm or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.
- Performance analysis of an algorithm is the process of calculating space required by that algorithm and time required by that algorithm.
- Performance analysis of an algorithm is performed by using the following measures:
  1. Space required to complete the task of that algorithm (Space Complexity). It includes program space and data space
  2. Time required to complete the task of that algorithm (Time Complexity).

Need of Algorithm Analysis:

- Let's take an example. I want to go from city 'X' to city 'Y', there can be many ways to go there. I can go there by bus, by train and also by bicycle. Depending on the availability and convenience we choose the one that suits us.
- Similarly, in computer science multiple algorithms are available for solving the same problem. (Example: Sorting problem has many algorithms like bubble, insertion, quick, merge and many more).
- Performance/algorithm analysis helps us to select the best algorithm from multiple algorithms to solve a problem.
- The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

1.2.1 Space Complexity [April 15, 17 Oct. 16]

- When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle.
- For any algorithm, memory is required for the following purposes:
    1. Memory required to store program instructions,
    2. Memory required to store constant values,
    3. Memory required to store variable values, and
    4. And for few other things.
- Space complexity of an algorithm can be defined as, "total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm".
- Generally, when a program is under execution it uses the computer memory for the following three reasons:
    1. **Instruction Space**: It is the amount of memory used to store compiled version of instructions.
    2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
    3. **Data Space:** It is the amount of memory used to store all the variables and constants.

Note: When we want to perform analysis of an algorithm based on its space complexity, we consider only data space and ignore instruction space as well as environmental stack. That means we calculate only the memory required to store variables, constants, structures, etc.

- To calculate the space complexity, we must know the memory required to store different data type values (according to the compiler).

Example 1	Example 2
`int square(int x) { return x*x; }`	`int sum(int A[], int n) { int sum = 0, i; for(i = 0; i < n; i++) sum = sum + A[i]; return sum; }`
In above piece of code, it requires 2 bytes of memory to store variable 'x' and another 2 bytes of memory is used for return value. (Assume int takes 2 bytes)	In above piece of code it requires 'n*2' bytes of memory to store array variable 'a[]'. 6 bytes of memory for integer variables 'sum', 'i' and 'n'. Another [2 bytes of memory for return value]. (Assume int takes 2 bytes)
That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'x'. This space complexity is said to be Constant Space Complexity.	That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the amount of memory depends on the input value of 'n'. This space complexity is said to be Linear Space Complexity.

1.2.2 Time Complexity

- Every algorithm requires some amount of computer time to execute it's instruction to perform the task. This computer time required is called time complexity.
- Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion.
- Time complexity of an algorithm can be defined as, "the total amount of time required by an algorithm to complete its execution."
- The total time taken by the algorithm or program is calculated using the sum of the time taken by each of executable statement in algorithm or program.
- Time required by each statement depends on:
  1. Time required for executing it once.
  2. Number of times the statement is executed.

  Product of above two gives time required for that particular statement.
- The total number of times a statement gets executed is define as its frequency count. Calculation of exact time may be difficult.
- Compute execution time of all executable statements. Summation of all is total time required for that algorithm or program.
- Generally, when we sum the frequency count of all the statements, we get polynomial.

- In analysis, we are interested in the order of magnitude of an algorithm i.e. we are interested in only those statements, which have the greatest frequency count.
- Consider following examples to calculate the time complexity.

1.
```
int sum(int a, int b)
{
  return a+b;
}
```
In above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires same amount of time i.e. 2 units.

If any program requires fixed amount of time for all input values then its time complexity is said to be **Constant Time Complexity**.

2.
```
int sum(int A[], int n)
{
  int sum = 0, i;
  for(i = 0; i < n; i++)
  sum = sum + A[i];
  return sum;
}
```
For the above code, time complexity can be calculated as follows...

int sum(int A[], int n) {	**Cost**	**Frequency**	**Total**
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1 + 1 + 1	(n + 1)	3n + 3
sum = sum + A[i];	2	N	2n
return sum;	1	1	1
}			
			5n + 5

- In above calculation:

  Cost is the amount of computer time required for a single operation in each line. Frequency is the amount of computer time required by each operation for all its operations. Total is the amount of computer time required by each operation to execute.
- So above code requires '4n+5' Units of computer time to complete the task. Here the exact time is not fixed. And it changes based on the n value. If we increase the n value, then the time required also increases linearly. Totally it takes '4n+4' units of time to complete its execution and it is linear time complexity.

1.2.3 Graphical Understanding of the Relation between different Functions of n

- In this section we will study graphical understanding of the relation between different functions of n.

1. **The Constant Function f(n) = C:**

- For any argument n, the constant function f(n) assigns the value C. It does not matter what the input size n is, f(n) will always be equal to the constant value C. The most fundamental constant function is f(n) = 1.
- The constant function is useful in algorithm analysis, because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers. Executing one instruction a fixed number of times also needs constant time only.
- Constant algorithm does not depend on the input size. Examples: arithmetic calculation, comparison, variable declaration, assignment statement, invoking a method or function.

2. **The Linear Function f(n) = n:**

- Linear function is useful in the analysis of algorithm. Given an input value n, the linear function f assigns the value n itself.
- This function arises in an algorithm analysis any time we do a single basic operation for each of n elements.
- For example, comparing a number x to each element of an array of size n will require n comparisons. The linear function also represents the best running time we hope to achieve for any algorithm that processes a collection of n inputs.
- Whenever n doubles, the running time doubles. Example: To display the elements of an array of size n.

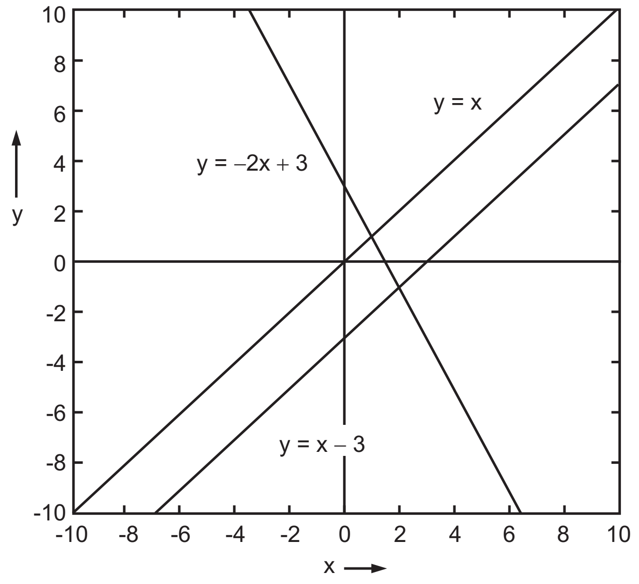


Fig. 1.8: Linear Function

3. **The Logarithm Function f(n) = logn:**

- It is one of the interesting aspects of the analysis of data structures and algorithms. The general form of a logarithm function is f(n) = $logb^n$, for some constant b > 1. This function is defined as follows:

  x = $logb^n$, if and only if $b^x = n$
- The value b is known as the base of the logarithm. Computing the logarithm function for any integer n is not always easy, but we can easily compute the smallest integer greater than or equal to $logb^n$, for this number is equal to the number of times we can divide n by b until we get a number less than or equal to 1. For example, log327 is 3, since 27/3/3/3 = 1. Likewise, log212 = 4, since 12/2/2/2/2 = 0.75 <= 1.
- The base-two approximating arises in the algorithm analysis, since a common operation in many algorithms is to repeatedly divide an input in half. In fact, the most common base for the logarithm in computer science is 2. We typically leave it off when it is 2.
- Logarithm function gets slightly slower as n grows. Whenever n doubles, the running time increases by a constant. Example: binary search.

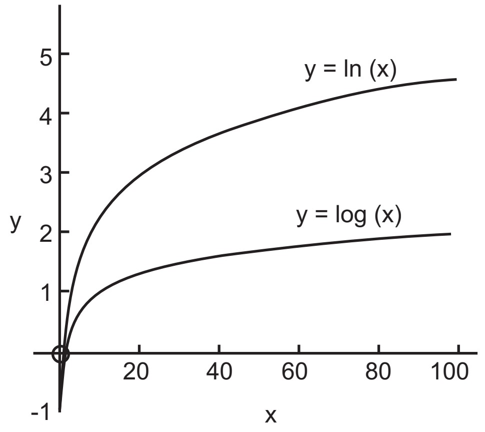


Fig. 1.9: Logarithmic Function

4. **The NLogN Function f(n) = nlogn:**

- This function grows a little faster than the linear function and a lot slower than the quadratic function (n^2).
- If we can improve the running time of solving some problem from quadratic to NLogN, we will have an algorithm that runs much faster in general. It scales to a huge problem, since whenever n doubles, the running time more than doubles.
- **Example:** Merge sort, which will be discussed later.

5. **The Quadratic Function f(n) = n^2:**

- It appears a lot in the algorithm analysis, since there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. In such cases, the algorithm performs n*n = n^2 operations.

- The quadratic function can also be used in the context of nested loops where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on. That is, the number of operations is 1 + 2 + 3 + ... + (n − 1) + n.
- For any integer n >= 1, we have 1 + 2 + 3 + ... + (n-1) + n = n*(n+1) / 2.
- Quadratic algorithms are practical for relatively small problems. Whenever n doubles, the running time increases fourfold.
- **Example:** Some manipulations of the n by n array.

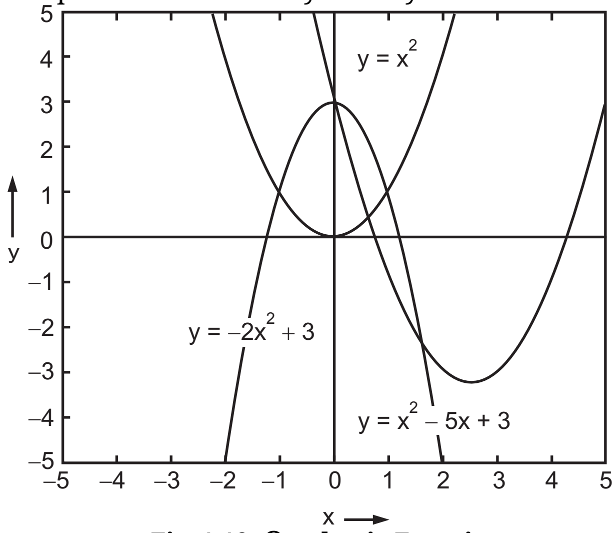


Fig. 1.10: **Quadratic Functions**

6. **Polynomials:**

- The functions we have learned so far can be viewed as all being part of a larger class of functions, the polynomials. A polynomial function is a function of the form:

  $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1x + a_0$

  where $a_0, a_1, \ldots, a_n$ are constants, called the coefficients of the polynomial, and an != 0. Integer n, which indicates the highest power in the polynomial, is called the degree of the polynomial.

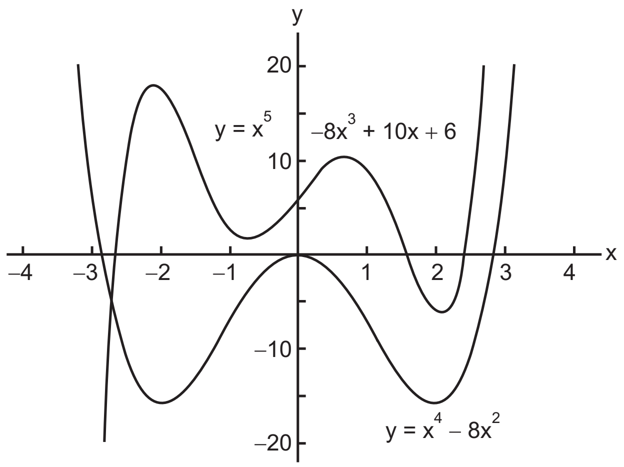


Fig. 1.11: **Polynomial Function**

7. The Exponential Function f(n) = b^n:

- In this function, b is a positive constant, called the base, and the argument n is the exponent. In the algorithm analysis, the most common base for the exponential function is b = 2.
- For instance, if we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the nth iteration is 2n.
- Exponential algorithm is usually not appropriate for practical use.

  Example: Towers of the Hanoi.

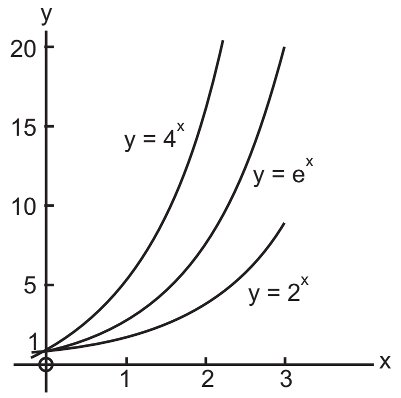


Fig. 1.12: Exponential Function

- Let's compare the growth rates for the above functions in the following table:

	Constant	Logarithmic	Linear	N-log-N	Quadratic	Exponential
N	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(2^n)
1	1	1	1	1	1	2
2	1	1	2	2	4	4
4	1	2	4	8	16	16
8	1	3	8	24	64	256
16	1	4	16	64	256	65536

1.2.4 Asymptotic Notations [Oct. 17]

- Whenever, we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required.
- So instead of taking exact amount of resource we represent that complexity in a general form (notation) which produces the basic nature of that algorithm. We use that general form (notation) for analysis process.
- Asymptotic notation of an algorithm is a mathematical representation of its complexity.

- Asymptotic notations represent the efficiency and performance of an algorithm in a systematic and meaningful manner.

Note: In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here, complexity may be space complexity or time complexity).

- For example: Consider the following time complexities of two algorithms:
  Algorihtm 1 : $5n^2 + 2n + 1$
  Algorihtm 2 : $10n^2 + 8n + 3$
- Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. 'n' value). In above two time complexities, for larger value of 'n' the term in algorithm 1 '2n + 1' has least significance than the term '$5n^2$', and the term in algorithm 2 '8n + 3' has least significance than the term '$10n^2$'.
- Here, for larger value of 'n' the value of most significant terms ($5n^2$ and $10n^2$) is very larger than the value of least significant terms (2n + 1 and 8n + 3). So for larger value of 'n' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.
- There are three types of Asymptotic Notations with respect to above case of time complexity and those are Big - Oh (O), Big - Omega (Ω) and Big - Theta (θ).
- Lets understand the relation between above mentioned cases of time complexity and asympotatic notation with help of an example.
- **For example:** Let us take the example of searching for an element in an unordered array of length N. The time complexity is given by:
  1. **Best Case:** Fastest time to complete, with optimal inputs chosen. Best case for 'searching' algorithm is element found at first position. Here, we need to do one comparison so time complexity of searching algorithm is Ω. In best case, Big - Omega notation (Ω) describes the best case of an algorithm time complexity. Big - Omega notation always indicates the minimum time required by an algorithm for all input values. Big - Omega notation is used to define the lower bound of an algorithm in terms of time complexity.
  2. **Worst Case:** Slowest time to complete, with pessimal inputs chosen. Worst case for searching algorithm is element found at last position or not found at all. Here, we need to do N comparison so time complexity of searching algorithm is O(N) in worst case. Big - Oh notation (O) describes the worst case of an algorithm time complexity. Big - Oh notation always indicates the maximum time required by an algorithm for all input values. Big - Oh notation is used to define the upper bound of an algorithm in terms of time complexity.
  3. **Average Case:** Average time to complete with random input chosen. Average case for searching algorithm is element found in between the list. Here we need to do N/2 comparison so time complexity of searching algorithm is (log N) in best case. Big - Theta (θ) notation is used to define the average bound of an algorithm in

terms of time complexity. Big - Theta notation always indicates the average time required by an algorithm for all input values. Theta notation describes the average case of an algorithm time complexity.

1.2.5 Big-Oh Notation (O) [April 16]

- Big-Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.
- Big-Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.
- Big-Oh Notation can be defined as follows:

  "Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C*g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n))".
- Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis.

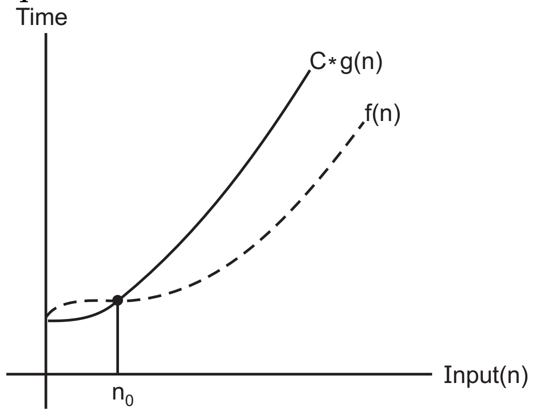


Fig. 1.13

- In above graph after a particular input value n_0, always C*g(n) is greater than f(n) which indicates the algorithm's upper bound.
- Example: Consider the following f(n) and g(n):

  f(n) = 3n + 2

  g(n) = n
- If we want to represent f(n) as O(g(n)) then it must satisfy f(n) <= C*g(n) for all values of C > 0 and n_0> = 1

  f(n) <= C*g(n)

  3n + 2 <= C*n
- Above condition is always true for all values of C = 4 and n >= 2. By using Big - Oh notation we can represent the time complexity as follows:

  3n + 2 = O(n)

- The most commonly used Big O descriptions are:
  1. **O(1)** describes an algorithm that will always execute in the same time (or space) regardless of the input size. If any algorithm having O(1) complexity then it is called as constant time algorithm.
  2. **O(logN)** takes a fixed additional amount of time each time the input size doubles. If any algorithm having O(logN) complexity, then it is called as logarithmic time algorithm.
  3. **O(N)** describes an algorithm whose performance will grow linearly and in direct proportion to the input size. If any algorithm having O(N) complexity, then it is called as linear time algorithm.
  4. **O(N^2)** represents an algorithm whose performance is directly proportional to the square of the input size. If any algorithm having O(N^2) complexity, then it is called as polynomial time algorithm.
  5. **O(2^N)** denotes an algorithm whose growth doubles with each addition to the input size. If any algorithm having O(2^N) complexity, then it is called as exponential time algorithm.
- Below is the table to understand the relation between input size and time complexity.

Input Size	Time to Complete				
	O(1)	O(logN)	O(N)	O(N^2)	O(2^N)
1	1	1	1	1	1
2	1	2	2	4	4
4	1	3	4	16	16
8	1	4	8	64	256
16	1	5	16	254	65536

1.2.6 Big-Omega Notation (Ω) [April 15, 18]

- Big - Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.
- That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.
- Big - Omega Notation can be defined as follows:

  "consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C*g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n))".

- Consider the following graph drawn for the values of f(n) and C*g(n) for input (n) value on X-Axis and time required is on Y-Axis.

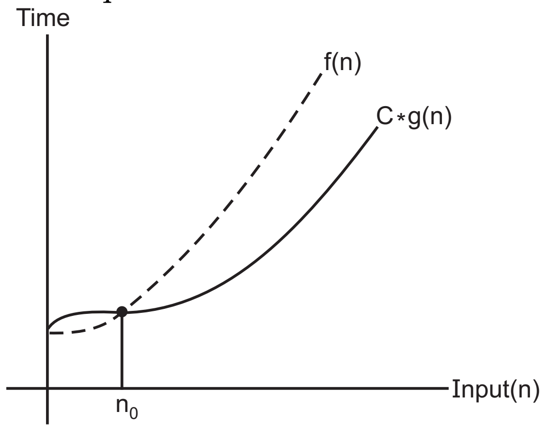


Fig. 1.14

- In above graph after a particular input value n_0, always C*g(n) is less than f(n) which indicates the algorithm's lower bound.
- Example: Consider the following f(n) and g(n):

  f(n) = 3n + 2

  g(n) = n

  If we want to represent f(n) as Ω(g(n)) then it must satisfy f(n) >= C*g(n) for all values of C > 0 and n_0>= 1

  f(n) >= C*g(n)

  3n + 2 <= C*n

  Above condition is always TRUE for all values of C = 1 and n >= 1. By using Big - Omega notation we can represent the time complexity as follows:

  3n + 2 = Ω(n)

1.2.7 Big-Theta Notation (θ) [Oct. 17]

- Big θ is the formal method of expressing the average bound of an algorithm's running time. It is measure of the average amount of time required for the algorithm to complete.
- Below is the definition of Big - Theta notation:

  Consider function f(n) is the time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) >= C_2 g(n) for all n >= n_0, C_1, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as θ(g(n)).
- Consider the graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis.
- In this graph after a particular input value n0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

- **Example:** Consider the following f(n) and g(n)...

  f(n) = 3n + 2

  g(n) = n
- If we want to represent f(n) as θ(g(n)) then it must satisfy C_1 g(n) <= f(n) >= C_2 g(n) for all values of C_1, C_2 > 0 and n0 >= 1.

  C_1 g(n) <= f(n) >= C_2 g(n)

  C_1 n <= 3n + 2 >= C_2 n
- Above condition is always TRUE for all values of C_1 = 1, C_2 = 4 and n >= 1. By using Big-Theta (θ) notation we can represent the time complexity as 3n + 2 = θ(n).

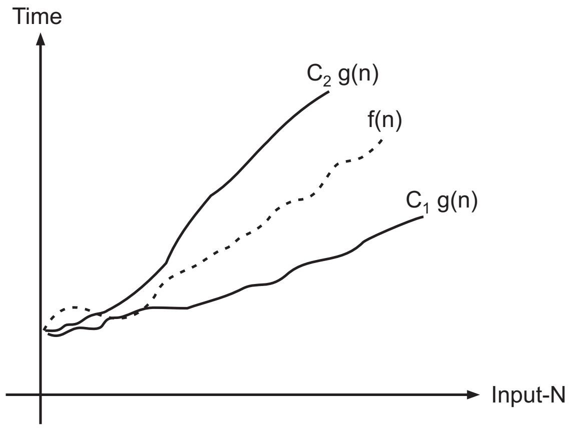


Fig. 1.15

1.2.8 Simple Algorithms and its Complexity as Examples

- **Following examples shows the space requirement:**

  (a) `double x[3]`

  Ans.: 24 bytes

  (b) `int max[10][10]`

  Ans.: 200 bytes

  (Assuming int takes 2 bytes).
- **Following examples shows the time requirement:**

  (a)
```
for(i = 0, i < n; i++)
  printf("%d \n", i);
```
  The output is numbers from 0 to n – 1. The condition (i < n) is checked n + 1 times, n times when it was true and once when it becomes false. The 2nd statement printf has been executed n times. Hence, total number of statements executed = n + 1 + n = 2n + 1.

  (b)
```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    x = x * y;
```
  Here, statement x = x * y executes n * n times. Hence, frequency count is n^2.

  (c) Find time complexity of n^m.

Algorithm	Frequency Count
(i) Read n, m	1
(ii) Let x = 1	1
(iii) For i = 1 to m	m + 1
(iv) x = x * n	m
(v) print x	1
(vi) stop	

Total frequency count = 2 + m + 1 + m + 1

= 2 m + 4, order of magnitude m

Time complexity = O(m)

(d) Find time complexity of nested for ...loop.

Algorithm	Frequency Count
`For i = 1 to n do`	N
`For j = i + 1 to n do`	$n \times (n - 1)$
`For k = j + 1 to n do`	$n \times (n - 1) \times (n - 2)$
`x = x + 1`	$n \times (n - 1) \times (n - 2)$
	Time complexity = O(n^3)

(e) Find time complexity of do ...while loop.

Algorithm	Frequency Count
`i = 1`	1
`do{`	–
`j++`	20
`if (i = = 20)`	20
`  break;`	1
`  i++`	20
`}while (i ≤ n);`	20
	Time complexity = O(1)

(f) Find time complexity of for loop.

Algorithm	Frequency Count
`x = n/2`	1
`for (i = 0, i + x < x; i++)`	n/2 + 1
`{`	–
`  k++;`	n/2
`  j++;`	n/2
`}`	–
	Time complexity = O(n/2)

(g) Compute time complexity for the following algorithm:

```
if(x>y)
{
  x=x+1;
}
else
{
for(i=1;i<=N;i++)
{
  x=x+1;
}
}
```

Ans.:

	Frequency	
	x > y	**x < = y**
`if(x > y)`	1	1
`{`	1	–
`    x=x + 1;`	–	–
`}`	–	–
`else`	–	–
`{`	–	–
`    for(i = 1; i<=N; i++)`	–	N + 1
`    {`	–	N
`        x = x+1;`	–	–
`    }`	–	–
`}`	–	–
Total	2	2N + 2

PRACTICE QUESTIONS

Q.I Multiple Choice Questions:

1. The logical or mathematical model of a particular organization of data is called ______.
   (a) Data Structure (b) Abstract Data Type
   (c) Primitive Data Type (d) Algorithm
2. In an ADT ______.
   (a) Data are declared
   (b) Operations are defined
   (c) Data and Operations are encapsulated
   (d) All of the above
3. A data structure is an aggregation of ______.
   (a) Atomic data
   (b) Composite data
   (c) Atomic and Composite data into a set with define relationship
   (d) None of the above
4. ______ is / are the linear data structures.
   (a) Array (b) Linked list
   (c) String (d) All (a), (b) and (c)
5. ______ is a non-linear data structures.
   (a) Linked list (b) Stack
   (c) Graph (d) Queue

6. Time complexity of a program refer to ______.
   (a) Complexity involved with the input time of a program
   (b) Complexity involved in space mission and control
   (c) Amount of time a program needs to run for completion/execution.
   (d) None of the above
7. Space complexity of an algorithm is the maximum amount of _______ required by it during execution.
   (a) Memory space (b) Operations
   (c) Time (d) None of the above
8. An algorithm that indicates the amount of temporary storage required for running the algorithm, i.e., the amount of memory needed by the algorithm to run to completion is termed as _______.
   (a) Big Oh O(f) (b) Big Omega Ω(f)
   (c) Space complexity (d) Time complexity
9. To verify whether a function grows faster or slower than the other function, we have some asymptotic or mathematical notations, which is/are ______.
   (a) Big Omega Ω(f) (b) Big Oh O(f)
   (c) Both (a) and (b) (d) None of these
10. A function in which f(n) is Ω (g(n)), if there exist positive values k and c such that f(n)>=c*g(n), for all n>=k. This notation defines a lower bound for a function f(n):
   (a) Big Oh O(f) (b) Big Omega Ω(f)
   (c) Both (a) and (b) (d) None of the above
11. An algorithm that requires ______ operations to complete its task on n data elements is said to have a linear runtime.
   (a) $2n + 1$ (b) $3n^2 + 3n + 2$
   (c) $n^3 + 9$ (d) 10
12. The complexity of adding two matrices of order m*n is
   (a) $m + n$ (b) mn
   (c) m^2n^2 (d) $\log (m + n)$
13. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?
   (a) X will always be a better choice for small inputs
   (b) X will always be a better choice for large inputs
   (c) Y will always be a better choice for small inputs
   (d) X will always be a better choice for all inputs
14. The function that derives the running time of an algorithm and its memory space requirements for a given set of inputs is referred as algorithm ______.
   (a) analysis (b) notations
   (c) complexity (d) All of these

15. In simple term, data structure is the data definition while data ______ are its implementations.
    (a) objects (b) type
    (c) complexity (d) All of these
16. Using which notation, we can compute the average possible amount of time that an algorithm will take for its completion.
    (a) Omega (b) Theta
    (c) Oh (d) All of these
17. What is the time complexity of following code:
```
int a = 0;
for (i = 0; i < N; i++)
{
    for (j = N; j > i; j--)
    {
    a = a + i + j;
    }
}
```
    (a) O(N) (b) O(N*log(N))
    (c) O(N * Sqrt(N)) (d) O(N*N)
18. What is the time, space complexity of following code:
```
int a = 0, b = 0;
for (i = 0; i < N; i++)
{
    a = a + rand();
}
for (j = 0; j < M; j++)
{
    b = b + rand();
}
```
    (a) O(N * M) time, O(1) space (b) O(N + M) time, O(N + M) space
    (c) O(N + M) time, O(1) space (d) O(N * M) time, O(N + M) space
19. What is the time complexity of following code:
```
int a = 0, i = N;
while (i > 0)
{
     a += i;
     i /= 2;
}
```
    (a) O(N) (b) O(Sqrt(N))
    (c) O(N/2) (d) O(logN)

Answers

1. (a)	2. (d)	3. (c)	4. (d)	5. (c)	6. (c)	7. (a)	8. (c)	9. (c)	10.(b)
11. (a)	12. (b)	13. (b)	14. (c)	15. (a)	16. (b)	17. (d)	18. (c)	19. (d)	

Q.II Fill in the Blanks:

1. The ______ plays an important role in programming, as all operations are carried out on data.
2. A data structure and its operations can be packaged together into an entity called an ______.
3. ______ is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
4. Data ______ represents an object having a data
5. A ______ data structure is said to be linear if its elements form a sequence or a linear list.
6. ______ complexity is the measure of the running time of an algorithm for a given set of inputs.
7. ______ complexity is the measure of the amount of memory space required by an algorithm for its complete execution, for a given set of inputs.
8. When data is processed, organized and presented in a context to make it useful is called ______.
9. Using ______ notation, we can compute the maximum possible amount of time that an algorithm will take for its completion.

Answers

1. Data	2. ADT	3. Algorithm	4. Object	5. Linear
6. Time	7. Space	8. Information	9. Big-O	

Q.III State True or False:

1. The data is raw, unorganized facts that need to be processed however after processing, it becomes information.
2. A data structure as a way of organizing data that specifies:
   (i) A set of data element i.e. a data object, and
   (ii) A set of operations which are applied to this data object.
3. The space complexity of a program or algorithm is the amount of time required by the program statement to execute.
4. The space complexity of an algorithm or program is the amount of memory that it needs to run to completion.
5. Using omega notation, we can compute the minimum possible amount of time that an algorithm will take for its completion.
6. Asymptotic notation describes complexity in terms of three common measures namely, Best case (fastest possible), Worst case (slowest possible time) and Average case (average time).
7. Linked list is a data structure use for storing data in the form of a list.
8. Stack is a non-linear data structure.

9. Derived data types in C include integer (int), floating point (float), character (char) and void.
10. Data objects can access the methods and information stored in the data structure to store, process and retrieve information.
11. Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
12. Data object are a method of representing of logical relationships between individual data elements related to the solution of a given problem.

Answers

1. (T)	2. (T)	3. (F)	4. (T)	5. (T)	6. (T)
7. (T)	8. (F)	9. (F)	10. (T)	11. (T)	12. (F)

Q. IV Answer the following Questions:

(A) Short Answer Questions:

1. What is data structure?
2. List types of data structures.
3. What is data object?
4. What is data type?
5. List types of data types in C language.
6. What is meant asymptotic notations?
7. What is algorithm?
8. What is ADT?
9. What is algorithm analysis?
10. List various types of asymptotic notations.
11. What is complexity of algorithm?
12. What are the examples of linear data structures?
13. Define the following terms:
    (i) Data type.
    (ii) Data object.
    (iii) Algorithm.
    (iv) Time complexity.
    (v) Space complexity.
    (vi) Big-Oh notation.
    (vii) Omega notation.
14. List operations on data structures.

(B) Long Answer Questions:

1. Define data structure. Why we need data structure?
2. With the help of example explain the term data object.
3. Describe the term ADT with its advantages.

4. Why analysis of an algorithm is important?
5. What are different characteristics of an algorithm.
6. What are the advantages of data structure?
7. Compare linear and non-linear data structure.
8. Write short note on: Types of data structure.
9. With suitable example describe ADT.
10. Write note on: Space complexity.
11. Explain time complexity with an example.
12. Describe Big Oh with example.
13. With the help of example describe Big Omega notation.
14. Explain Theta notation with example.

UNIVERSITY QUESTIONS AND ANSWERS

April 2015

1. Define data structure. **[1 M]**

Ans. Refer to Section 1.1.

2. Define Omega Notation (Ω). **[1 M]**

Ans. Refer to Section 1.2.6.

3. Define the term Space complexity. **[1 M]**

Ans. Refer to Section 1.2.1.

April 2016

1. Define the term Big O notation. **[1 M]**

Ans. Refer to Section 1.2.5.

2. What are the advantages of ADT? **[1 M]**

Ans. Refer to Section 1.1.2.5.

October 2016

1. Define Data Object. **[1 M]**

Ans. Refer to Section 1.1.2.4.

2. Define Space Complexity. **[1 M]**

Ans. Refer to Section 1.2.2.

April 2017

1. What is the component of space complexity? **[1 M]**

Ans. Refer to Section 1.2.1.

2. What are the advantages of ADT? **[1 M]**

Ans. Refer to Section 1.1.2.5.

3. What are different asymptotic notations? **[1 M]**

Ans. Refer to Section 1.2.4.

October 2017

1. What is meant by an abstract data type? **[1 M]**

Ans. Refer to Section 1.1.2.5.

2. Define Theta notation (θ). **[1 M]**

Ans. Refer to Section 1.2.7.

3. Define the term Data structure. **[1 M]**

Ans. Refer to Section 1.1.2.6.

April 2018

1. Define omega (Ω) notation. **[1 M]**

Ans. Refer to Section 1.2.6.

2. What is ADT? What are the advantages of ADT? **[1 M]**

Ans. Refer to Section 1.1.2.5.

■■■

CHAPTER 2

Array as a Data Structure

Objectives ...

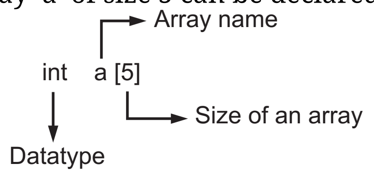
- ➢ To learn Array Data Structure with its Representation
- ➢ To study different Types of Arrays
- ➢ To learn different Applications of Arrays
- ➢ To study Memory Representation of One-Dimensional and Multi-Dimensional Array
- ➢ To study Basic Concepts of Sorting with Types
- ➢ To understand different Sorting and Searching Methods with their Time Complexity

2.0 INTRODUCTION TO ARRAY

- An array is a data structure for storing more than one data item that has a similar data type. In simple words, an array is a collection of homogeneous (similar type) data elements.

Definition of Array:

- An array is a finite ordered collection of homogeneous data elements which provides random access to the elements.
- The array elements share a common name. Declaration of an array needs three things: array data type, name and size.
- **Example:** In 'C' integer array ‘a’ of size 5 can be declared as follows:



- The Fig. 2.1 shows the array of integers with capacity five.

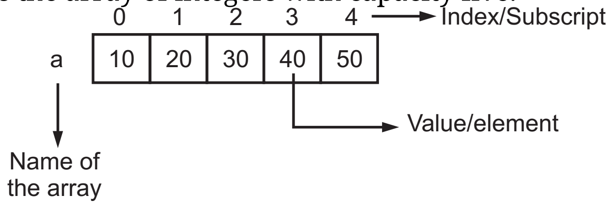
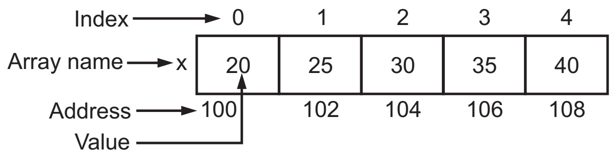


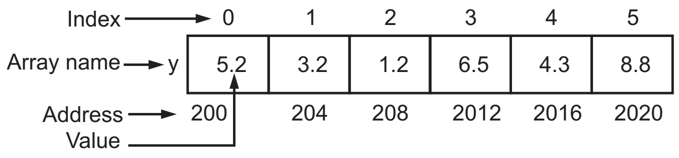
Fig. 2.1: Array of Integers

- Array allows to store collection of information in one organized place. It is used when program have to handle large amount of data.
- All elements are stored in continuous memory location. Each element is stored at a specific location.
- Location is called as an index or subscript. An index or subscript of an array is integer constant. It begins with 0 and ends with n-1, where n is the size of the array.
- The size and type of an array cannot be changed after its compilation; hence array are called as static data structure.

Array Initialization and Representation:

- Arrays can be initialized at the time of declaration.
- **Example:**
  1. int x[5]={20, 25, 30, 35, 40}
  2. float y[6] = {5.2, 3.2, 1.2, 6.5, 4.3, 8.8}
  3. char alpha[3] = {‘A’, ‘B’, ‘C’}
- The above array can be represented as follows:

1.


2.


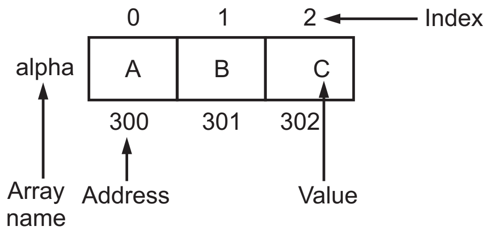
3.


Fig. 2.2: Representation of Array

- In 'C', the amount of storage bits per element depends upon the data type of the array. In 'C', size are as follows:

  Integer → 2 or 4 bytes (see address field of Fig. 2.2 (a)).

  Float → 4 bytes (see address field of Fig. 2.2 (b)).

  Char → 1 byte (see address field of Fig. 2.2 (c)).

Terminology of Array Data Structure:

1. **Size of Array:** Number of elements in an array is called as the size of array. Size of array is also called as length or dimension. Dimension once defined cannot be modified after compilation. Hence, arrays are called as static data structures the size is always mentioned in the bracket ([ ]).
2. **Base:** The address of the first element (0^{th}) element is base address. In other words, base of an array is the address of memory location where the first element of the array is located. Base of array is defined when the program is executed. Value of base varies at every run of the program. It is not defined by the programmer.
3. **Type:** Types refer to data type. Data type represents the kind of information i.e. type of data to be stored is an array.
4. **Index:** An element of an array is referenced by a subscript like A[i] and also as Ai. This subscript is known as index. Index of first element is always 0.
5. **Word:** Word indicates the space required for an element. In each memory location, a computer can store a data piece.
6. **Range:** It is index of an array, i.e. the value of x varies from the lower bound to the upper bound while writing or reading elements from an array. For example, for num[10] the range of the index is 0 to 9.

Advantages of Arrays:

1. Arrays helps to arrange the data (sorting) items in particular order (ascending/ descending).
2. Data items searching is faster using arrays.
3. Arrays saves the memory space.
4. Arrays permit efficient random access in constant time O(1).
5. Ordered lists such as polynomials are most efficiently handled using arrays.
6. Arrays are useful to form the basis for several more complex data structures, such as heaps, hash tables and can be used to represent strings, stacks and queues.
7. Arrays are easily managing large amount of information.
8. It is used to represent multiple data items of same type by using only single name.

Disadvantages of Arrays:

1. Array is rigid data structure.
2. Array cannot be dynamically resized the most languages.
3. The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.
4. Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or reduced.

Applications of Arrays:

1. **Array can be used for Sorting Elements:** We can store elements to be stored in an array and then by using different sorting techniques we can sort the element. Different sorting techniques like bubble sort, insertion sort, selection sort and so on.

2. **Array can be used in CPU Scheduling:** CPU scheduling is generally managed by Queue. Queue can be managed and implemented using the array. Array may be allocated dynamically i.e., at run time.
3. **Array can be used in Recursive Function:** When the function calls another function or the same function again then the current values are stores onto the stack and those values will be retrieved when control comes back.
4. **Array can perform Matrix Operation:** Matrix operations can be performed using the array. We can use 2-D array to store the matrix. Matrix can be multi-dimensional.
5. Arrays are used to **implement mathematical vectors and matrices**, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records.
6. Arrays are used to **implement other data structures**, such as heaps, hash tables, deques, queues, stacks, strings, and so on.
7. Arrays can be used to **determine partial or complete control flow** in programs.

Types of Arrays:

- Arrays are indexed collections of data. The data can be of either a built-in type or a user-defined type.
- Arrays are classified into two types namely Single Dimensional Array/One-Dimensional Array and Multi-Dimensional Array.

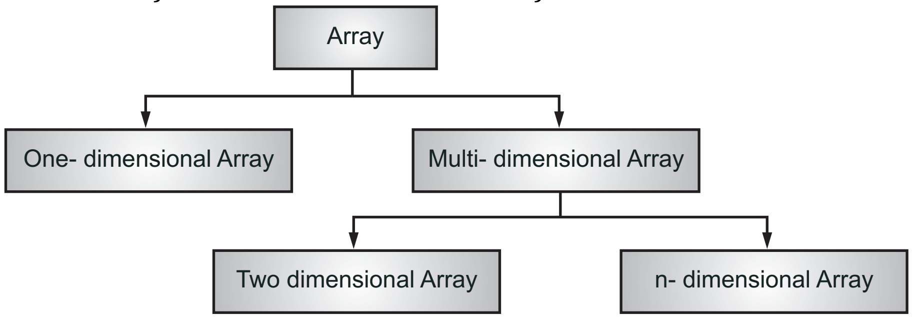


Fig. 2.3: Types of Array

1. **Single Dimensional Array/One Dimensional Array:**

- The simplex form of an array is one-dimensional array. The array is given a name and its elements are referred to by their subscripts or indices.
- One-dimensional array or linear array requires only one index to access an element. The general **form/syntax** of declaration of 1D Arrays is given below:

```
data_type variable_name[size];
```

- Here, data_type specifies the type of element that will be contained in the array such as int, float or char and size specifies the maximum number of elements that can be stored inside the array.
- For example, `char name[10];` declares name as a character array (string) variable that can hold a maximum of 10 characters.

- The general **form/syntax** of initialization of arrays is given below:

```
data_type array_name[size]={list of values};
```

- The values in the list are separated by commas. For example, consider the following statement,

```
int number[3]={0,0,0};
```

- Above example declare an array of size 3 and will assign zero to each element. If the number of values in the list is less than the size of array, then only those elements will be initialized and the remaining elements will be set to zero automatically.
- An array is represented in the memory by using a sequential mapping i.e. every element is at fixed distance apart. The Fig. 2.4 shows the memory representation of one-dimensional array.

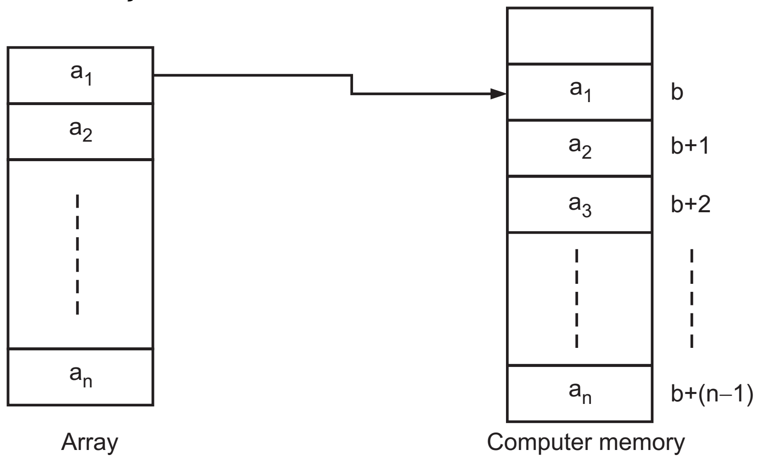


Fig. 2.4: Memory representation of array N

- Where b is the base address, which is the address of the first element. Using this base address the computer calculates the address of any element of an array by using the following formula:

  Address ($i^{th)}$ element = Base address + (offset of the i^{th} element from base address)

  Whereh offset is computed as:

  Offset of i^{th} element = (no. of element before i^{th} element) * (size of each element)

Example 1: If int a[5] is an array of 5 elements. Each integer is of 2 bytes and base address is 100, then find the address of 3rd and 5th element.

Address of 3rd element = Base address + (offset of the i^{th} element from base address)

a [2] (i.e. 3rd element) =100 + ($\underbrace{2}_{\text{Offset}} \times \underbrace{2}_{\text{size}}$)

= 104

a [4] = 100 + (4 * size)

(i.e. 5th element) = 100 + (4 × 2)

= 108

2. **Two Dimensional Array:**

- An array with two subscripts is known as two dimensional array. Two dimensional arrays also known as double dimension arrays.
- The two dimensional array is also referred as a matrix. It is a combination of rows and columns.
- A two dimensional array is a collection of elements placed in m rows and n columns. The Fig. 2.5 shows the representation of 2D array memory. The array is A[m][n] containing m rows and n columns.

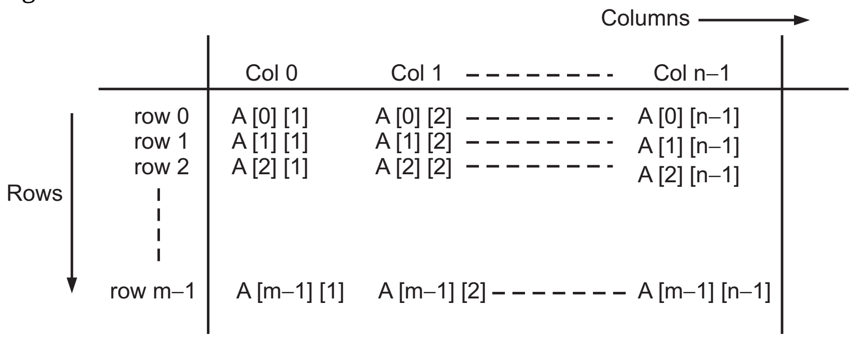


Fig. 2.5: Logical View of Array A[m] [n]

- A 2D array is a logical data structure useful in programming. A 2D array differentiates between the logical and physical view of data.
- The **syntax** for declaration of two dimensional array is given below:

```
data_type array_name [row_size] [col_size];
```

Example: `int mark [2][2];`

- Two-dimensional array initialized by following **syntax**:

```
<data_type> <array_name> [row_subscript] [column];
```

Example:
```
int a [3] [3] = {1, 2, 3
                 5, 6, 7
                 8, 9, 10};
```

Memory Representation of 2D array:

- Let A be a two dimensional array m × n. The elements in the array A will be stored either column by column i.e. column major representation or row by row i.e. row major representation.

1. **Row-major Representation:**

- All elements of a matrix or 2D array get stored in memory in a linear fashion. Hence, 2D array can be considered as 1D array in memory.
- The representation of 2D array A[2][3] in memory is shown in Fig. 2.6.

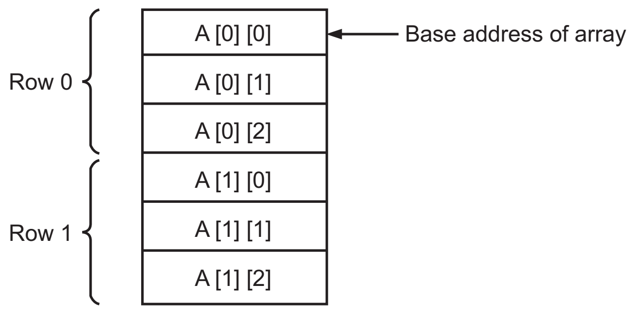


Fig. 2.6: Representation of 2D Array in Row-major Order

Let us calculate the address of element in 2D array.

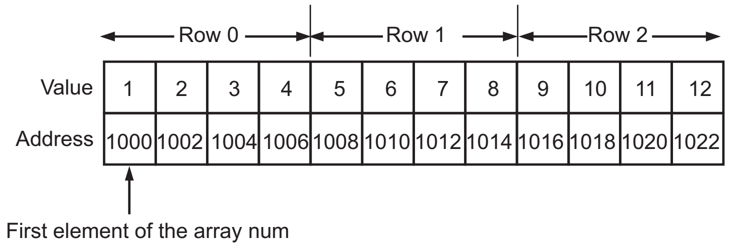
- Formula to calculate address of (i, j)th element of 2-D array of $m \times n$ dimension is:

```
arr[i][j] = baseAddress + [(i x no_of_cols + j) x size_of_data_type]
```

- Where, arr is a two dimensional array and i and j denotes the ith row and jth column of the array.

Example 2: To calculate address of num[2][3] of integer array of size 3×4 with base address 1000.

```
int num[3] [4] = {
    {1, 2, 3, 4}
    {5, 6, 7, 8}
    {9, 10, 11, 12}
}
```



	Row 0				Row 1				Row 2			
Value	1	2	3	4	5	6	7	8	9	10	11	12
Address	1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	1020	1022

Fig. 2.7: Row-Major Representation of 2D Array in Memory

In the case of num array the baseAddress = 1000, no_of_cols = 4 and size_of_data_type = 2 i.e. each element is taking 2 bytes in memory.

So, we can compute the memory address location of the element num[2][3] as follows. Address of element 12 which is present at cell 2,3

$$\begin{aligned} \text{num}[2][3] &= \text{baseAddress} + [(i \times \text{no\_of\_cols} + j) \times \text{size\_of\_data\_type}] \\ &= 1000 + [(2 \times 4 + 3) \times 2] \\ &= 1000 + [(8 + 3) \times 2] \\ &= 1000 + 22 \\ &= 1022 \end{aligned}$$

Example 3: To calculate address of A(1, 2) of integer array of size 2 × 3 with base address 100.

$$\begin{aligned}\text{Address } (1, 2) &= \text{baseAddress} + [(i \times \text{no\_of\_cols} + j) \times \text{size\_of\_data\_type}]\\ &= 100 + [(1 \times 3 + 2) \times 2]\\ &= 100 + [5 \times 2]\\ &= 110\end{aligned}$$

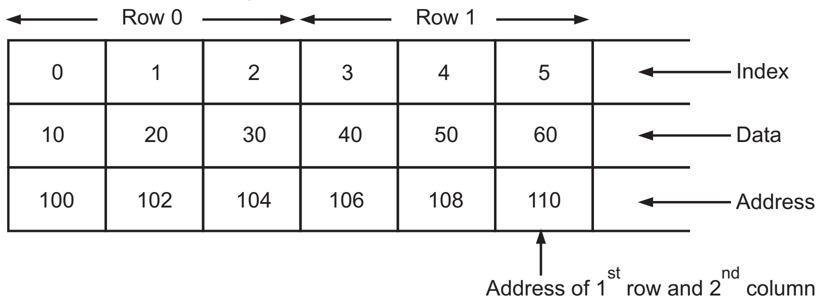


Fig. 2.8

2. Column-major Representation:

- Column-major order is a similar method of flattening arrays onto linear memory, but the columns are listed in sequence.
- The Fig. 2.9 shows the 2D array A[2][3] representation in memory.

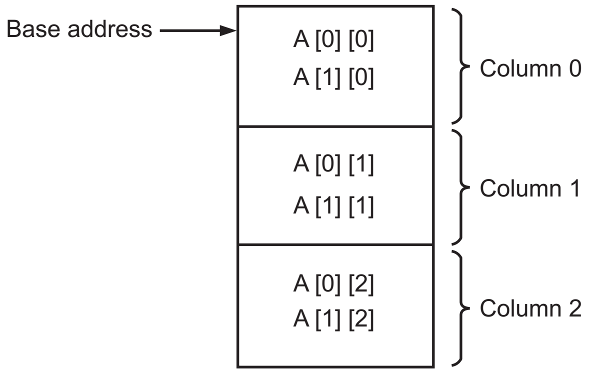


Fig. 2.9: Representation of 2D Array in Column Major Order

- To find the address of (i, j)th element in the 2-D array with column major is,

```
arr[i][j] = baseAddress + [(j x no_of_rows + i) x size_of_data_type]
```

where, arr is a two dimensional array and i and j denotes the ith row and jth column of the array.

Example 4: int A[2][3] = {1, 2, 3, 4, 5, 6} Assuming A is stored in a column major order with first element of A is at address 1000 and each integer occupying 2 bytes. What would be the address of the element A[1][2]?

Let A be 2-D array of size 2 × 3. The column major representation of array A is shown as follows:

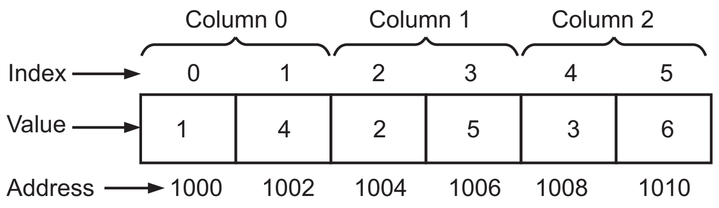


Fig. 2.10

In the case of A array the baseAddress = 1000, no_of_rows = 2 and size_of_data_type = 2 i.e. each element is taking 2 bytes in memory.

So, we can compute the memory address location of the element A[1][2] as follows. Address of element 6 which is present at cell 1, 2.

A[1][2] =baseAddress + [(j × no_of_rows + i) × size_of_data_type]

=1000 + [(2 × 2 + 1) × 2]

=1000 + [(4 + 1) × 2]

= 1000 + 10

=1010

3. Multi-dimensional Array:

- An array of arrays is called as multi-dimensional array. In simple words, an array created with more than one dimension (size) is called as multi-dimensional array.
- Multi-dimensional array can be of two dimensional array or three dimensional array or four dimensional array or more.
- Most popular and commonly used multi-dimensional array is two dimensional (2D) array. The 2D arrays are used to store data in the form of table. We also use 2D arrays to create mathematical matrices.
- The **syntax** of declaring a multi-dimensional array in C is:

```
data_type array_name[a][b][c]…[n].
```

Two dimensional array: `int two_d[10][20];`

Three dimensional array: `int three_d[10][20][30];`

- The multi-dimensional array can have more than two dimensions. An n-dimensional array A is a collection of $m_1 \times m_2 \times m_3 \dots \times m_n$ elements. All the elements in the array are referred by subscripts, $k_1, k_2, \dots k_n$ where,

$$\begin{aligned} 1 &\leq k_1 \leq m_1 \\ 1 &\leq k_2 \leq m_2 \\ &\;\;\vdots \\ 1 &\leq k_n \leq m_n \end{aligned}$$

The array A is denoted by, $A[k_1, k_2, \dots, k_n]$

Consider 3D array: int A[2] [3] [4]

The array has 2 × 3 × 4 = 24 elements in it.

The representation in memory with row-major and column major is shown in Fig. 2.11.

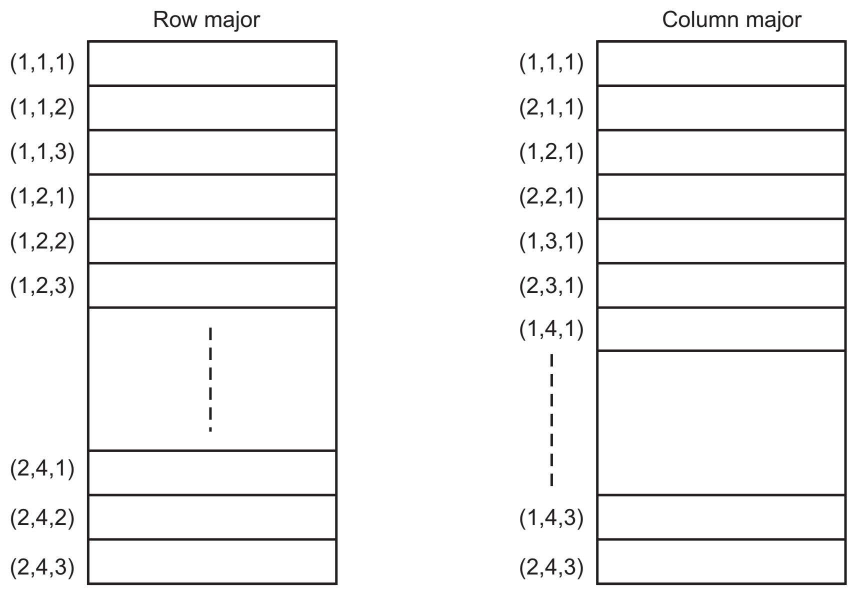


Fig. 2.11: Representation of Multi-Dimensional Array

Example 5: Consider the declaration of array A[5 …10, –5 …10] of integers. Assuming A is stored in a row major order with first element of A is at address 1000 and each integer occupying 4 bytes. What would be the lowest byte address of the element A[5, 0]?

Ans.:

A[5 … 10, − 5 … 10] is equivalent to A[6, 16] i.e. A has 6 rows and 16 columns.

To find A[5] [0] by row-major order.

$$
\begin{aligned}
a[i]\,[j] &= \text{Base address} + (i \times \text{no\_of\_col} + j) \times \text{size\_bytes} \\
&= 1000 + (5 \times 16 + 0) \times 4 \\
&= 1000 + 80 \times 4 \\
&= 1000 + 320 \\
&= 1320
\end{aligned}
$$

2.1 ARRAY FOR ADT

- The array is an Abstract Data Type (ADT) that holds a collection of elements accessible by an index.
- The array is a basic abstract data type that holds an ordered collection of items accessible by an integer index. These items can be anything from primitive types such as integers to more complex types like instances of classes.
- Arrays are defined as ADT's because they are capable of holding contiguous elements in the same order. And they permit access for the specific element via index or position.
- An element is stored in a given index and they can be retrieved at a later time by specifying the same index.

- The Array (ADT) have one property, they store and retrieve elements using an index. The array (ADT) is usually implemented by an Array (Data Structure).
- Showing that an array is an ADT an array satisfied the operations like CREATE, INSERT, DELETE, MERGE, TRAVERSAL, SEARCH, SORT etc. Thus, by using 1D (one-dimensional) array we can perform these operations so an array acts as an ADT.

2.1.1 Operations on Arrays

- Fig. 2.12 shows various operations on arrays data structure and explained as below:
  1. **Insertion** : Adding a new element to the list.
  2. **Deletion** : Removing on element from the list.
  3. **Searching** : Finding the location of the element with a given value or the record with a given key.
  4. **Sorting** : Arranging the elements in some type or order.
  6. **Traversal** : Processing each element in the list.

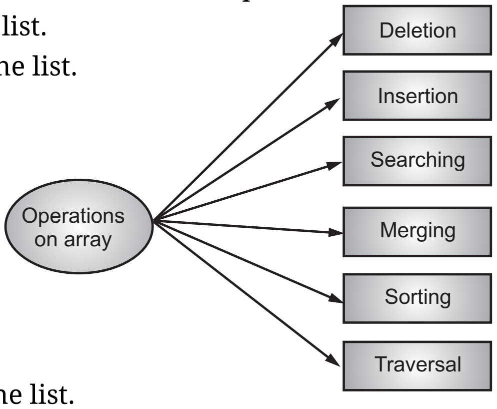


Fig. 2.12

- Here we will concentrate only on insertion and deletion operation of an array. Searching and sorting operation are explained in next sections.

Insertion of an Element in an Array:

- In order to insert an element in an array, first read the element to be inserted, read the index/location where to insert the element then make a space for new element.
- To make a space for new element, shift all elements from (size-1) to (index) to one index/location ahead and increase the size of array by one and insert element at specified indexed position.

Insert Algorithm:

- Let A [Max] is an array and size specifies length of an array.

```
1.  Accept the item to be inserted into element.
2.  Accept the location into loc where element to be inserted.
3.  Repeat for i=size-1 down to loc.
4.      Set a[i+1]=a[i].
5.  Set size=size+1.
6.  Set a[loc]=element.
```

Program 2.1: Program for insertion of an element in array.

```
#include <stdio.h>
#include <process.h>
#define MAX 100    //Maximum number of elements that can be stored
```

```
int main()
   {
      int size=0,a[MAX];
      setbuf(stdout, NULL);       //Turn off buffering of stdout
      int i,element,index;
      printf("Enter the value of n:\n");
      scanf("%d", &size);
      //Code for creation of Array
      printf("Enter n elements:\n");
      for (i = 0; i < size; i++)
         scanf("%d", &a[i]);
      //Code for displaying elements of Array
      printf("\n**** Elements in Array before Insertion*****.\n");
      for(i=0;i<size;i++)
         printf("%d  ",a[i]);
      //Code for insertion in Array
      printf("\nEnter the element to be inserted :\n");
      scanf("%d", &element);
      printf("Enter the index where element to be inserted:\n");
      scanf("%d", &index);
      if(index<0 || index>size-1)
      {
         printf("Invalid index \n");
         return 0;
      }
      else
      {
         //Make space at the specified location
         for (i = size-1; i >= index; i--)
         {
               a[i+1] = a[i];
         }
         size++;  // No of elements increased by 1
         a[index] = element;
      }
      printf("**** Elements in Array after Insertion*****.\n");
      for(i=0;i<size;i++)
         printf("%d  ",a[i]);
      return 0;
   }
```

Output:

```
Enter the value of n:
3
Enter n elements:
1
2
3
**** Elements in Array before Insertion*****.
1  2  3
Enter the element to be inserted :
4
Enter the index where element to be inserted:
0
**** Elements in Array after Insertion*****.
4  1  2  3
```

Deletion of an Element in an Array:

- In order to delete an element from an array, first read the index/location where to delete the element.
- To delete specified indexed element, move all elements from (index+1) to (size-1) one index/location down and decrease the number of elements.

Delete Algorithm:

- Let A [Max] is an array and size specifies length of array.

```
1. Accept the location into loc where element to be deleted.
2. Repeat for i=loc+1 to size-1.
3.     Set a[i-1]=a[i].
4. Set size=size-1.
```

Program 2.2: Program for deletion of an element in array.

```
#include <stdio.h>
#include <process.h>
#define MAX 100    //Maximum number of elements that can be stored
int main()
{
   int size=5,i,index;
   int a[5]={10,20,30,40,50};
   setbuf(stdout, NULL);      //Turn off buffering of stdout
   printf("**** Elements in Array before deletion*****.\n");
   for(i=0;i<size;i++)
      printf("%d  ",a[i]);
   printf("\nEnter index of the element to be deleted :\n");
   scanf("%d", &index);
```

```
      if(index<0 || index>size-1)
      {
         printf("invalid position");
         return 0;
      }
      for (i = index+1; i <= size-1; i++)
      {
         a[i-1] = a[i];
      }
      size--;  // No of elements reduced by 1
      printf("**** Elements in Array after deletion*****\n");
      for(i=0;i<size;i++)
      printf("%d  ",a[i]);
   }
```

Output:

```
   **** Elements in Array before deletion*****.
   10  20  30  40  50
   Enter index of the element to be deleted :
   4
   **** Elements in Array after deletion*****
   10  20  30  40
```

Program 2.3: Menu driven program for array to create, insert, delete and display elements in array.

```
   #include <stdio.h>
   #include <process.h>
   #define MAX 100    //Maximum number of elements that can be stored
   int num=0,a[MAX];
   void create();
   void insert();
   void delete();
   void display();
   int main()
   {
       int ch;
       setbuf(stdout, NULL);   //Turn off buffering of stdout
       printf("*** Array Menu ***");
       printf("\n1.Create\n2.Insert\n3.Delete\n4.Display\n5.Exit");
```

```
    while(1)    //infinite loop
    {
        printf("\nEnter your choice(1.Create 2.Insert 3.Delete
                                                4.Display 5.Exit):\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: create();
                    break;
            case 2: insert();
                    break;
            case 3: delete();
                    break;
            case 4: display();
                    break;
            case 5: exit(0);
            default: printf("\nWrong Choice!!");
        }
    }
    return 0;
}
void create()
{
   int i;
   printf("Enter the value of n:\n");
   scanf("%d", &num);
   printf("Enter n elements:\n");
   for (i = 0; i < num; i++)
         scanf("%d", &a[i]);
}
void insert()
{
   int i,element,index;
   printf("Enter the element to be inserted :\n");
   scanf("%d", &element);
   printf("Enter the index after which element to be inserted:\n");
   scanf("%d", &index);
   //Make space at the specified location
      for (i = num-1; i >= index; i--)
      {
         a[i+1] = a[i];
        }
        num++;  // No of elements increased by 1
        a[index] = element;
}
void delete()
{
    int i,index;
    printf("Enter index of the element to be deleted :\n");
    scanf("%d", &index);
    for (i = index+1; i <= num-1; i++)
    {
        a[i-1] = a[i];
    }
    num--;  // No of elements reduced by 1
}
void display()
{
     int i;
     printf("**** Elements in Array*****.\n");
     for(i=0;i<num;i++)
           printf("%d  ",a[i]);
}
```

Output:

```
*** Array Menu ***
1.Create
2.Insert
3.Delete
4.Display
5.Exit
Enter your choice(1.Create 2.Insert 3.Delete 4.Display 5.Exit):
1
Enter the value of n:
5
Enter n elements:
1
2
3
4
5
Enter your choice(1.Create 2.Insert 3.Delete 4.Display 5.Exit):
```

```
2
Enter the element to be inserted :
6
Enter the index after which element to be inserted:
4
Enter your choice(1.Create 2.Insert 3.Delete 4.Display 5.Exit):
4
**** Elements in Array*****.
1  2  3  4  5  6
Enter your choice(1.Create 2.Insert 3.Delete 4.Display 5.Exit):
3
Enter index of the element to be deleted :
0
Enter your choice(1.Create 2.Insert 3.Delete 4.Display 5.Exit):
4
**** Elements in Array*****.
2  3  4  5  6
Enter your choice(1.Create 2.Insert 3.Delete 4.Display 5.Exit):
```

2.2 ARRAY APPLICATIONS: SEARCHING

- Searching is a technique of finding an element from the given data list or set of the elements like an array, list, or trees. It is a technique to find out an element in a sorted or unsorted list.
- For example, consider an array of 10 elements. These data elements are stored in successive memory locations. We need to search an element from the array.
- In the searching operation, assume a particular element n is to be searched. The element n is compared with all the elements in a list starting from the first element of an array till the last element.
- In other words, the process of searching is continued till the element is found or list is completely exhausted.
- When the exact match is found then the search process is terminated. In case, no such element exists in the array, the process of searching should be abandoned.
- The complexity of any searching method is determined from the number of comparisons performed among the collected elements in order to find the element.
- The time required for the operation is dependent on the complexity of the operation or algorithm.
- In other words, the performance of searching algorithm can be measured by counting the comparisons in order to find the given element from the list.
- There are following three cases in which an element can be found:
  1. In the **best case**, the element is found during the first comparison itself.
  2. In the **worst case**, the element is found only in the last comparison.
  3. Whereas in the **average case**, number of comparisons should be more than comparisons in the best case and less than the worst case.

- Searching algorithms are used to find the element in the list. There are two searching algorithms, Linear search and Binary search.
- Linear search is easiest and straight most searching technique as search element is compared to all the elements in the array in sequence.
- The searched element is compared with first element which is positioned at index 0 then it is compared with second element which is positioned at index 1 and so on.
- If search element is matched with one of the element in array, then one can print that desired element found else desired element not exists in array.
- Binary search uses divide and conquer approach to finds the position of a specified value within a sorted array.
- The prerequisite to search an element in array using binary search is, the list should be sorted in ascending order.

2.2.1 Linear Search or Sequential Search [April 15]

- Linear search, also called as orderly search or sequential search, because every key element is searched from first element in an array i.e., a[0] to last element in an array i.e a[n-1].
- Linear search algorithm works by comparing every element in an array with the key element. If a key element match with any element in the array, it stops search and return the location of key element in the array.
- It is usually very simple to implement and is practical when the list has only a few elements or when performing a single search in an unordered list.

Algorithm for Linear Search:

- Linear Search where the result of the search is supposed to be either the location of the list item where the desired value is found; or an invalid location to indicate that the desired element does not occur in the list.
- Consider array 'A' of 'n' elements need to be searched for the element 'find'.

```
    Linear_Search(A[], n, key)
Step 1 : i = 0, Take the first element in the list i.e. a[i]
Step 2 :     //If the element in the list is equal to the desired element then return i
                 if(A[i]==key) then
                 return i
Step 3 : if i<n then
             i=i+1
                 Go to Step 2
Step 4 : if i = n then return -1
Step 5 : Stop
```

Program 2.4: C program for linear search.

```
#include <stdio.h>
int linear_search(int [], int, int);
int main()
{
   int a[10], key, c, n, position;
   printf("Input number of elements in array\n");
   scanf("%d", &n);
   printf("Enter %d numbers\n", n);
   for (int i = 0; i < n; i++)
      scanf("%d", &a[i]);
   printf("Enter number to search\n");
   scanf("%d",&key);
   position = linear_search(a, n, key);
   if (position == -1)
      printf("%d is not present in array.\n", key);
   else
      printf("%d is present at location %d.\n", key, position+1);
   return 0;
}
int linear_search(int a[], int n, int key)
{
   int i;
   for (i = 0 ;i < n ; i++ )
   {
      if (a[i] == key)
         return i;
   }
   return -1;
}
```

Output:

```
Input number of elements in array
5
Enter 5 numbers
5  6  3  4  8
Enter number to search
1
1 is not present in array.
```

Complexity:

- For a list with n items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed.
- The worst case is when the value is not in the list (or occurs only at the end of the list), in which case n comparisons are needed. So complexity is O(n).

Advantages:

1. The linear search is simple: It is very easy to understand and implement.
2. It does not require the data in the array to be stored in any particular order.

Disadvantage:

1. If array size i.e., n is very large then sequential search is not efficient.
2. Inversely, when a key element matches the last element in the array or a key element does not match any element then linear search algorithm is a worst case.

2.2.2 Sequential Searching Variations

- Three useful variations on the sequential search algorithm are the sentinel search, the probability search and the ordered list search.

2.2.2.1 Sentinel Search

- Sentinel search is a similar to the linear search where the number of comparisons is reduced as compared to a traditional linear search.
- When a linear search is performed on an array of size N then in the worst case a total of N comparisons are made when the element to be searched is compared to all the elements of the array with (N + 1) comparisons.
- In sentinel search, we replace the last element of the list with the search element itself and run a while loop to see if there exists any copy of the search element in the list and quit the loop as soon as we find the search element.
- The while loop makes only one comparison in each iteration and it is sure that it will terminate since the last element of the list is the search element itself.
- So in the worst case (if the search element does not exist in the list) then there will be at most N+2 comparisons (N comparisons in the while loop and 2 comparisons in the if condition). Which is better than (2N+1) comparisons as found in Simple Linear Search. Time complexity of sentinel search is O(n).
- The following program illustrate that how the sentinel search optimizes the loop and determine after the loop completes whether we found actual data or the sentinel.

Program 2.5: Program for sentinel search.

```
#include<stdio.h>
int main()
{
    int a[10], key, c, n, position;
    printf("input number of elements in array\n");
    scanf("%d", &n);
    printf("enter %d numbers\n", n);
```

```
      for (int i = 0; i < n; i++)
         scanf("%d", &a[i]);
      printf("enter number to search\n");
      scanf("%d",&key);
      position = linear_search(a, n, key);
      printf("result is %d at index %d\n", key, position );
      /*printf("%d", position);
      if (position == -1)
         printf("%d is present at location %d.\n", key, position);
      else
         printf("%d is not present in array.\n", key);*/
      return 0;
   }
   int linear_search(int a[], int n, int key)
   {
       int i = 0;
       a[n] = key;
       while (a[i] != key) {
           i++;
       }
       return i;
   }
```

Output:

```
   input number of elements in array
   5
   enter 5 numbers
   3 5 7 8 6
   enter number to search
   8
   result is 8 at index 3
```

Algorithm: In the following algorithm, the loop tests two conditions: the end of the list and the target not being found.

- When the inner loop of a program tests two or more conditions, we should try to reduce the testing to just one condition.
- If we know that the target will be found in the list, we can eliminate the test for the end of the list.
- The Algorithm seqSearch (list, last, item, locn) Locate the item in an unordered list of elements.

  Pre and Post define in the algorithm.
- A target is put in the list by adding an extra element (sentinel entry) at the end of the array and placing the target in the sentinel. We can then optimize the loop and determine after the loop completes whether we found actual data or the sentinel.
- The obvious disadvantage is that the rest of the processing must be careful to never look at the sentinel element at the end of the list. The pseudocode for the sentinel search is shown in Algorithm.

Algorithm: Sentinel Search (list, last, target, locn)

Locate the target in an unordered list of elements.

```
Pre     list must contain element at the end for sentinel 1 last is index to last data
        element in the list target contains the data to be located locn is address of index
        in calling algorithm
Post    if found--matching index stored in locn and found set true
        if not found--last stored in locn and found false
        Return found true or false
1. set list[last + 1] to target
2. set i to 0
3. loop (target not equal list[i])
    (i)  increment i
4.  end loop
5.  if (i <= last)
    (i)  set found to true
    (ii) set locn to i
6.  else
    (i)  set found to false
    (ii) set locn to last
7.  end if
8.  return found
end Sentinel Search
```

2.2.2.2 Probability Search

- In the probability search, the data in the array are arranged with the most probable search elements at the beginning of the array and the least probable at the end.
- It is especially useful when relatively few elements are the targets for most of the searches.
- To ensure that the probability ordering is correct over time, in each search we exchange the located element with the element immediately before it in the array.
- When the target is less than or equal to the last element, the last element becomes a sentinel, allowing us to eliminate the test for the end of the list.
- A typical implementation of the probability search is shown in Algorithm.

Algorithm: Probability Search (list, last, target, locn)

Locate the target in a list ordered by the probability of each element being the target-most probable first, least probable last.

```
Pre   list must contain at least one element
      last is index to last element in the list
      target contains the data to be located
      locn is address of index in calling algorithm
```

```
Post  if found--matching index stored in locn,
      found true, and element moved up in priority
      if not found -last stored in loon & found false
Return found true or false
1. find target in list
2. if (target in list)
   (i)   set Round to true
   (ii)  set locn to index of element containing target
   (iii) if (target after first element)
         (a) move element containing target up one location
   (iv)  end if
3. else
   (i)   set found to false
4. end if
5. return found
end ProbabilitySearch
```

2.2.2.3 Ordered List Search

- Although we generally recommend a binary search when searching a list ordered on the key (target), if the list is small it may be more efficient to use a sequential search.
- When searching an ordered list sequentially, however, it is not necessary to search to the end of the list to determine that the target is not in the list.
- We can stop when the target becomes less than or equal to the current element we are testing.
- In addition, we can incorporate the sentinel concept by bypassing the search loop when the target is greater than the last item.
- In other words, when the target is less than or equal to the last element, the last element becomes a sentinel, allowing us to eliminate the test for the end of the list.
- Although it can be used with array implementations, the ordered list search is more commonly used when searching linked list implementations. The pseudo code for searching an ordered array is found in Algorithm.

```
Algorithm: Ordered List Search (list, last, target, locn)
Locate target in a list ordered on target.
Pre   list must contain at least one element last is index to last element in the list
      target contains the data to be located locn is address of index in calling
      algorithm
Post  if found-matching index stored in locn-found true
      if not found--locn is index of first element > target or locn equal last and found
      is false
Return found two or false
1. if (target less than last element in list)
   (i)  find the element less than or equal to target
   (ii) set locn to index of element
```

```
2. else
   (i) set lock to last
3. end if
4. if (target in list)
   (i) set found to true
5. else
   (i) set found to false
6. end if
7. return found
end OrderedListSearch
```

2.2.3 Binary Search [April 15, 16, 17, 18]

- Binary search is quicker than the linear search. However, it cannot be applied on unsorted data structure. The binary search is based on the approach divide-and-conquer.
- The binary search starts by testing the data in the middle element of the array. This determines target is whether in the first half or second half.
- If target is in first half, we do not need to check the second half and if it is in second half no need to check in first half.
- Similarly, we repeat this process until we find target in the list or not found from the list. Here we need three variables to identify first, last and middle elements.
- Binary Search algorithm can be iterative or recursive.

1. **Recursive Algorithm for Binary Search:**

   Step 1 : The key is compared with item in the middle position of an array.

   Step 2 : If the key matches with item, return it and stop.

   Step 3 : If the key is less than mid positioned item, then the item to be found must be in first half of array, otherwise it must be in second half of array.

   Step 4 : Repeat the procedure for lower (or upper half) of array until the element is found.

Recursive C Function for Binary Search:

```
int BinarySearch(int key, int A[], int L, int H)
{
    if(L > H)               //key does not exit
        return – 1;
    if(key == A[mid])       //base case
        return mid;
    else if (key < A[mid])  //recursive case I
        BinarySearch(key, A, L, mid - 1);
    else                    //recursive case II
        BinarySearch(key, A, mid + 1, H);
}
```

Example 6:

Consider array a, n=7, key=11

	0	1	2	3	4	5	6
a	5	7	11	19	30	35	42

Step 1 : Initialization lb = 0, ub = 6

Step 2 : mid $= \left\lfloor \frac{(lb + ub)}{2} \right\rfloor = \left\lfloor \frac{(0 + 6)}{2} \right\rfloor = 3$

Step 3 : As 11 < a[3], first half need to be checked
ub = mid - 1 = 3 - 1 = 2

Step 4 : mid $= \left\lfloor \frac{(0 + 2)}{2} \right\rfloor = \left\lfloor \frac{2}{2} \right\rfloor = 1$

Step 5 : As 11 > a[1], second half need to be checked.
*l*b = mid + 1 = 1 + 1 = 2

Step 6 : mid $= \left\lfloor \frac{(2 + 2)}{2} \right\rfloor = \left\lfloor \frac{4}{2} \right\rfloor = 2$

Now 11 == a[2] return 2
i.e. 11 is present at position 2 in an array a.

Example 7: n = 9, key = 25.

A	2	6	13	21	36	47	63	81	97
	0	1	2	3	4	5	6	7	8

$$\begin{tikzcd}[row sep=small]
-1 & \\
BS(A,0,8,25) \arrow[u] \arrow[d, shift right] & mid = 4 \\
BS(A,0,3,25) \arrow[u, shift right, "-1"'] \arrow[d, shift right] & mid = 1 \\
BS(A,2,3,25) \arrow[u, shift right, "-1"'] \arrow[d, shift right] & mid = 2 \\
BS(A,3,3,25) \arrow[u, shift right, "-1"'] \arrow[d, shift right] & mid = 3 \\
BS(A,4,3,25) \arrow[u, shift right, "-1"'] &
\end{tikzcd}$$

As -1 value is returned by call BS(A,0,8,25) that means key 25 is not present in given sorted array A.

2. Iterative / Non-Recursive Binary Search Algorithm:

```
Step 1 : Initialize low = 0, high = n − 1
Step 2 :     While low <= high
Step 3 :     mid = $\left\lfloor \left(\frac{low + high}{2}\right) \right\rfloor$
Step 4 :     If a[mid] == Item
Step 5 :             Set Pos = mid
Step 6 :             Break and Jump to step 10
Step 7 :         Else if Item< a[mid]
Step 8  :     high = mid −1
Step 9  :         Else    low = mid +l
Step 10 :  If Pos < 0
Step 11 :         Print "Element is not found"
Step 12 :  Else   Print Pos
```

Iterative C function for Binary Search(Non-Recursive):

```
int bsearch(int a[ ],int n, int key)
{
     int lb = 0,ub,mid;
     lb = 0;
     ub = n-1;
     while(lb <= ub )
   {
         mid=(lb+ub)/2;
         if(key < a[mid])
               ub = mid-1;
        else if(key > a[mid])
               lb=mid+1;
        else if(key == a[mid])
               return(mid);
   }
return -1;
}
```

Example 8:

Index	0	1	2	3	4	5
Data	65	80	84	91	95	99

(a) Search for the key of 91 in the above sequence:

Iteration	Low	High	Mid	Found?	Action
1	0	5	2	No	Move low to mid + 1
2	3	5	4	No	Move high to mid – 1
3	3	3	3	Yes	Done! Index is mid

The key was found at the third index in the given sequence.

(b) Search for the key of 82.

Iteration	Low	High	Mid	Found?	Action
1	1	6	3	No	Move high to mid −1
2	1	2	1	No	Move low to mid + 1
3	2	2	2	No	Move low to mid + 1
4	3	2	This ends it!		

The value of low cannot be greater than high; this means that the key is not in the vector. So, the algorithm repeats until either the key is found or until low > high, which means that the key is not there.

Program 2.6: Program for binary search.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int first, last, middle, n, i, find, a[100];
   setbuf(stdout, NULL);
   printf("Enter the size of array: \n");
   scanf("%d",&n);
   printf("Enter n elements in Ascending order: \n");
   for (i = 0; i < n; i++)
      scanf("%d",&a[i]);
   printf("Enter value to be search:\n");
   scanf("%d", &find);
   first = 0;
   last = n - 1;
   middle = (first+last)/2;
   while (first <= last)
   {
      if (a[middle] < find)
         first = middle + 1;
      else if (a[middle] == find)
      {
         printf("Element found at index %d.\n",middle);
         break;
      }
      else
         last = middle - 1;
      middle = (first + last)/2;
   }
   if (first > last)
      printf("Element Not found in the list.");
   return 0;
}
```

Output:

```
Enter the size of array:
5
Enter n elements in Ascending order:
1
4
7
9
11
Enter value to be search:
7
Element found at index 2.
```

Complexity:

- The maximum number of key comparisons are $\log_2 n$. So complexity is O(log n).

Advantage:

1. It is suitable for storage structure that support direct access.
2. It is efficient for large list.
3. It is very time efficient as time complexity is O(log n).

Disadvantage:

1. Array should be sorted before applying this algorithm on it.
2. It is not suitable for unsorted data.

2.2.4 Comparison of Searching Methods

- Following table gives the difference between Linear Search (Sequential Search) and Binary Search:

Sr. No.	Parameters	Linear Search	Binary Search
1.	Approach	Sequential approach.	Divide and conquer approach.
2.	Works on	Sorted and unsorted list of elements.	Works only on sorted list of elements.
3.	Working	It works by comparing the value to be searched with every element of the array/list one by one in a sequence until a match is found.	In binary search it divide the table into two parts, a lower value part and an upper-value part, after dividing It will check with the middle value if its lesser than the searched element than it goes to right half else it goes to left half.
4.	Time complexity	Search time is O(n).	Search time is O(log n).
5.	Speed	It slower than binary search.	Binary search is quicker/faster than linear search.
6.	Recommended	Searching in small list.	Searching in large list.

2.3 SORTING

- Sorting refers to the process of the process of arranging a list of elements in a particular order. The elements are arranged in increasing or decreasing order of their key values.
- Sorting algorithm specifies the way to arrange data in a particular order. Some sorting algorithms in C language are insertion sort, selection sort, merge sort, bubble sort and so on.
- The different sorts are shown in Fig. 2.13.

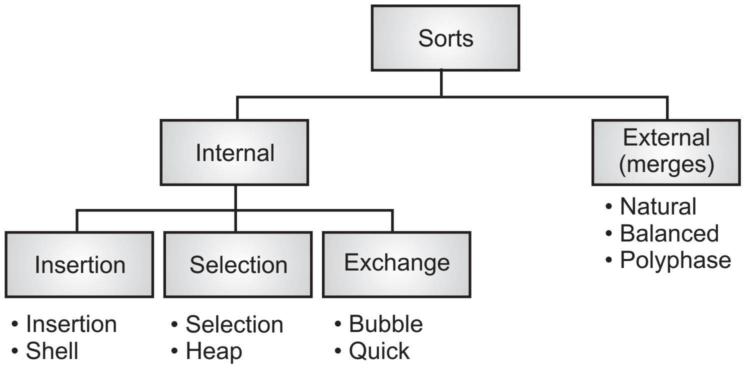


Fig. 2.13: Sort Classification

2.3.1 Sorting Terminology

- Sorting is a technique to rearrange the elements in ascending or descending order, which can be numerical, lexicographical, or any user-defined order.
- Data can be arranged either in ascending order or in descending order which is called sort order.

For example, if data is,	30	36	50	49	22
Ascending order:	22	30	36	49	50
Descending order:	50	49	36	30	22

2.3.1.1 Internal Sorting

- Internal sorting method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process.
- If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting.
- There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints.
- An internal sort is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory.
- Examples: Bubble sort, Insertion sort, Quick sort.

2.3.1.2 External Sorting

- External sorting large amount of data requires external or secondary memory. This process uses external memory such as Hard Disk, floppy, tape etc. to store the data which does not fit into the main memory.
- So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.
- External sorting algorithms that can handle massive amounts of data. Example: Merge sort.

2.3.1.3 Stable Sorting [April 17]

- A sorting algorithm is said to be stable if it preserves the order for all records with duplicate keys; means if for all records, if keys of records are equal then the equal key records are appeared in sorted table as same as the order in which they appeared unsorted table.
- The stability of a sorting algorithm is concerned with how the algorithm treats equal (or repeated) elements.
- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.
- Examples of stable sorting algorithms are bubble sort, merge sort and insertion sort. Some examples of unstable sorting algorithms are heap sort and quick sort.
- **Example**:

  Original unsorted table of student name and marks:

Name	Akash	Abhay	Omkar	Sharvari	Puja	Ruta
Marks	75	90	80	85	85	82

  Stable sorted table is: (Mark is key)

Name	Abhay	Sharvari	Puja	Ruta	Omkar	Akash
Marks	90	85	85	82	80	75

  Unstable sorted table is:

Name	Abhay	Puja	Sharvari	Ruta	Omkar	Akash
Marks	90	85	85	82	80	75

- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.

2.3.1.4 In-place Sorting

- Sorting algorithms may require some extra space for comparison and temporary storage of few data elements.

- These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting.
- Bubble sort is an example of in-place sorting.
- However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting.
- Merge-sort is an example of not-in-place sorting.

2.3.2 Comparison Based Sorting

- There are two types of sorting as shown in the Fig. 2.14.

$$\begin{tikzcd} & \textbf{Sorting} \arrow[dl] \arrow[dr] & \\ \text{Comparison Based Sorting} & & \text{Non Comparison Based Sorting} \end{tikzcd}$$

Examples: Bubble sort, Insertion sort, Selection sort, Quick sort, Merge sort

Examples: Counting Sort, Radix Sort

Fig. 2.14

- In a comparison based sorting algorithms, we compare elements of an array with each other to determines which of two elements should occur first in the final sorted list.
- In comparison based sorting, elements of an array are compared with each other to find the sorted array.
- Comparison Based Soring techniques are bubble sort, selection sort, insertion sort, Merge sort, quicksort, heap sort etc.
- These techniques are considered as comparison based sort because in these techniques the values are compared, and placed into sorted position in different phases.
- In non-comparison based sorting, the precondition is used in the given range.
- The non-comparison based sorting algorithms perform sorting without comparing the elements rather by making a certain assumption about the data they are going to sort.

2.3.2.1 Lower Bound on Comparison Based Sorting

- Further we will discuss the sorting methods and its time complexity. The comparison based Soring techniques such as bubble sort, selection sort, insertion sort, Merge sort, quicksort, heap sort etc. will discuss.
- For lower bound on comparison based sorting, the time complexity in Big-O in the worst case of all these method is considered.
- In the worst case bubble sort, selection sort, insertion sort, quicksort have time complexity O(n^2), and for merge sort time complexity is O(n log n).
- Therefore O(n log n) is the lower bound on comparison based sorting because no sorting algorithm will be faster than O(n log n).
- Note that according to a strict mathematical interpretation of the Big-O notation, Merge sort surpasses other sort in efficiency in worst case.

- Big-O is based on an analytical evaluation of the algorithms, not an evaluation of the code. Depending on the algorithm's implementation, the actual run time can be affected.

2.4 SORTING METHODS

- In data structure sorting is important because of retrieval of information/data became easier when it sorted in some predefined order.
- Sorting can be performed in many ways. Over time, several methods have been devised to sort information, following specific algorithms. Some examples of these algorithms are bubble sort, selection sort, quick sort, merge sort, and so on.

Objectives of Sorting Algorithms:

1. Minimize exchanges of data and reduce processing time.
2. In case of large data, move data from secondary storage (external sorting).
3. If possible, retain all the data in main memory; so that random access into an array can be effectively used, (internal sorting).
4. In most of the sorting algorithms, the time complexity ranges from O(n log n) to $O(n^2)$.
5. In efficiency of sorting method is measured by the run time used for execution of algorithm. One sorting method may use different runtimes on different machines.
6. Normally, those programs that requires less time, require more space and vice versa. But some algorithm use both minimum time and minimum space having complexity O(n log n).

2.4.1 Bubble Sort [Oct. 16, 17]

- In this sorting method, the list is divided into two sub-lists sorted and unsorted. The smallest element is bubbled from unsorted sub-list. After moving the smallest elements, the imaginary wall moves one element ahead.
- The bubble sort was originally written to bubble up the highest element in the list. But there is no difference whether highest / lowest element is bubbled.
- This method is easy to understand but time consuming. In this type, two successive elements are compared and swapping is done.
- Thus, step-by-step entire array elements are checked. Given a list of 'n' elements the bubble sort requires up to n-1 passes to sort the data.
- To arrange the elements in the ascending order, we compare each item in the list with the item next to it and swapping them if required. The algorithm repeats these comparison process until n - 1 passes where n is the number of input items.
- For correct order, the larger values 'Bubble' to the end of the list, while smaller values 'sink' towards the beginning of the list.
- Suppose, the list of number A[0], A[1], A[2], - - A[n-1] is in memory, which is to be sorted.

  Algorithm: (Arranging elements in ascending order).

  Step 1 : Read N, number of elements in the list.

  Step 2 : Read array A[0], A[1] - - - A[n-1]

```
Step 3  :  for i = 0 to N - 1 do
              for j = 0 to N – i - 1 do
                  if (A[j] > A[j + 1])  then
                      temp = A[j]
                      A[j] = A[j + 1]
                      A[j + 1] = temp
Step 4  :  Stop
```

Example 9: Consider the array A containing following elements to be sorted using bubble sort:

13, 11, 14, 15, 19, 9

Solution: N = 6 and A = {13, 11, 14, 15, 19, 9}

Pass 1: i = 0, j=0 to 5

```
13    11    14    15    19    9
Compare and swap if required            ← exchange as (13 > 11)
  11  13    14    15    19    9
11    13    14    15    19    9
11    13    14    15    19    9
11    13    14    15    19    9           ← exchange as (19 > 9)
11    13    14    15    9    [19]
```

One largest element is bubbled at end of the list.

Pass 2: i=1, j=0 to 4

```
11    13    14    15    9     19
11    13    14    15    9     19
11    13    14    15    9     19
11    13    14    15    9     19      ←—— exchange as (15 > 9)
11    13    14    9    [15]  [19]
                         ↑
                         └——— the second largest element
```

Pass 3: i=2, j=0 to 3

11 13 14 9 15 19

11 13 14 9 15 19

11 13 14 9 15 19 ← exchange as (14>9)

11 13 9 $\boxed{14}$ $\boxed{15}$ $\boxed{19}$

3rd largest element

Pass 4: i=3, j=0 to 2

11 13 9 14 15 19

11 13 9 14 15 19 ← exchange as (13>9)

11 9 $\boxed{13}$ $\boxed{14}$ $\boxed{15}$ $\boxed{19}$

Pass 5: i=4, j=0 to 2

11 9 13 14 15 19 ← exchange as (11>9)

9 11 13 14 15 19 ← **Sorted array**

Complexity of Bubble Sort:

- The number of passes required may be between 1 and n − 1. There are (n − 1) comparisons in the first iteration, (n − 2) comparisons in the second iteration and 1 comparison in the last iteration. The total number of comparisons,

$$= (n-1) + (n-2) + (n-3) + \ldots + 1$$
$$= \frac{n(n-1)}{2}$$

Thus, the total number of comparisons in $\frac{n(n-1)}{2}$ then time complexity = $O(n^2)$.

Worst case complexity = $O(n^2)$

Average case complexity = $O(n^2)$

Best case complexity = $O(n^2)$

Advantages and Disadvantages:

1. Simple and Easy to implement.
2. It is slowest method $O(n^2)$.
3. Inefficient for large array size.

Program 2.7: To sort an array using bubble sort.

```
#include<stdio.h>
#define MAX 20
void Bubble_sort (int A[MAX], int n);
void display (int A[MAX], int n);
void main()
{
    int A[MAX], n, i;
    printf("Enter the number of elements in the Array: ");
    scanf("%d", &n);
    printf("\nEnter the elements:\n\n");
    for(int i=0; i<n; i++)
     {
        printf(" Array[%d] = ",i);
        scanf("%d", & A[i]);
     }
    Bubble_sort(A, n);
    display (A, n);
}
/* Bubble sort function */
void Bubble_sort(int A[MAX], int n)
{
    int i, j, temp;
    for (i=0; i < n; i++)
    {
        for(j=0; j<n-i-1; j++)
        {
            if(A[j] > A[j + 1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp; /* swap */
            }
        }
    }
}
/* display the sorted list */
void display(int A[MAX], int n)
{
    for(int i=0; i < n; i++)
        printf("%d", A[i]);
}
```

Output:

```
Enter the number of elements in the Array: 5
Enter the elements:
 Array[0] = 5
 Array[1] = -9
 Array[2] = 0
 Array[3] = 7
 Array[4] = -2
The Sorted Array is:
-9   -2    0    5    7
```

2.4.2 Insertion Sort

[April 17, 18]

- In insertion sort the element is inserted at an appropriate place. Here the array is divided into two parts sorted and unsorted sub-array.
- In each pass, the first element of unsorted sub array is picked up and moved into the sorted sub array by inserting it in suitable position.

Algorithm:

```
Step 1 : Set i = 1;
Step 2 : key = A[i];
Step 3 : Set j = i - 1;
Step 4 : If key < A[j] then
           A[j+1] = A[j];
           j = j - 1;
         repeat step 2 until j >= 0
Step 5 : A[j+1]= key;
         i=i+1;
Step 6 : Repeat step from 2 to 5 till i <N
Step 7 : Stop
```

Example 10:

Consider array a

0	1	2	3	4
22	14	25	10	5

n = 5

Pass 1: i = 1, j = 0, k = 14

← Sorted array →	← Unsorted array →			
22	14	25	10	5
j = 0	i = 1			

As 14 < a[0] i.e. 22, so shift j^{th} element to $(j + 1)^{th}$ position

0	1	2	3	4
22	22	25	10	5

i.e. a[1] = 22 and j = j − 1 = −1

0	1	2	3	4
14	22	25	10	5

← Sorted → array | ← Unsorted → array

a[j + 1] = key
i.e. a[0] = 14

Pass 2: i = 2, j = 1, k = 25

← Sorted → array | ← Unsorted → array

14	22	25	10	5

j = 1 i = 2

As 25 < 22 → No

So a[2] = 25

Pass 3: i = 3, j = 2, k = 10

14	22	25	10	5

j = 2 i = 3

As 10 < 25, a[3] = a[2] i.e. a[3] = 25, j = j − 1 = 1

14	22	25	25	5

j = 1 i = 3

As 10 < 22, a[2] = a[1] i.e. a[2] = 22, j = j − 1 = 0

14	22	22	25	5

j = 0 i = 3

As 10 < 14, a[1] = a[0] i.e. a[1] = 14, j = j − 1 = −1

14	14	22	25	5

a[j + 1] = key i.e. a[0] = 10

10	14	22	25	5

← Sorted array → | ←Unsorted array→

Pass 4: i = 4, j = 3, k = 5

Compare k < a[j] and shift a[j] to a[j + 1]

And when j = − 1, store

a[j + 1] = key

i.e. a[0] = 5

10	14	22	25	25

10	14	22	22	25

10	14	14	22	25

10	10	14	22	25

5	10	14	22	25

← Sorted array

Complexity of Insertion Sort:

Total number of comparisons are:

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} \text{ which is } O(n^2)$$

Worst case complexity $O(n^2)$

Average case complexity $O(n^2)$

If array is already sorted then in every pass only one comparison is made, so

Best case complexity is $O(n)$

Advantages and Disadvantages:

1. Relatively simple and Easy to implement.
2. Inefficient for large array as time complexity is $O(n^2)$.
3. The insertion sort is highly efficient if the array is already in almost sorted order.

Program 2.8: To implement insertion sort.

```
#include<stdio.h>
#define MAX 20
void insertionsort(int A[MAX], int n)
{
    int i, j, key;
    for (i = 1; i < n; i++)
    {
          key = A[i];
    for(j = i-1;(j >= 0) && (key < A[j]); j--)
                    A[j+1] = A[j];
                    A[j+1] = key;
    }
}
void main( )
{
     int A[MAX], i, n;
     printf("How many elements you want to sort?\n");
     scanf("%d",&n);
     printf("\nEnter the Elements into an array:\n");
     for (i = 0;  i < n; i++)
         scanf( "%d", &A[i] );
     insertionsort (A, n); /* function call */
     printf("\nElements after sorting: \n");
     for( i= 0 ; i < n; i ++ )
         printf ("%d\t", A[i] );
}
```

Output:

```
How many elements you want to sort?
5
Enter the Elements into an array:
6      4      8      -9      0
Elements after sorting:
-9      0      4      6      8
```

2.4.3 Selection Sort

- This sorting algorithm, iterates through the array and finds the smallest number in the array and swaps it with the first element if it is smaller than the first element. Next, it goes on to the second element and so on until all elements are sorted.
- Assume the list containing n elements. The first elements are compare with the remaining (n-1) elements. The smallest element is placed at the first location.
- Again, the second element is compare with remaining (n-2) elements, If the element found is lesser than the second element then the swap operation is done. In this way, the entire array is checked for the smallest element and then swap.
- In selection sort, the smallest element is exchanged with the first element of the unsorted list of elements (the exchanged element takes the place where smallest element is initially placed).
- Then the second smallest element is exchanged with the second element of the unsorted list of elements and so on until all the elements are sorted.
- Selection sort performs in-place comparison which means that the array is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
- During Iteration 1, smallest element in the array (index 0 to last index) is selected and that selected element is replaced with first position element.
- During iteration 2 smallest element from subarray (starting from index 1 to last index) is selected and that selected element is replaced with second position element and so on.
- In other words, the idea of the selection sort is to search the smallest element in the list and exchange it with the element in the first position. Then, find the second smallest element and exchange it with the element in the second position, and so on until the entire array is sorted.
- In selection sort, iteration 1 looks for smallest element in the array and replace first position with that smallest element. After that iteration 2 look for smallest element present in the subarray, starting from index 1, till the last index and replace second position with that smallest element. This is repeated, until the array is completely sorted.

Selection Sort Algorithm:

- Let Array[Max] is integer array, N is size of array.

```
1.  Repeat for i= 0 to N-2.
2.      Set min =Array[i] and loc=i.
3.      Repeat for j= i+1 to N-1.
4.          IF MIN>Array[j] then.
5.              set MIN= [j] and loc= j;
6.      Interchange  Array [i] with Array[loc].
```

Example on Selection Sort:

- Consider the array: [5, -1, 2, 0, 3]. The first element is 5. The next part we must find the smallest number from the remaining array. The smallest number from -1,2 0 and 3 is -1. So, we replace 5 by -1.
- The new array is [5, 2, 0, 3] Again, this process is repeated. Finally, we get the sorted array as [-1,0, 2, 3, 5].

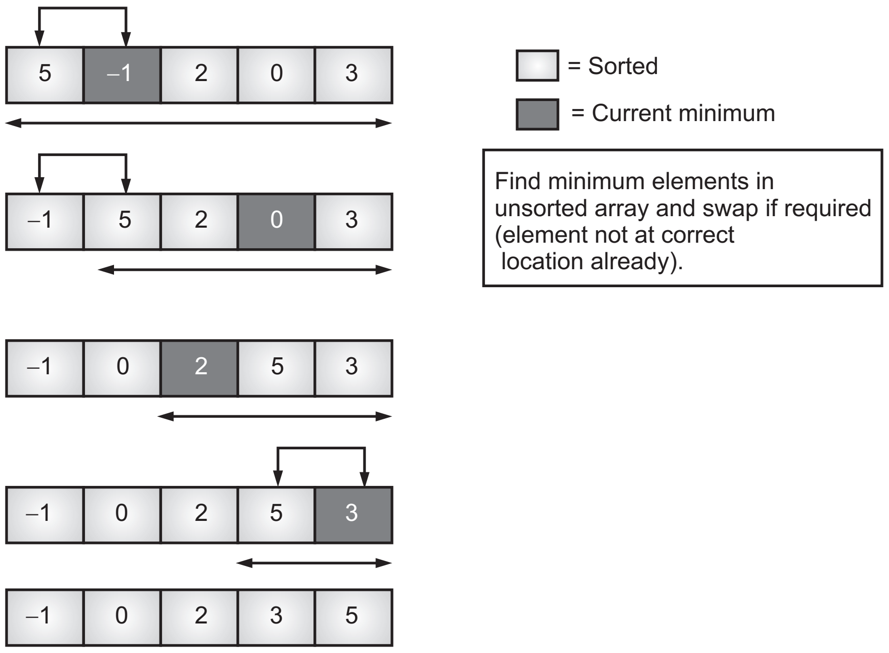

= Sorted
= Current minimum
Find minimum elements in unsorted array and swap if required (element not at correct location already).

Fig. 2.15

Program 2.9: Program for selection sort.

```
#include <stdio.h>
int main()
{
    int array[100], n, i, j, position, t;
    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for (i = 0; i < n; i++)
    scanf("%d", &array[i]);
    for (i = 0; i < (n - 1); i++) // finding minimum element (n-1) times
    {
       position = i;
       for (j = i + 1; j < n; j++)
       {
          if (array[position] > array[j])
          position = j;
       }
       if (position != i)
       {
          t = array[i];
          array[i] = array[position];
          array[position] = t;
       }
    }
    printf("Sorted list in ascending order:\n");
    for (i = 0; i < n; i++)
        printf("%d\n", array[i]);
    return 0;
 }
```

Output:

```
Enter number of elements
10
Enter the numbers
5
-1
8
7
12
3
6
9
-4
3
```

```
Sorted list in ascending order:
-4
-1
3
3
5
6
7
8
9
12
```

Time Complexity of Selection Sort:

- The mathematical expression for the iterations will be equal to (n-1) + (n-2) + ... + (n-(n-1)). Thus, the expression become n*(n-1)/2. Thus the number of comparisons is proportional to (n^2). Therefore, the time complexity of selection sort is O(n^2).
- Therefore, Time complexity of selection sort in worst case is O(N^2), in Best Case is O(N^2) and in Average case is O(N^2).

Advantages and Disadvantage of Selection Sort:

1. Selection sort is simple and easy to implement.
2. It gives 60% performance improvement over bubble sort.
3. Selection sort perform poor when dealing with large number of elements.

Program 2.10: Program for selection sort.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i,j,n,loc,temp,min,a[30];
    setbuf(stdout, NULL);
    printf("Enter the size of array:\n");
    scanf("%d",&n);
    printf("Enter n elements:\n");
    for(i=0;i<n;i++)
    {
   scanf("%d",&a[i]);
    }
    //computation of selection sort
    for(i=0;i<n-1;i++)
    {
        min=a[i];
        loc=i;
        for(j=i+1;j<n;j++)
        {
            if(min>a[j])
            {
                min=a[j];
                loc=j;
            }
        }
        temp=a[i];
        a[i]=a[loc];
        a[loc]=temp;
    }
    printf("The sorted array after Selection sort is:\n");
    for(i=0;i<n;i++)
    {
        printf("%d ",a[i]);
    }
    return 0;
}
```

Output:

```
Enter the size of array:
5
Enter n elements:
5
7
1
4
2
The sorted array after Selection sort is:
1 2 4 5 7
```

2.4.4 Merge Sort **[April 15, 16, 18]**

- The basic concept of merge sort is divides the list into two smaller sub lists of approximately equal size.
- Recursively repeat this procedure till only one element is left in the sub list. After this, various sorted sub lists are merged to form sorted parent list. This process goes on recursively till the original sorted list arrived.

Algorithm for Merge Sort:

- Merge sort is based on the divide-and-conquer paradigm. Its worst case running time has a lower order of growth than insertion sort.

- Since we are dealing with sub problems, we state each sub problem as sorting a sub array A[p .. r]. Initially, p = 1 and r = n, but these values change as we recourse through sub problems.

  To sort A[p .. r]:

  1. **Divide Step:** If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two sub arrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].
  2. **Conquer Step:** Conquer by recursively sorting the two sub arrays A[p .. q] and A[q + 1 .. r].
  3. **Combine Step:** Combine the elements back in A[p .. r] by merging the two sorted sub arrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).
- Note that the recursion bottoms out when the sub array has just one element, so that it is trivially sorted.
- To sort the entire sequence A[1 … n], make the initial call to the procedure MERGE-SORT (A, 1, n).

  MERGE-SORT (A, p, r)

```
1.  IF p < r                        // Check for base case
2.     THEN q = [ ( p + r ) / 2 ]   // Divide step
3.           MERGE (A, p, q)        // Conquer step.
4.           MERGE ( A, q + 1, r )  // Conquer step.
5.           MERGE ( A, p, q, r )   // Combine step.
```

Example: A list of unsorted elements are: 38 27 43 3 9 82 10.

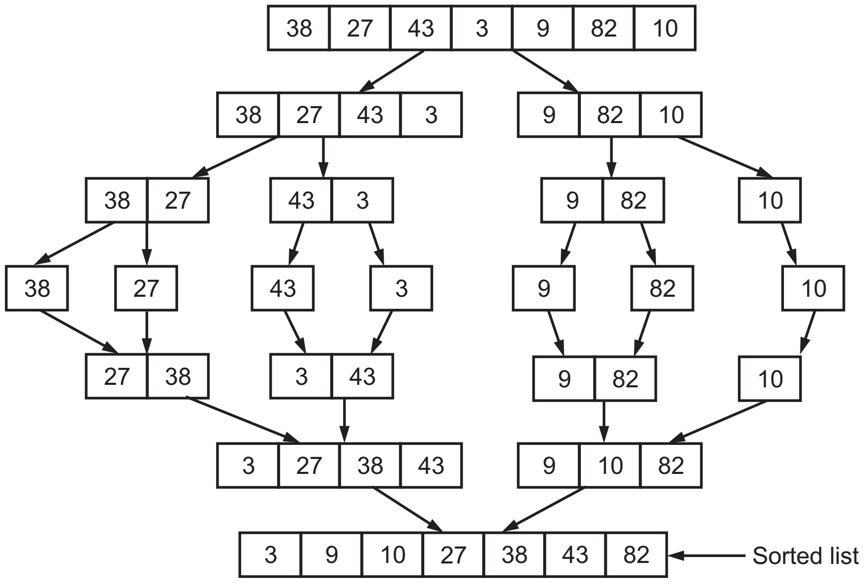


Fig. 2.16

Complexity of Merge Sort:

- Merge sort is having more than $\log_2 n$ iterations. The maximum comparisons are $n \times n \log_2 n$.

Worst case complexity	:	O(n log n)
Best case complexity	:	O(n log n)
Average case complexity	:	O(n log n)

Advantages and Disadvantages:

1. Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.
2. Merge sort is not in place because it requires additional memory space to store the auxiliary arrays.

Program 2.11: To implement merge sort.

```
#include<stdio.h>
void disp();
void mergesort(int,int,int);
void msortdiv(int,int);
int a[50],n;
void main()
{
    int i;
    printf("\nEnter the n value:");
    scanf("%d",&n);
    printf("\nEnter elements for an array:");
    for(i=0;i<n;i++)
       scanf("%d",&a[i]);
    printf("\nBefore Sorting the elements are:\n");
    for(i=0;i<n;i++)
       printf("%d\t",a[i]);
    msortdiv(0,n-1);
    printf("\nAfter Sorting the elements are:\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
}
void mergesort(int low,int mid,int high)
{
    int t[50],i,j,k;
    i=low;
    j=mid+1;
    k=low;
```

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

        while(j<=high)
           t[k++]=a[j++];
        for(i=low;i<=high;i++)
           a[i]=t[i];
    }
    void msortdiv(int low,int high)
    {
        int mid;
        if(low!=high)
        {
           mid=((low+high)/2);
           msortdiv(low,mid);
           msortdiv(mid+1,high);
           mergesort(low,mid,high);
        }
    }
```

Output:

```
Enter the n value:5
Enter elements for an array:
65
49
-12
105
3
Before Sorting the elements are:
65      49      -12      105      3
After Sorting the elements are:
-12      3      49       65        105
```

2.4.5 Quick Sort **[April 16, 17]**

- It is more popular and fastest sorting method. It follows divide and conquer method i.e. numbers are divided and again subdivided and division goes on until it is not possible to divide further the procedure is applied recursively to the two parts of the array, on either side of the pivot element.

- Quick sort is also called partition-exchange sort.
- This algorithm divides the input list into three main parts:
  1. Elements less than the Pivot element.
  2. Pivot element(Central element).
  3. Elements greater than the pivot element.
- Pivot element can be any element from the array, it can be the first element, the last element or any random element.
- Recursive algorithm consists of four steps:
  1. If there is one or less element in the array to be sorted, return immediately.
  2. Pick an element in the array to serve as a "pivot" element, (Usually, the left most element in the array).
  3. Split the array into two parts one with elements smaller than the pivot and the other with elements larger than the pivot.
  4. Recursively repeat the algorithm for both halves of the original array.

Algorithm:

1. In the first pass, the 0th element is the pivot element.
2. To obtain the proper position of the pivot we traverse the list in both the directions using the indices low and high respectively. We initialize low to that index which is one more than index of the pivot element (1st position).
3. The index low is incremented till we get an element at the lowth position greater than pivot element.
4. Similarly, we initialize high to n-1 (last position) and go on decrementing high till we get an element having the value less than the pivot element.
5. We then check whether low and high have crossed each other. If not then we interchange elements at the low and high position and continue the process of incrementing low and decrementing high till low and high cross each other.
6. When low and high cross each other, we interchange the pivot element and element at high position and we find that the elements to the left of pivot element are less than it and the elements to its right are greater than it. Hence, we get splitted list.
7. Repeat the same procedure with each sub-list.

Example 11: Sort the following list using quick sort method:

24, 30, 27, 32, 11, 21, 19

Solution:

24		30	27	32	11	21	19

Select pivot = 24

Step 1 :

	Lbound						Unbound
	↓						↓
	0	1	2	3	4	5	6
a	24	30	27	32	11	21	19
	↑	↑					↑
	Pivot	low					high

pivot = 24, low = 1, high = 6.

Step 2 : a[low] <= pivot ? No.
then check a[high] > pivot ? No.
If both conditions are false then interchange the values of a[low] and a[high].

	0	1	2	3	4	5	6
a	24	19	27	32	11	21	30
	↑	↑					↑
	pivot	low					high

Step 3 : 19 <= 24
low ++

	0	1	2	3	4	5	6
a	24	19	27	32	11	21	30
	↑		↑				↑
	pivot		low				high

Step 4 : 27 <= 24 ? No
30 > 24 ? Yes
high--

	0	1	2	3	4	5	6
a	24	19	27	32	11	21	30
	↑		↑			↑	
	pivot		low			high	

Step 5 : 27 <= 24? No
21 > 24 ? No
Interchange a[low] and a [high]

	0	1	2	3	4	5	6
a	24	19	21	32	11	27	30
	↑		↑			↑	
	pivot		low			high	

Step 6 : a[low] <= pivot ? yes
low++

	0	1	2	3	4	5	6
a	24	19	21	32	11	27	30
	↑			↑		↑	
	pivot			low		high	

Step 7 : 32 <= 24? No
27 > 24? Yes
high --

	0	1	2	3	4	5	6
a	24	19	21	32	11	27	30
	↑			↑	↑		
	pivot			low	High		

Step 8 : 11 >24? No
Interchange a[low] & a [high]

	0		1	2	3	4	5	6
a	24		19	21	11	32	27	30
	↑				↑	↑		
	pivot				Low	high		

Step 9 : 11 < 24 ? Yes
low++

	0	1	2	3	4	5	6
a	24	19	21	11	32	27	30
	↑				↑↑		
	pivot				low high		

Step 10: a[low] <= pivot ? No
a[high] > pivot ? Yes
high --

	0	1	2	3	4	5	6
a	24	19	21	11	32	27	30
	↑			↑	↑		
	pivot			High	low		

Step 11 : if low > high
Interchange a [high] and pivot

	0	1	2		3		4	5	6
a	11	19	21		24		32	27	30

Here we got two partitions.
Recursively apply, similar steps on sublists.

	0	1	2	3	4	5	6
a	11	19	21	24	27	30	32

This is final sorted array.

Time Complexity of Quick sort:

Best case	: O (n log n)
Average case	: : O (n log n)
Worst case	: O (n^2)

Advantages of Quick Sort:

1. This is faster sorting method among all.
2. Its efficiency is also relatively good.
3. It requires relatively small amount of memory.

Disadvantages of Quick Sort:

1. It is complex method of sorting so; it is little hard to implement than other sorting methods.

Program 2.12: To implement quick sort.

```
#include<stdio.h>
void quicksort(int[ ],int,int);
void main( )
{
    int low, high, pivot, t, n, i, j, a[10];
    printf("\nHow many elements you want to sort ? ");
    scanf("%d",&n);
    printf("\nEnter elements for an array:");
    for(i=0; i<n; i++)
        scanf("%d",&a[i]);

    low=0;
    high=n-1;
    quicksort(a,low,high);
    printf("\nAfter Sorting the elements are:");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
}
void quicksort(int a[ ],int low,int high)
{
    int pivot,t,i,j;
    if(low<high)
    {
        pivot=a[low];
        i=low+1;
        j=high;
        while( i < j)
        {
          while(pivot>=a[i] && i<=high)
               i++;
            while ( pivot < a[j] && j >= low)
               j--;
            if(i<j)
            {
                t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
        }
```

```
          a[low]=a[j];
          a[j]=pivot;
          quicksort(a,low,j-1);
          quicksort(a,j+1,high);
      }
  }
```

Output:

```
How many elements you want to sort ? 5
Enter elements for an array:
10
89
-7
29
-1
After Sorting the elements are:-7   -1     10     29     89
```

2.5 ALGORITHM DESIGN STRATEGIES : DIVIDE AND CONQUER STRATEGY

- An algorithm is a set of instructions applied to solve a particular problem. Divide-and-conquer is an algorithm design strategy.
- In the divide and conquer strategy, the problem is divided into several small sub-problems. Then the sub-problems are solved recursively and combined to get the solution of the original problem.
- The divide and conquer approach involves the following steps at each level:

  Step 1 : **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.

  Step 2 : **Conquer** the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.

  Step 3 : **Combine** the solutions to the sub-problems into the solution for the original problem.

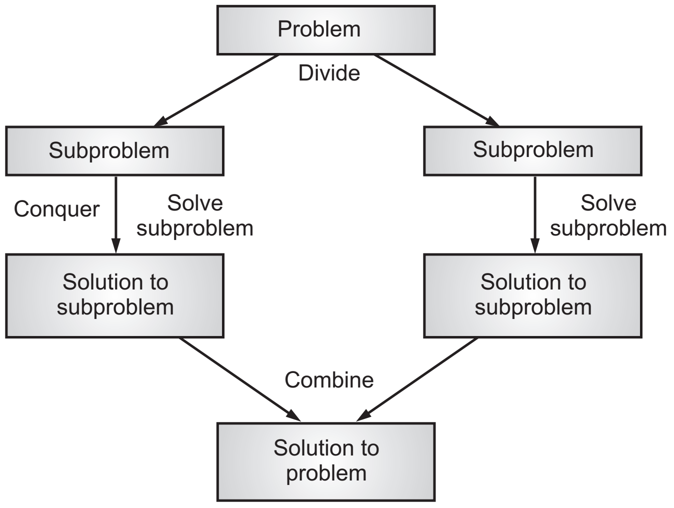


Fig. 2.17

- The divide and conquer approach is applied in the algorithms like Binary search, Quick sort, Merge sort and Strassen's Matrix Multiplication.

  Example:

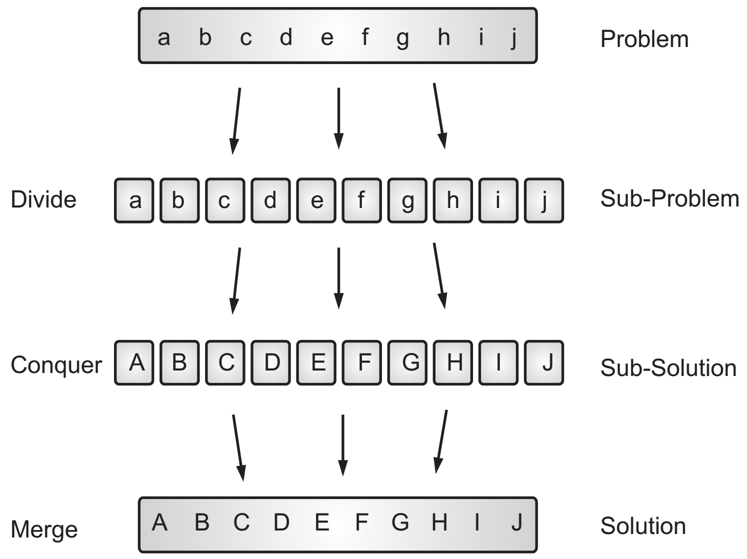


Fig. 2.18

Other Strategies for Algorithm Design:

1. **Greedy Algorithm Design Strategy:**

- An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain.
- As being greedy, the closest solution that seems to provide an optimum solution is chosen.
- A greedy algorithm is very easy to apply to complex problems. It decides which step will provide the most accurate solution in the next step.
- This algorithm is a called greedy because when the optimal solution to the smaller instance is provided, the algorithm does not consider the total program as a whole. Once a solution is considered, the greedy algorithm never considers the same solution again.
- A greedy algorithm works recursively creating a group of objects from the smallest possible component parts.
- Recursion is a procedure to solve a problem in which the solution to a specific problem is dependent on the solution of the smaller instance of that problem.

2. **Backtracking Algorithm Design Strategy:**

- Backtracking is a technique/strategy based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time.
- It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

- Eight queen problem, Sudoku puzzle and going through a maze are popular examples where backtracking algorithm is used.
- In backtracking, we start with a possible solution, which satisfies all the required conditions.
- Then we move to the next level and if that level does not produce a satisfactory solution, we return one level back and start with a new option.

3. **Dynamic Programming Design Strategy:**

- In dynamic programming the problems can be divided into some sub-problems and it stores the output of some previous sub-problems to use them in future.
- Dynamic programming is an optimization technique, which divides the problem into smaller sub-problems and after solving each sub-problem, dynamic programming combines all the solutions to get ultimate solution. Unlike divide and conquer method, dynamic programming reuses the solution to the sub-problems many times.
- Recursive algorithm for Fibonacci Series is an example of dynamic programming.

2.6 COMPLEXITY ANALYSIS OF SORTING METHODS

- Algorithm analysis is a technique used to measure the effectiveness and performance of the algorithms.
- Algorithm analysis helps to determine the quality of an algorithm based on several parameters such as user-friendliness, maintainability, security, space usage and usage of other resources.
- The complexity of sorting algorithm is analyzed using the amount of time required for running the program and the amount of space required for the program.
- To get the amount of time required, we have to estimate the number of comparisons and data movement required to sort the data.
- During the sorted process, the data are traversed many times. Each traversal of the data is referred to as a sort pass. Various sorting methods are analyzed in the cases like – best case, worst case or average case.
- Following table gives complexity analysis for sorting methods:

Sorting Methods	Best Case	Average Case	Worst Case
1. Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
2. Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$
3. Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$
4. Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
5. Quick Sort	$O(N \log N)$	$O(N \log N)$	$O(N^2)$

2.7 NON COMPARISON BASED SORTING

- There are some sorting algorithms that perform sorting without comparing the elements rather by making a certain assumption about the data they are going to sort. The process is known as non-comparison sorting and algorithms are known as the non-comparison based sorting algorithms.

- Non-comparison sorting includes Counting sort which sorts using key-value, Radix sort, which examines individual bits of keys, and Bucket Sort which examines bits of keys.

2.7.1 Counting Sort

- Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers. So, it is an integer sorting algorithm.
- Counting sort is an integer-based sorting algorithm for sorting an array whose keys lies between a specific range.
- It counts the number of elements that have each distinct key value and then use those counts to determine the positions of each key value in the output.
- Counting sort can be used to find most frequent letter in a file or sort a limited range array efficiently.
- It is often used as a subroutine in radix sort sorting algorithm and because of this, it is important for counting sort to be a stable sort.
- Counting sort operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence.
- Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items.
- Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

How Counting Sort Works?

1. Find out the maximum element (let it be max) from the given array.

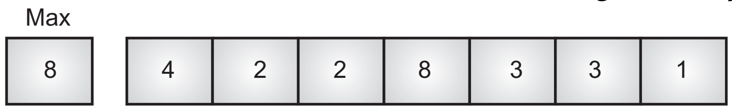


Fig. 2.19: Given Array

2. Initialize an array of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.

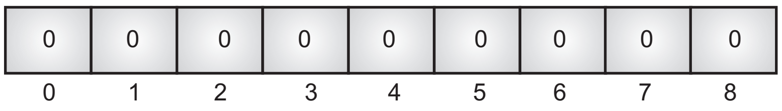


Fig. 2.20: Count Array

3. Store the count of each element at their respective index in count array For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position.

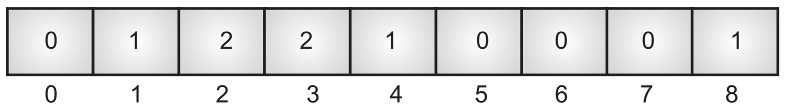


Fig. 2.21: Count of each Element Stored

4. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.

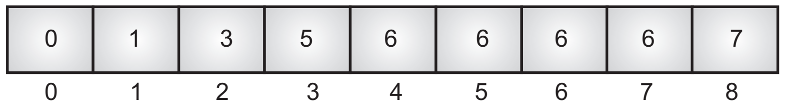


Fig. 2.22: Cumulative Count

5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.

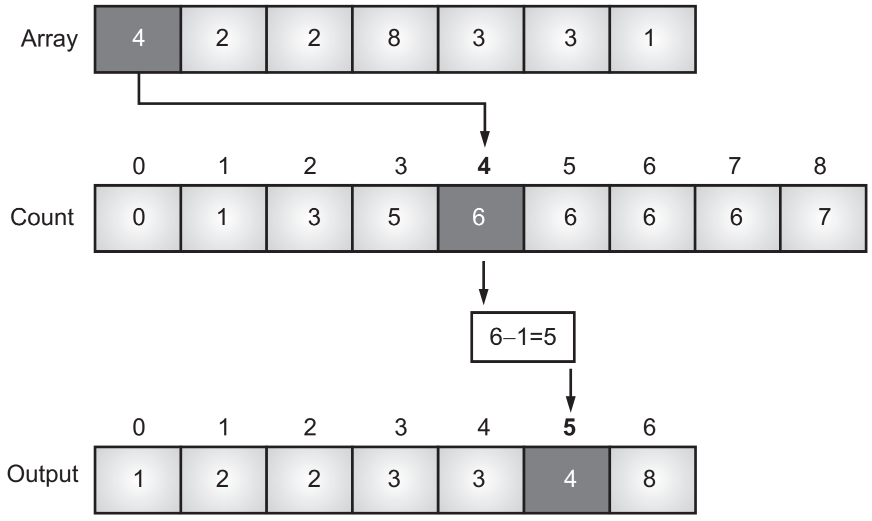


Fig. 2.23: Counting Sort

6. After placing each element at its correct position, decrease its count by one.

Counting Sort Algorithm:

```
countingSort(array, size)
  max <- find largest element in array
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Complexity Analysis:

Complexity	Best Case	Average Case	Worst Case
Time Complexity	Ω(n + k)	θ(n + k)	O(n + k)

Advantages and Disadvantages:

1. Counting sort is faster than other comparison-based sorting techniques.
2. Counting sort is a stable sorting algorithm. It keeps the number with the same value in the output as they were in the input array.
3. Counting sort is simple to code. Counting sort can be used to sort negative inputs also.
4. Counting sort is not a comparison-based sorting algorithm. It counts the frequency of each value in the input.
5. Counting sort operated only on Integers.
6. Counting sort performs its best when the number of integers to be sorted is not large. It works best on smaller integers.
7. Counting sort requires O(n+k) extra storage to store temporary data values.

Program 2.13: Program for counting sort.

```
#include <stdio.h>
#include <conio.h>
void Counting_sort(int [], int, int);
void main()
{
 int n,i,k=0,A[15];
   clrscr();
   printf("\n\n

   \t\t\t----------Counting Sort----------\n\n\n");
   printf("Enter the number of input : ");
   scanf("%d",&n);
   printf("\nEnter the elements to be sorted :\n");
   for (i=1;i<=n;i++)
   {
      scanf("%d",&A[i]);
      if(A[i] > k)
      {
         k = A[i];
      }
   }
   Counting_sort(A, k, n);
   getch();
 }
```

```
void Counting_sort(int A[], int k, int n)
{
  int i, j;
  int B[15], C[100];
  for(i = 0; i <= k; i++)
     C[i] = 0;
  for(j =1; j <= n; j++)
     C[A[j]] = C[A[j]] + 1;
  for(i = 1; i <= k; i++)
     C[i] = C[i] + C[i-1];
  for(j = n; j >= 1; j--)
  {
     B[C[A[j]]] = A[j];
     C[A[j]] = C[A[j]] - 1;
  }
  printf("\t\t\t----Sorted Array Using Counting Sort----\n\n\n" );
  printf("\nThe Sorted array is : ");
  for(i=1;i<=n;i++)
  {
     printf("\t");
     printf("%d",B[i]);
  }
}
```

Output:

```
                        ----- Counting Sort -----
Enter the number of input : 6
Enter the elements to be sorted :
2
4
3
0
2
1
               ----- Sorted Array Using Counting Sort -----
The sorted array is :  0     1      2       2       3       4
```

2.7.2 Radix Sort

- Radix sort it is a non-comparison based algorithm suitable for integer values to be sorted. It sorts the elements digit by digit.
- Radix sort starts sorting the elements according to the least significant digit of the elements. This partially sorted list is then sorted on the basis of second least significant bit and so on.
- Bucket/radix sort method is generally used in that case, when we want to sort the names in alphabetical order.
- Radix sort is not an in-place sorting algorithm as it requires extra additional space.
- In following algorithm, first find the largest element of the array (e.g. 123) and we run loop until we reach the largest digit place of that number. Store the count of digits in count[]array.
- Change count[i] so that count[i] now contains actual position of this digit in result[]. Build the resultant array. Store final resultant array into main array A[] which contains sorted numbers according to current digit place.

Radix Sort Algorithm:

- Each key in A[1..n] is a d-digit integer.

```
1.  get largestNum from array
2.  while(largestNum/digitPlace>0)
3.  int count[10] = {0};
4.  for i = 0 to n do
5.      count[key of(A[i]) in pass j]++
6.  for k = 1 to 10 do
7.      count[k] = count[k] + count[k-1]
8.  for i = n-1 downto 0 do
9.      result[ count[key of(A[i])] ] = A[j]
10.     count[key of(A[i])]--
11. for i=0 to n do
12.     A[i] = result[i]
13.     digitPlace *= 10;
```

- **Example:** Let given input string is 468, 537, 9, 721, 13, 35, 123, 27, apply radix sort algorithm on it. Since, largest number is 721 and its digits are 3 in number then total number of passes will be 3.

Pass 1 : Bucket sort the numbers while scanning the input from left to right by **one's digit** (least significant digit).

0	1	2	3	4	5	6	7	8	9
	721		13 123		35		537 27	468	9

Remove the elements from the buckets from 0 to 9 in first in first out manner. After first pass 721, 13, 123, 35, 537, 27, 438, 9.

Pass 2 : Bucket sort the numbers while scanning the input from left to right by **ten's digit** (next least significant digit).

0	1	2	3	4	5	6	7	8	9
9	13	721 123 27	35 537 438						

Remove the elements from the buckets from 0 to 9 in first in first out manner. After second pass 9, 13, 721, 123, 27, 35, 537, 438.

Pass 3 : Bucket sort the numbers while scanning the input from left to right by hundred's digit (most significant digit).

0	1	2	3	4	5	6	7	8	9
9 13 27 35	123			438	537		721		

Remove the elements from the buckets from 0 to 9 in first in first out manner. After third pass 9, 13, 27, 35, 123, 438, 537, 721. Numbers are sorted now.

Complexity of Radix Sort:

Complexity	Best Case	Average Case	Worst Case
Time Complexity	Ω(n + k)	θ(nk)	O(nk)
Space Complexity			O(n + k)

Advantages and Disadvantages of Radix Sort:

1. Radix sort algorithm is well known for its fastest sorting algorithm for numbers and even for strings of letters.
2. Radix sort algorithm is the most efficient algorithm for elements which are arranged in descending order in an array.
3. Radix sort is much less flexible than other sorts. Hence, for every different type of data it needs to be rewritten.
4. Radix sort takes more space compared to quick sort.
5. It has poor efficiency for most elements which are already arranged in ascending order in an array.

Program 2.14: Program for radix/bucket sort.

```
#include<stdio.h>
#include<stdlib.h>
// This  function gives maximum value in array[]
int getMax(int A[], int n)
{
    int i;
    int max = A[0];
    for (i = 1; i < n; i++){
        if (A[i] > max)
            max = A[i];
    }
    return max;
}
// Main Radix Sort sort function
void radixSort(int A[], int n)
{
    int i,digitPlace = 1;
    int result[n]; // resulting array
    // Find the largest number to know number of digits
    int largestNum = getMax(A, n);
    //we run loop until we reach the largest digit place
    while(largestNum/digitPlace >0)
     {
        int count[10] = {0};
        for (i = 0; i < n; i++)
            count[ (A[i]/digitPlace)%10 ]++;
        for (i = 1; i < 10; i++)
            count[i] += count[i - 1];
        for (i = n - 1; i >= 0; i--)
        {
            result[count[ (A[i]/digitPlace)%10 ] - 1] = A[i];
            count[ (A[i]/digitPlace)%10 ]--;
        }
        for (i = 0; i < n; i++)
            A[i] = result[i];
        digitPlace *= 10;
    }
}
int main()
{
    int arr[20],n,i;
    setbuf(stdout, NULL);
    printf("Enter size of Array: ");
    scanf("%d",&n);
    printf("\nEnter %d elements:\n\n",n);
    for(i=0 ; i<n ; i++)
    {
     printf(" Array[%d] = ",i);
     scanf("%d",&arr[i]);
    }
```

```
        radixSort(arr, n);
        printf("\nThe Sorted Array is:\n\n");
        for(i=0 ; i<n ; i++)
        {
         printf(" %4d",arr[i]);
        }
        return 0;
    }
```

Output:

```
Enter size of Array: 5
Enter 5 elements:
Array[0] = 121
Array[1] = 70
Array[2] = 965
Array[3] = 432
Array[4] = 12
The Sorted Array is:
12   70   121  432  965
```

2.8 COMPARISON OF SORTING METHODS

- Time and space complexity comparison table for different sorting methods:

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	$\theta(N^2)$	$O(N^2)$	O(1)
Selection Sort	$\Omega(N^2)$	$\theta(N^2)$	$O(N^2)$	O(1)
Insertion Sort	Ω(N)	$\theta(N^2)$	$O(N^2)$	O(1)
Merge Sort	Ω(N log N)	θ(N log N)	O(N log N)	O(N)
Quick Sort	Ω(N log N)	θ(N log N)	$O(N^2)$	O(N log N)
Radix Sort	Ω(N k)	θ(N k)	O(N k)	O(N + k)
Count Sort	Ω(N + k)	θ(N + k)	O(N + k)	O(k)

PRACTICE QUESTIONS

Q.I Multiple Choice Questions:

1. Which of the following data structure store the homogeneous data elements?
   (a) Array (b) Linked list
   (c) Tree (d) None of the above

2. Arrays are best data structures _____.
   (a) for relatively permanent collections of data
   (b) the size of the structure and the data in the structure are constantly changing
   (c) for both of above situation
   (d) None of the above situation
3. The memory address of the first element of an array is called _____.
   (a) floor address (b) foundation address
   (c) first address (d) base address
4. The memory address of the fifth element of an array can be calculated by the formula _____.
   (a) LOC(Array[5]=Base(Array)+w(5-lower bound), where w is the number of words per memory cell for the array
   (b) LOC(Array[5])=Base(Array[5])+(5-lower bound), where w is the number of words per memory cell for the array
   (c) LOC(Array[5])=Base(Array[4])+(5-Upper bound), where w is the number of words per memory cell for the array
   (d) None of the above
5. Which of the following expressions access the (i, j)th of a $m \times n$ matrix stored in column major form?
   (a) $n \times (i - 1) + j$ (b) $m \times (j - 1) + i$
   (c) $m \times (n - j) + j$ (d) $n \times (m - i) + j$
6. The smallest element of an array's index is called its _____.
   (a) Lower bound (b) Upper bound
   (c) Range (d) Extraction
7. Which of the following is also called as partition exchange sort?
   (a) Bubble (b) Insertion
   (c) merge (d) Quick
8. What is an internal/in-place sorting algorithm?
   (a) Algorithm that uses tape or disk during the sort
   (b) Algorithm that uses main memory during the sort
   (c) Algorithm that involves swapping
   (d) Algorithm that are considered 'in place'
9. What is an external/out-place sorting algorithm?
   (a) Algorithm that uses tape or disk during the sort
   (b) Algorithm that uses main memory during the sort
   (c) Algorithm that involves swapping
   (d) Algorithm that are considered 'in place'

10. A sorting technique is called stable if it _____.
    (a) Takes O(n log n) times
    (b) Maintains the relative order of occurrence of non-distinct elements
    (c) Uses divide-and-conquer paradigm
    (d) Takes O(n) space
11. Which of the following is not stable sort?
    (a) Bubble sort (b) Insertion sort
    (c) Merge sort (d) Quick sort
12. Which of the following is not an internal sorting algorithm?
    (a) Bubble sort (b) Insertion sort
    (c) Merge sort (d) Quick sort
13. Which of the following is example of an external sorting algorithm?
    (a) Bubble sort (b) Insertion sort
    (c) Merge sort (d) Quick sort
14. The complexity of Bubble sort algorithm is _____.
    (a) O(n) (b) O(log n)
    (c) $O(n^2)$ (d) O(n log n)
15. If the given input array is sorted or nearly sorted, which of the following algorithm gives the best performance?
    (a) Insertion sort (b) Quick sort
    (c) Merge sort (d) None of the above
16. The process of finding the existence of a particular data item in a list is known as ______.
    (a) searching (b) sorting
    (c) merging (d) All of these
17. ______ is an operation in which all the elements of a list are arranged in a predetermined order.
    (a) searching (b) sorting
    (c) merging (d) All of these
18. In which strategy the problem in hand, is divided into smaller sub-problems and then each problem is solved independently.
    (a) divide and conquer (b) Greedy
    (c) Backtracking (d) All of these
19. Non-comparison based algorithm includes.
    (a) Radix sort (b) Count sort
    (c) Bucket sort (d) All of these

20. Following which algorithm is a non-comparative sorting algorithm and most preferred algorithm for the unsorted list.
    (a) Radix sort (b) Count sort
    (c) Bucket sort (d) All of these

Answers

1. (a)	2. (a)	3. (d)	4. (a)	5. (b)	6. (a)	7. (d)	8. (b)
9. (a)	10. (b)	11. (d)	12. (c)	13. (c)	14. (b)	15. (a)	16. (a)
17. (b)	18. (a)	19. (d)	20. (a)				

Q.II Fill in the Blanks:

1. _____ a kind of data structure that can store a fixed-size sequential collection of elements of the same type.
2. The representation of a two-dimensional (2D) array in memory is not like the grid-like structure of a _____.
3. _____ search is a method for finding a particular value in an array list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.
4. The process of arranging data in some logical order is known as _____.
5. The complexity of sorting algorithm calculates the _____ time of a function in which 'n' number of items are to be sorted.
6. If all the data that is to be sorted can be adjusted at a time in the main memory, the _____ sorting method is being performed.
7. _____ sort is sorting algorithm is an in-place comparison-based algorithm.
8. The _____ sort manipulates the elements to be sorted within the array or list space that contained original unsorted input.
9. _____ sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
10. In _____ strategy, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.
11. _____ sort is an in-place comparison-based sorting algorithm.
12. _____ sort is a non-comparison-based algorithm.

Answers

1. Arrays	2. matrix	3. Linear	4. sorting	5. running	6. internal
7. Selection	8. in-place	9. Quick	10. divide and conquer	11. Insertion	12. Radix

Q.III State True or False:

1. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
2. Searching refers to the operation of arranging a set of data in some given manner.
3. A binary search is a sorting algorithm, that is used to search an element in a sorted array.
4. Bubble sort is a sorting algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
5. The quicksort algorithm is a sorting algorithm that works by selecting a pivot point, and thereafter partitioning the number set, or array, around the pivot point.
6. Selection sort is also known as partition-exchange sort.
7. Merge sort is a sorting technique based on divide and conquer technique.
8. When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.
9. A binary search searches an element or value from an array till the desired element or value is not found and it searches in a sequence order.
10. A sorting technique is stable if it does not change the order of elements with the same value.
11. Quick sort is an algorithm based on divide and conquer approach in which the array is split into subarrays and these sub-arrays are recursively called to sort the elements.
12. The divide and conquer technique decomposes complex problems recursively into smaller sub-problems. Each sub-problem is then solved and these partial solutions are recombined to determine the overall solution.
13. All array elements are stored in a contiguous block of computer memory.
14. Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order.
15. Radix sort is a sorting algorithm that sorts items by scanning individual digits. It does not involve comparison between the items being sorted.

Answers

1. (T)	2. (F)	3. (T)	4. (T)	5. (T)	6. (F)	7. (T)	8. (T)
9. (F)	10. (T)	11. (T)	12. (T)	13. (T)	14. (T)	15. (T)	

Q.IV Answer the following Questions:

(A) Short Answer Questions:

1. What is an array?
2. List types of arrays.
3. What is sorting?
4. What is searching?
5. List variants for linear search.
6. What is meant binary search?
7. What is sorting algorithm?
8. What is merge sort?
9. What is insertion sort?
10. List various types of sorting algorithms.
11. What is complexity of bubble sort?
12. What are the examples of internal sorting?
13. Define the following terms:
    (i) Sorting.
    (ii) Searching.
    (iii) In-place sorting.
    (iv) Stable sorting.
14. List operations on array data structures.
15. List algorithm design strategies.

(B) Long Answer Questions:

1. Define array. How to declare it? Explain with its memory representation.
2. With the help of example explain linear search.
3. Describe the term binary search with its advantages.
4. What quick sort? Describe with example.
5. What is 2D array? Explain its row and column representations.
6. Compare sequential and binary search.
7. Write short note on: Merge sort.
8. With suitable example describe selection sort.
9. With the help of example describe bubble sort.
10. Explain time complexity various sorting algorithms.
11. Describe divide and conquer strategy of algorithm with example.
12. With the help of example describe insertion sort. Also state its advantages and disadvantages.
13. Explain quick sort with example.
14. How to insert and delete operations on arrays? Explain in detail with program.

15. Sort the following numbers using bubble sort method:
    (a) 108, 3, 97, 65, 71, 23, 57, 93, 100
    (b) 55, 22, 66, 33, 44, 11
16. Write an algorithm for Merge Sort. Give the time complexity of your algorithm. Show the stepwise execution of the algorithm for the following list of data.
    24, 11, 9, 2, 6, 5, 4, 3
    10, –5, 0, 20, –15, 50, 40, –20, 30
17. Sort the following sequence using quick sort algorithm.
    35, 45, 42, 57, 26
18. What is the best case and worst case efficiency of the following algorithm?
    (a) Bubble sort
    (b) Insertion sort
    (c) Quick sort.
19. Sort the following numbers using bubble sort method:
    108, 3, 97, 65, 71, 23, 57, 93, 100
20. Sort the following data using bubble sort:
    55, 22, 66, 33, 44, 11
21. Sort the given array using quick sort algorithm: 35, 45, 42, 57, 26.
22. Write short note on: Array as an ADT.
23. Describe sentinel search, probability search, ordered list search with example.
24. Compare different sorting methods.

UNIVERSITY QUESTIONS AND ANSWERS

April 2015

1. What is an advantage of a Binary search over linear search? **[1 M]**
Ans. Refer to Section 2.2.3.
2. Sort the following data using merge sort method: 22, 4, 6, 13, 12, 18, 27 **[5 M]**
Ans. Refer to Section 2.2.4.
3. Write an algorithm of Linear Search. Also state its best case and worst case time complexity. **[3 M]**
Ans. Refer to Section 2.2.1.

April 2016

1. What is time complexity of merge sort? **[1 M]**
Ans. Refer to Section 2.2.4.
2. Show all the steps of sorting the data using quick sort: 24, 30, 27, 32, 11, 21, 19. **[5 M]**
Ans. Refer to Section 2.4.5.
3. Write the steps for creating a binary search tree for the data: 13, 4, 25, 3, 21, 20, 7. **[3 M]**
Ans. Refer to Section 2.2.3

October 2016

1. Sort the following elements using Bubble Sort:
(Write all passes) 91, 21, 41, 71, 51, 31, 81. **[5 M]**

Ans. Consider the elements 91, 21, 41, 71, 51, 31, 81:

Pass i = 1

j = 0	91	21	41	71	51	31	81
j = 1	21	91	41	71	51	31	81
j = 2	21	41	91	71	51	31	81
j = 3	21	41	71	91	51	31	81
j = 4	21	41	71	51	91	31	81
j = 5	21	41	71	51	31	91	81
j = 6	21	41	71	51	31	81	$\boxed{91}$

Pass i = 2

j = 0	21	41	71	51	31	81	91
j = 1	21	41	71	51	31	81	91
j = 2	21	41	71	51	31	81	91
j = 3	21	41	51	71	31	81	91
j = 4	21	41	51	31	71	81	91
j = 5	21	41	51	31	71	$\boxed{81}$	$\boxed{91}$

Pass i = 3

j = 0	21	41	51	31	71	81	91
j = 1	21	41	51	31	71	81	91
j = 2	21	41	51	31	71	81	91
j = 3	21	41	31	51	71	81	91
j = 4	21	41	31	51	$\boxed{71}$	$\boxed{81}$	$\boxed{91}$

Pass i = 4

j = 0 $\underbrace{21 \quad 41} \quad 31 \quad 51 \quad 71 \quad 81 \quad 91$

j = 1 $21 \quad \underbrace{41 \quad 31} \quad 51 \quad 71 \quad 81 \quad 91$

j = 2 $21 \quad 41 \quad \underbrace{31 \quad 51} \quad 71 \quad 81 \quad 91$

j = 3 $21 \quad 41 \quad 31 \quad \boxed{51} \quad \boxed{71} \quad \boxed{81} \quad \boxed{91}$

Pass i = 5

j = 0 $\underbrace{21 \quad 31} \quad 41 \quad 51 \quad 71 \quad 81 \quad 91$

j = 1 $21 \quad \underbrace{31 \quad 41} \quad 51 \quad 71 \quad 81 \quad 91$

j = 2 $21 \quad 31 \quad \boxed{41} \quad \boxed{51} \quad \boxed{71} \quad \boxed{81} \quad \boxed{91}$

Pass i = 6

j = 0 $\underbrace{21 \quad 31} \quad 41 \quad 51 \quad 71 \quad 81 \quad 91$

j = 1 $21 \quad \boxed{31} \quad \boxed{41} \quad \boxed{51} \quad \boxed{71} \quad \boxed{81} \quad \boxed{91}$

April 2017

1. Define stable sorting. **[1 M]**

Ans. Refer to Section 2.3.1.3.

2. How are the elements of an array stored in memory? **[1 M]**

Ans. Refer to Section 2.0.

3. Sort the following data using quick sort: 12, 24, 9, 46, 31, 53, 33. **[5 M]**

Ans. Refer to Section 2.4.5.

4. Write an algorithm for binary search. Also state its complexity. **[3 M]**

Ans. Refer to Section 2.2.3.

October 2017

1. What is time complexity of bubble sort? **[1 M]**

Ans. Refer to Section 2.4.1.

2. Sort following data using Bubble sort: 32, 51, 85, 66, 23, 13, 10, 57. **[5 M]**

Ans. Refer to Section 2.4.1.

3. Sort the numbers using insertion sort method: 30, 40, 10, 50, 25, 35, 15. **[3 M]**

Ans. Refer to Section 2.4.2.

April 2018

1. What is worst and best time complexity of merge sort? **[1 M]**

Ans. Refer to Section 2.2.4.

2. Sort the following elements using Insertion Sort (write passes):
23, 6, 18, 29, 27, 4, 13. **[5 M]**

Ans. Refer to Section 2.4.2.

3. Write an algorithm for Binary search. **[3 M]**

Ans. Refer to Section 2.2.3.

■■■

CHAPTER 3

Linked List

Objectives …

- To study Basic Concepts of Linked List
- To learn Linked List Representations (Static and Dynamic)
- To study different Operations on Linked List
- To understand Types of Linked List
- To learn various Applications of Linked List
- To study Concepts of Polynomial Manipulation

3.0 INTRODUCTION

- The linked list is one of the most widely used data structures in C language. A linked list is a dynamic data structure. A linked list is a linear collection of data elements.
- In array data structure, space cannot be increased when required. It is static data structure. This drawback is removed with another linear data structure called as linked list.
- The arrays are static once the array is declared; its size remains the same. Linked list can be expanded or grows as per requirement.
- The memory allocated for array may be extra or insufficient. Linked list is more efficient in memory management.
- Array elements are stored in successive memory locations. Linked list data structure uses a pointer to point the next node which indicates that elements are logically contiguous.
- Insertion and deletion operations on array are time consuming because it requires shifting of elements. In linked list insertion and deletion are easy due to pointers.

3.1 LIST AS A DATA STRUCTURE [April 15, 18; Oct. 15]

- The term 'list' is refer to a linear collection of data items. A list is a series of linearly arranged finite elements (numbers) of same data type.
- A list can be defined as, "a collection of elements". A linked list is a linear collection of data elements called nodes, each pointing to the next node by means of pointers.
- In the list the elements are positioned one after the other and their position numbers appear in the sequence. The first element of the list is called as head and the last element is called as tail, (See Fig. 3.1).

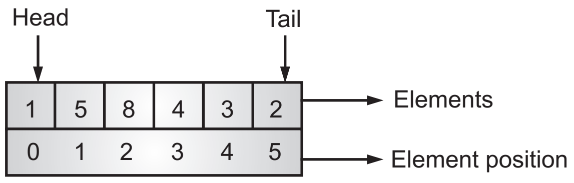


Fig. 3.1: Static List

- As shown in the Fig. 3.1, the element 1 is at head position (0^{th}) and element 2 is at tail position (5^{th}). The element 5 is predecessor of element 8 and 4 is successor.

Properties of List:

1. The list can be enlarged or reduced from both the ends.
2. The tail (ending) position of the list depends on how long the list is extended by the user.
3. Various operations such as transverse, insertion and deletion can be performed on the list.
4. The list can be implemented by applying static (array) or dynamic (pointer) implementation.

- A linked list is a linear data structure. A linked list is a list in which address/location (link) of next (previous) element is embedded in the current element.
- A linked list is a linear collection of data items called nodes, where the linear order is given by means of pointers. In linked list, adjacency between the elements are maintained by means of links or pointers.
- A link or pointer actually is the address (memory location) of the subsequent element. Thus, in a linked list, data (actual content) and link (to point to the next data) both are required to be maintained.

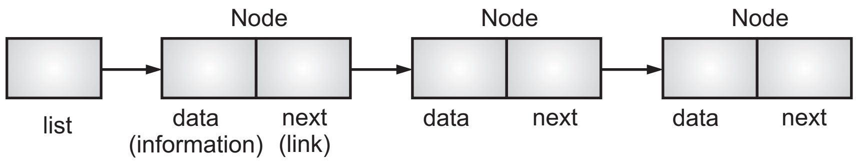


Fig. 3.2: Linear Linked List

- A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called HEAD. The last node points to NULL.

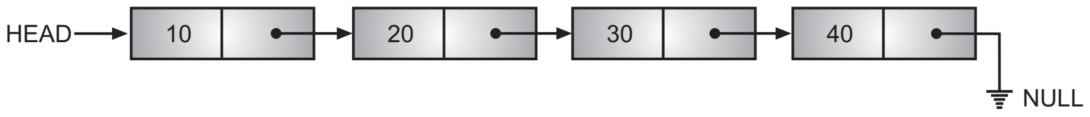


Fig. 3.3: Example of Linked List

3.1.1 Difference between Array and Linked List

- Following table gives difference between array and linked list:

Sr. No.	Parameters	Array	Linked List
1.	Definition	Array is a collection of elements of similar data type.	Linked list is an ordered collection of elements of same type, which are connected to each other using pointers.
2.	Accessing the element	Array supports random access hence nth element can be accessed directly by specifying its index.	Linked list supports sequential access hence to access nth element of a linked list, one has to travel entire list.
3.	Memory location	Elements are stored in contiguous memory location or consecutive manner in the memory.	New elements can be stored anywhere in the memory.
4.	Operations	In array, insertion and deletion operation takes more time, as the memory locations are consecutive and fixed.	Insertion and deletion operations are fast in linked list.
5.	Memory allocation	Memory is allocated as soon as the array is declared, at compile time. It is also known as static memory allocation.	Memory is allocated at runtime, as and when a new node is added. It is also known as dynamic memory allocation.
6.	Types	Array can be 1-dimensional, 2-dimensional or multidimensional.	Linked list can be singly, doubly or circular.
7.	Size	Size of the array is fixed must be known in advance.	Size of a linked list is variable.
8.	Nature	Array is an static data structure	Linked list is an dynamic data structure.

3.1.2 Definition of Linked List

- A linked list is ordered collection of nodes in which each node contains the data and link to its successor (and predecessor).
- A linked list can be defined as, "an ordered collection of finite homogeneous data elements called as nodes where the linear order is maintained by means of links or pointers".
- Each node of the list has two elements:
  1. The data being stored in the list.
  2. A pointer to the next node in the list.

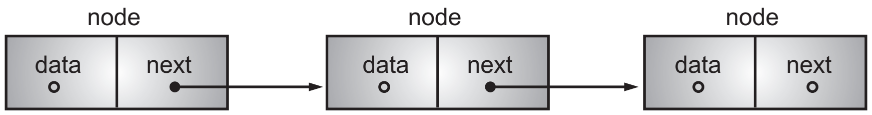


Fig. 3.4: Structure of Linked List

- The last node in the list contains a NULL pointer to indicate that it is the end or tail of the list.
- As data is added to a list, memory for a node is dynamically allocated. Thus, the number of nodes that may be added to a list is limited only by the amount of memory available.
- **Example:** We all know about very curious game "Treasure Hunt". In this, player is provided the initial clue of the first location. From first location clue, player gets the second and so on. To reach the final destination, the player has to pass through each and every location in the specific order. Even if the order of one of the locations is wrong, one will not get the clue for reaching the next location and hence, the player will not reach to the final destination. This is the concept of linked list.

3.1.3 Terminology in Linked List

- Basic terms used in linked list data structure are:
  1. **Node:** Each item in the linked list is called a node and contains two fields an information field (INFO/DATA) and a next address (LINK) field.

     An element in a linked list is specially termed as node, which can be viewed as shown in Fig. 3.5. A node consists of two fields namely DATA (to store the actual information or data value) and LINK (to point to the next node).

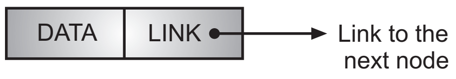


Fig. 3.5: Node (An Element in a Linked List)

A node in a linked list is usually a structure in C and can be declared as follows:

[April 17; Oct. 17]

```
struct node
{
   int data;
   struct node *link;
}*start=NULL;
```

Each node has two parts DATA (contains data element) and LINK (contains the address of the next node).

The LINK part of the last node is of the list contains a NULL value indicating the end of the list. The beginning of the list is indicated with the help of a special pointer called FIRST. Similarly, the end of the list can be indicated by a pointer called LAST.

Fig. 3.6 shows the logical representation of a linked list.

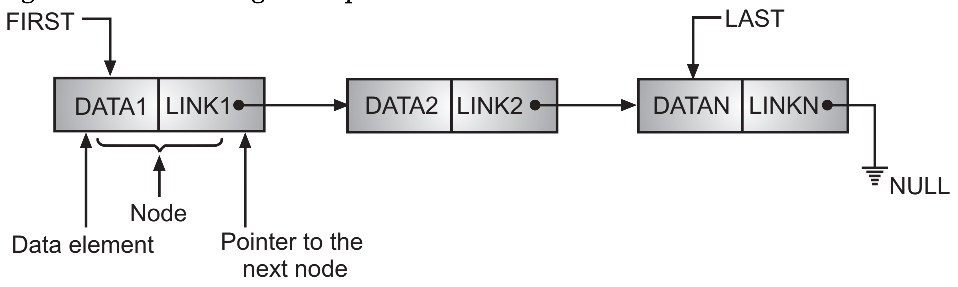


Fig. 3.6

2. **Information:** The information field holds the actual element or data on the list.
3. **Address:** A link or pointer actually is an address (memory location) of the subsequent element.
4. **Pointer:** The variable that stores the reference to another variable is what we call a *pointer*.
5. **Link:** The link field contains the address of next node in the list. It is also called as pointer which points to the next element in the list.
6. **Null Pointer:** The linked field of the last node contains Null rather than a valid address. It is a null pointer and indicates the end of the list. Null pointer is a pointer containing 0 address. P=NULL; The statement will set the pointer to NULL or 0 address.

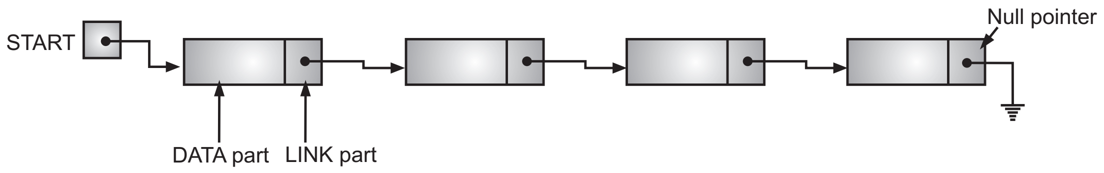


Fig. 3.7: NULL Pointer

7. **External Pointer:** It is a pointer to the very first node in the linked list, it enables us to access the entire linked list.
8. **Empty List:** If the nodes are not present in a linked list, then it is called an empty linked list or simply empty list. It is also called the null list. The value of the external pointer will be zero for an empty list.

3.1.4 Linked Organization Characteristics

- In linked list elements can be placed anywhere in the memory. Dynamic allocation i.e., size need not be known in advance in linked list.
- Insertion and deletion operations of linked list of data do not require any data movement. Each node in liked list is a collection of data and a link to its successor.
- Linked organization needs use of pointers and dynamic memory allocation. The only overhead is, need of additional space for the linked field of each element.

- **Example:** Link list with four nodes.

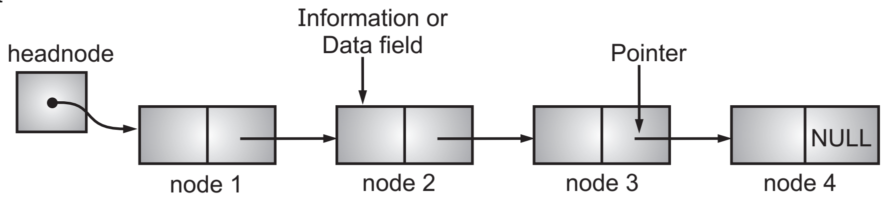


Fig. 3.8: Linked List with 4 Nodes

- Each node has two fields namely, Data field and Pointer field. Data field contain an information. Pointer field contains the address of next node.
- A linked list must always have at least one pointer pointing to first node (head) of the list. This pointer is must otherwise we have no way to access the linked list. This pointer is called head node.
- A special case is the list with no nodes, it is called empty list or null list. In this case head = NULL.
- The pointer of the last node of the linked list contains a value NULL. Every node must contain at least one link field.

3.1.5 Operations on Linked List

- Primitive operations on linked list are given as below:
  1. Create a linked list.
  2. Traverse a linked list.
  3. Insert a node in the linked list.
  4. Delete a node in the linked list.
- Few more operations are:
  1. Searching a node.
  2. Updating node.
  3. Counting length.
  4. Reverse the linked list.
  5. Sort the linked list.
  6. Concatenate two linked lists.
  7. Merge two sorted linked list.

Advantages of Linked List:

1. **Linked Lists are Dynamic Data Structures:** Unlike array, linked list can grow or shrink during the execution of a program.
2. **Efficient Memory Utilization:** Here memory is not defined (pre-allocated). Memory is allocated whenever it is required. And it is de-allocated (removed) when it is no longer needed.
3. **Insertion and Deletions are Easier and Efficient:** Linked lists provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. **Easy for Complex Application:** Many complex applications can be easily carried out with linked lists.

Disadvantages of Linked List:

1. **More Memory Required:** A linked list element requires more memory space because it also has to store address of next element in the list.
2. **Time Consuming:** Access to an arbitrary data item is little bit cumbersome and also time consuming.
3. **Difficult to Access Elements:** Accessing an element is little difficult in linked list than array as there is no index associated with each element. Thus to access a particular element it is mandatory to traverse all elements before it.

3.2 IMPLEMENTATION OF LINKED LIST

- Linked list can be implemented as:
  1. Static Implementation and
  2. Dynamic Implementation.

3.2.1 Static Representation of Linked List (Using Array)

- Static implementation can be implemented using arrays.
- A static data structure is an organization or collection of data in memory that is fixed in size. This results in the maximum size needing to be known in advance, as memory cannot be reallocated at a later point.
- **Example:** Following example elaborates concept of static linked list. Consider the one dimensional array A and the list L = {A, R, P, K, G, S, V, M} which is ordered set.
- The linked representation of the list L using array is shown in Fig. 3.9. The elements are not stored in contiguous block of locations as well as not stored in same order as in the list L.

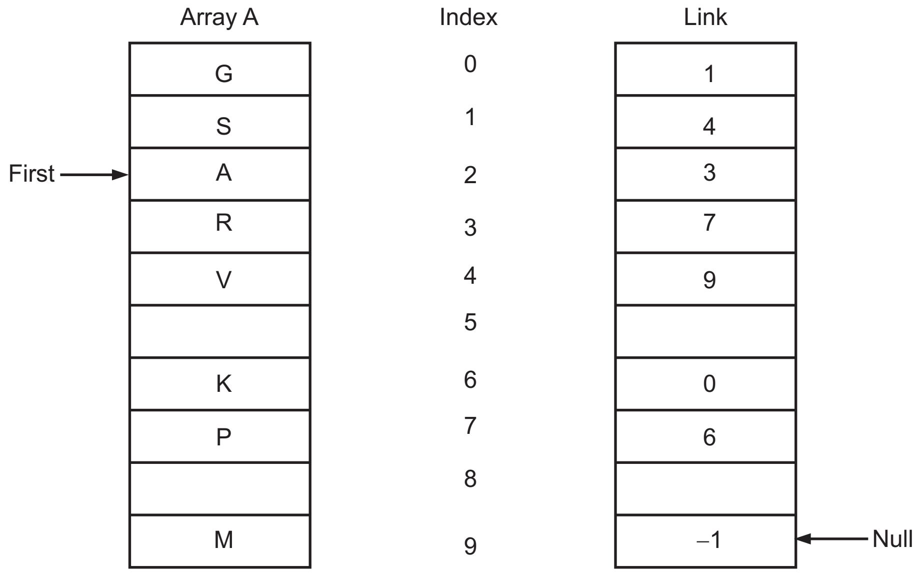


Fig. 3.9: Representation of Linked List using Array

- To maintain the order, the second array, 'Link' is used. The values in this array are the links. Here, the list start at 2nd location. A[first] = A. The second element stored at Link[First] = Link[2]. Hence, A[Link[First]] = A[Link[2]] = A[3] = R. The third element we get through 2nd element i.e. A[Link[3]] = A[7] = P and so on. The value of the last element set to -1 to represent end of list.
- Fig. 3.10 shows the above representation is different manner, after omitting unused locations.

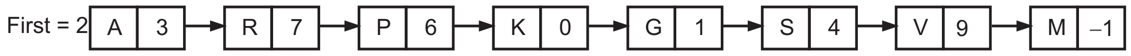


Fig. 3.10: Linked Organization

- The **node structure** of above can be define as:

```
#define max 20
struct node
{
  int data;
  int link;
};
struct node list[max];
```

- In above structure declaration a link field is actually array index, where the next element is stored.
- Let us discuss functions for the operations of the list:

1. **Create:**

/* create a list using array */

```
void create(struct node list[max]);
{
   int first = -1;
   int prev, current, i;
   for (i = 0; i < max; i++)
   {
      list[i].data = 0;
      list[i].link = -1;     //create an empty list by setting all links to -1
   }
   while(1)
   {
      printf("Enter the element /n");
      if(i == max)
      {
         printf("Array is full");
         exit(0);  /* Check array is full otherwise insert */
      }
```

```
            scanf ("%d", & list[i].data)
            /* check whether first is - 1 */
            current = i;
            if(first == - 1)
            {
               first = prev= current;
            }
            else
            {
               list[prev].link = current;
               prev = current;
            }
         } //while end
      } //function end
```

2. Insert a node:

```
   void Insert (struct node list[max], int first, int a, int b)
   {
      int i, j, prev, current, temp;
      for (i = 0; i < max; i++)
      {
         if(list[i].data == 0) //search for empty location
         break;
      }
      list[i].data = a;
      current = i;
      temp = first;
      while (temp ! = -1)
      {
         if (list[temp].data = b)
            break;
         else temp = list[temp].link;
      }
      prev = temp;
      list[current].link = list[prev].link;
      list[prev].link = current;
   }
```

3. Delete a node:

```
   void Delete(struct node list[max], int first, int a)
   {
      int i, j, prev, current, temp;
      temp = first;
      while (temp! = - 1)
      {
         if (list[temp.data == a)
            break;
            temp = list.[temp].link;
      }
      current = temp;
      temp = first;
      while (temp! = -1)
      {
         if(list[temp].data == current)
            break;
            temp = list[temp].link
      }
      prev = temp;
      list[prev].link=list[current]link;
      list[current].link = - 1;
      list[current].data = 0;
   }
```

4. **Displaying the list:**

```
   void Display (struct node list [max], int first)
   {
      int current;
      current = first;
      while(current ! == - 1)
      {
         printf("%d \t"; list[current].data);
         current = list[current].link;
      }
   }
```

Drawbacks of Static Representation:

1. A fixed number of nodes have to be represented at the start of the program execution.
2. To predict number of nodes prior is not always possible.
3. Memory is wastage due to static allocation.

- Hence, the node must be allocated to a program only when it is required, which is possible in dynamic memory allocation.
- Refer the Program 3.3 for creation and display of static linked list.

3.2.2 Dynamic Representation of Linked List

- For the dynamic implementation of lists, we can link all data elements/items using pointers.
- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. Each data element/node holds a pointer (link) to next node in the list.
- A linked list maintains data elements in a logical order rather than in physical order.
- **Example:** This is an example of singly linked list.

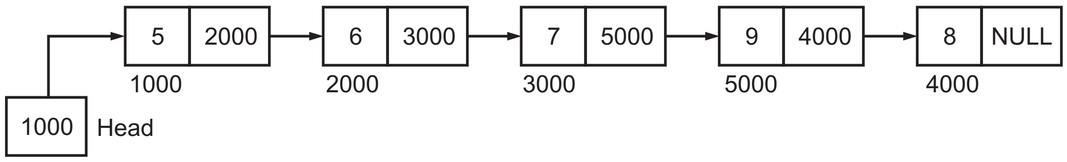


Fig. 3.11: Representation of linked list using dynamic memory allocation

- The **node structure** for a singly linked list in 'C' is:

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

- This structure holds two members:
  1. Member 1 is an integer type of data.
  2. Member 2 is pointer next which stores the address of the next node in the list.

Program 3.1: Program for creation and display of dynamic linked list.

```
#include <stdio.h>
#include <stdlib.h>
void display();
struct node
{
   int info;
   struct node *link;
}*start=NULL;
int main(void)
{
   setbuf(stdout, NULL);
   int data;
   char ch;
   struct node *q, *tmp;
```

```
    do
    {
       printf("Enter element :\n");
       scanf("%d",&data);
       tmp=malloc(sizeof(struct node));
       tmp->info=data;
       tmp->link=NULL;
       if(start==NULL)    //insertion of first node
          start=tmp;
       else               //insertion of subsequent nodes
       {
          q=start;
          while(q->link!=NULL)
                q=q->link;
          q->link=tmp;
       }
       printf("Do you want to insert more elements?");
       scanf(" %c",&ch);  //use a space before %c to clear from stdin
    }while(ch=='y'||ch=='Y');
    display();
    return 0;
 }
 void display()
 {
    struct node *q;
    if(start==NULL)
       printf("List is empty!!\n");
    else
    {
       printf("**** Elements in Linked List ****\n");
       q=start;
       while(q!=NULL)
       {
          printf("%d\t",q->info);
          q=q->link;
       }
    }
 }
```

Output:

```
Enter element:
10
Do you want to insert more elements?y
Enter element:
20
Do you want to insert more elements?n
**** Elements in Linked List ****
10   20
```

3.2.3 Internal and External Pointers

- A linked list must always have a head pointer. Depending on how we use the list, we may have several other pointers as well.
- For example, if we are going to search a linked list, we will need an additional pointer to the location where we found the data we were looking for.
- T_n the linked list the ordering of the elements is not done by their physical location in the memory but by logical links, which is stored in the link field called pointer.
    1. **Internal Pointer:** Those which form a linked list and pointer to the next node in the list.
    2. **External Pointer:** It is a pointer to the starting node. The external pointer contains the base i.e. address of the first node. Once a base address is available, its next successive nodes can be accessed.
- When there is no node in the list, it is called as empty list. If the external pointer were assigned a value NULL, the list would be empty. NULL pointer is external pointer not pointing to any element.

3.3 TYPES OF LINKED LIST (SINGLY, DOUBLY AND CIRCULAR)

- A list is an ordered list, which consists of different data items connected by means of a link or pointer this type of list is also called a linked list.
- A linked list is defined as, "an ordered collection of finite homogeneous data elements called as nodes where the linear order is maintained by means of pointers or links".
- Types of linked list are given below:

1. **Singly Linked List:** **[Oct. 17]**

- The way to represent a linear list is to expand each node to contain a link or pointer to the next node. This representation is called a one-way chain or singly linked list.
- In this type of linked list two nodes are linked with each other in sequential linear manner. In the singly link list each node has two fields one for data and other is for link reference of the next node. The last node's link field points to NULL.

- Fig. 3.12 shows singly linked list.

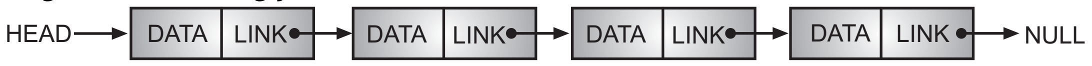


Fig. 3.12

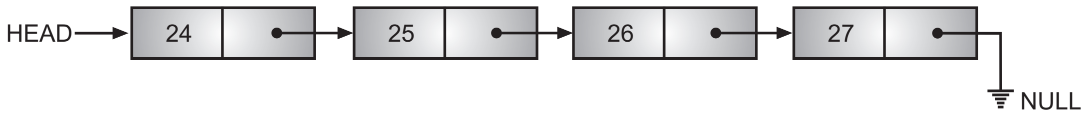


Fig. 3.13: Example of Singly Linked List

Advantages of Singly Linked List:

(i) Accessibility of a node in the forward direction is easier.

(ii) Insertion and deletion of nodes are easier.

Disadvantages of singly linked list:

(i) Accessing the preceding node of a current node is not possible as there is no backward traversal.

(ii) Accessing a node is time consuming.

2. Doubly Linked List:

- It is also called as two-way linked list. A linked list which can be traversed both in backward as well as forward direction is called doubly linked list.
- In this type of liked list each node holds two-pointer field. Pointers exist between adjacent nodes in both directions.
- In the doubly link list each node has three field two fields for link which is the reference to next and previous and one for data record. The node has null only to the first node of previous and last node at the next. The list can be traversed either forward or backward.
- Fig. 3.14 shows doubly linked list.

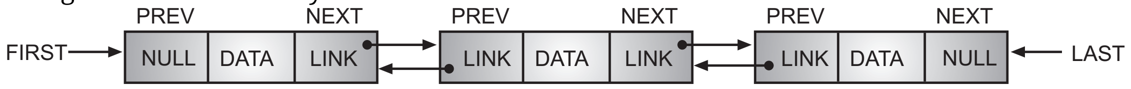


Fig. 3.14

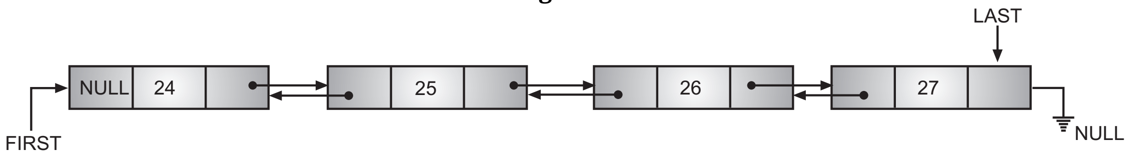


Fig. 3.15: Example of Doubly Linked List

Advantages of Doubly Linked List:

(i) Doubly linked list with header node, most of the problems can be solved very easily and effectively.

(ii) Insertion and deletion are simple as compared to other lists.

(iii) Efficient utilization of memory, no wastage of memory as in Sequential representation.

(iv) Bidirectional (both forward and backward) traversal helps in efficient and very easily modes can be accessible.

(v) Multiple stacks can be represented efficiently i.e., PUSH and POP operations are easier and efficient.

(vi) Doubly linked list are extensively used in trees because hierarchical structure of the tree can be easily represented.

Disadvantages of Doubly Linked List:

(i) A node in a linked requires more memory than a corresponding element in an array representation.

(ii) Each node requires two pointers (links) one is forward link and the other backward link requires additional storage for each field.

3. Circular Linked List: **[April 17]**

- A linked list in which the pointer of the last node points to the first node of the list is called circular linked list.
- In circular linked list the first and last node are adjacent. Fig. 3.16 shows circular linked list.

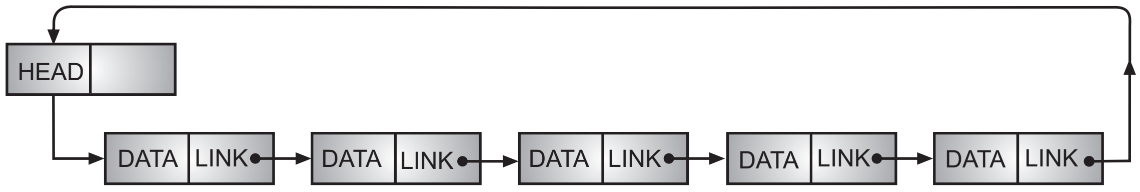


Fig. 3.16

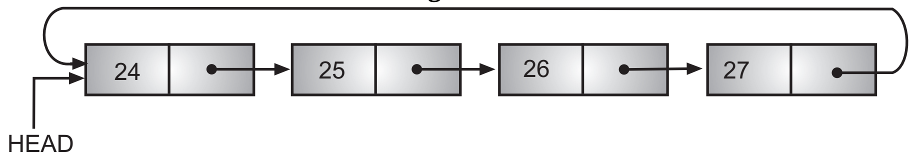


Fig. 3.17: Example of Circular Linked List

- Both singly and doubly linked lists can be made circular. In singly circular link list (SCL) (See Fig. 3.18) each node have two fields one for data record and other for link reference to the next node. The last node has link reference to the first node. There in no null present in the link of any node.
- A doubly circular linked list is a doubly linked list with first node linked to last node and vice-versa, (See Fig. 3.19).

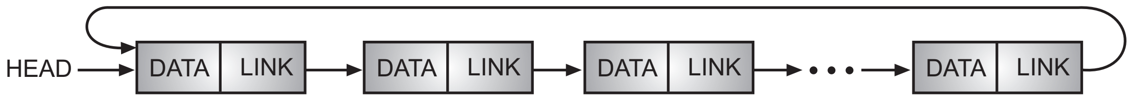


Fig. 3.18: Singly Circular Linked List

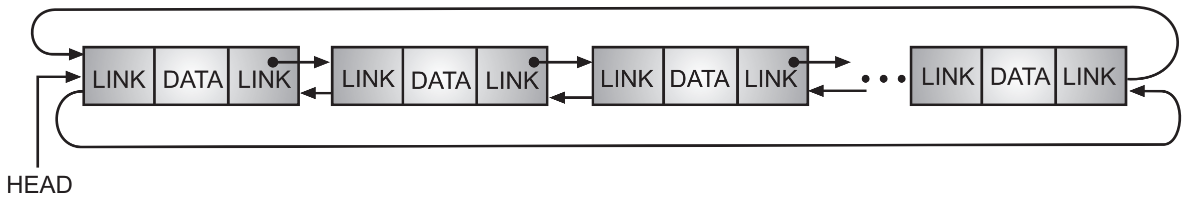


Fig. 3.19: Doubly Circular Linked List

Advantages of Circular Linked List:

(i) If we are at a node, then we can go to any node. But in linear linked list it is not possible to go to previous node.

(ii) It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes. But in doubly linked list, we will have to go through in between nodes.

Disadvantages of Circular Linked List:

(i) It is not easy to reverse the linked list.

(ii) If proper care is not taken, then the problem of infinite loop can occur.

(iii) If we at a node and go back to the previous node, then we cannot do it in single step. Instead we have to complete the entire circle by going through the in between nodes and then we will reach the required node.

3.4 OPERATIONS ON LINKED LIST

- In this section we will study various operations on linked list.

3.4.1 Operations on Singly Linked List

- A linked list in which every node has one link field to provide information about where the next node of list is called as singly linked list.
- In singly linked list, we can traverse only in one direction. Practically, singly linked list is referred as just linked list.
- The list which we have discussed till now is nothing but singly linked list. So the operations which are discussed above are same.
- The **node structure** is:

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

- Fig. 3.20 shows the logical representation of a linked list.

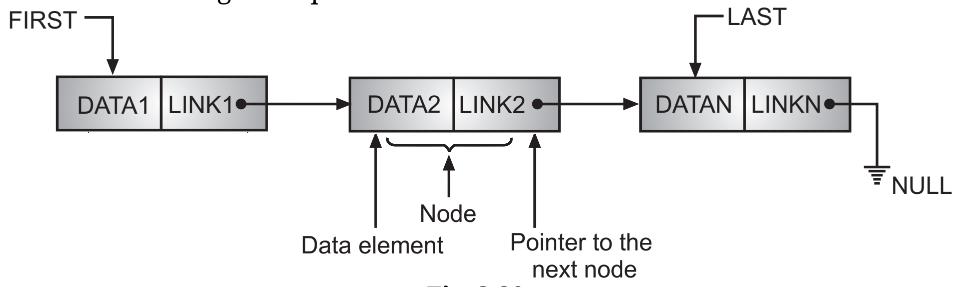


Fig. 3.20

- Fig. 3.21 shows the singly linked list.

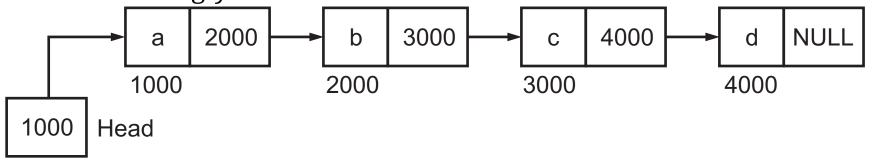


Fig. 3.21: Singly Linked List

3.4.1.1 Create a Linked List

- To create a linked list, we have to create a node one by one and append (insert at end) it to the list.
```
    initialize head = NULL;
```
- It is important to note that head is not a node, rather the pointer variable which store address of the first node of the list.

1. **Create a New Node:** To create a node following steps are required:
   - Allocate memory for the new node.
   - Store data in the new node's data field.
   - Set next field to NULL.
2. **Append:** Add newly created node at the end of linked list.

   Case 1:
```
   /* if empty list */
   head = NULL
   set head = newnode
```
   Case 2:
```
   /* if list is not empty */
   head != NULL
   set last->next = newnode
   last = newnode
```
- To illustrate these steps, consider the following example:

Example: Let us consider {Sharvari, Ritika, Avani} strings as data for linked list creation.

/* Assume 1000, 2000 and 3000 are addresses returned by malloc() function * /

Step 1 : `Initialize head = NULL;`

Step 2 : `Create a new node:`
`Consider node data as 'Sharvari'.`
```
o newnode=(struct node *)malloc (sizeof(struct node));
o strcpy(newnode->data,"Sharvari");
o newnode->next=NULL;
```

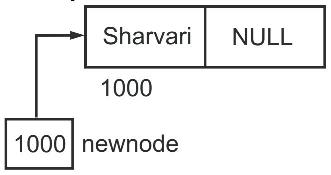


Fig. 3.22 (a)

Step 3 : Append this node in the linked list.
As `head = NULL`, case 1 is true.
`head=last=newnode;`

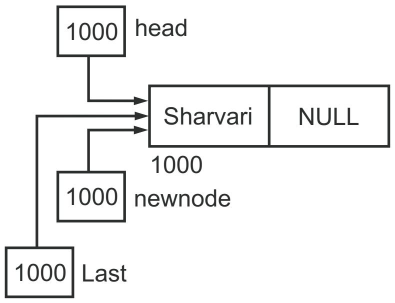


Fig. 3.22 (b)

Step 4 : Create a new node:

Consider node data as 'Ritika'.

- newnode=(struct node *)malloc (sizeof(struct node));
- strcpy(newnode->data,"Ritika");
- newnode->next=NULL;

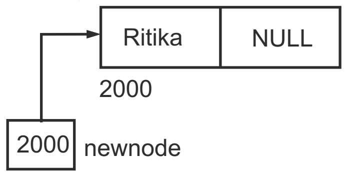


Fig. 3.22 (c)

Step 5 : Append this node in the linked list

As head -> NULL, so case 2 is true.

- last->next=newnode;
- last=newnode;

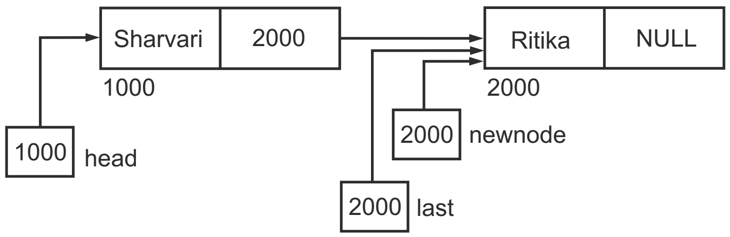


Fig. 3.22 (d)

Similarly insert node with data 'Avani' using above steps. You will get:

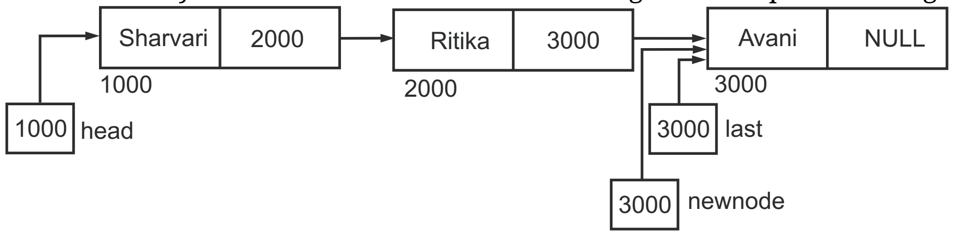


Fig. 3.22 (e)

Fig. 3.22 (a) (b) (c) (d) (e): Example of Create and Append Linked List

- In similar way, we can insert 'n' nodes into the linked list.
- **The 'C' function to append node into singly linked list is written as:**

```
struct node * append(struct node *head)
{
    struct node * newnode=NULL,*temp=head;
    newnode=(struct node *)malloc (sizeof(struct node));
    newnode->next=NULL;
    scanf("%d", &newnode->data);
    if(head==NULL)
        head = last = newnode;
    else
    {
        last ->next = newnode;
        last = newnode;
    }
   return head;
}
```

Note:

- The nodes do not actually store in sequential locations in memory.
- The address of node may change as we run the program number of times.
- The above task can be performed using dynamic memory management functions in C, such as malloc() and free().

3.4.1.2 Insertion of a Node

- There are various positions where a node can be inserted:
  1. Insert at the beginning of the list (first position)
  2. Insert at the end (last position)
  3. Insert at any position (middle of the list).

1. To Insert Node at First Position:

- The dotted line represents link manipulation needed to add new node at the first position.

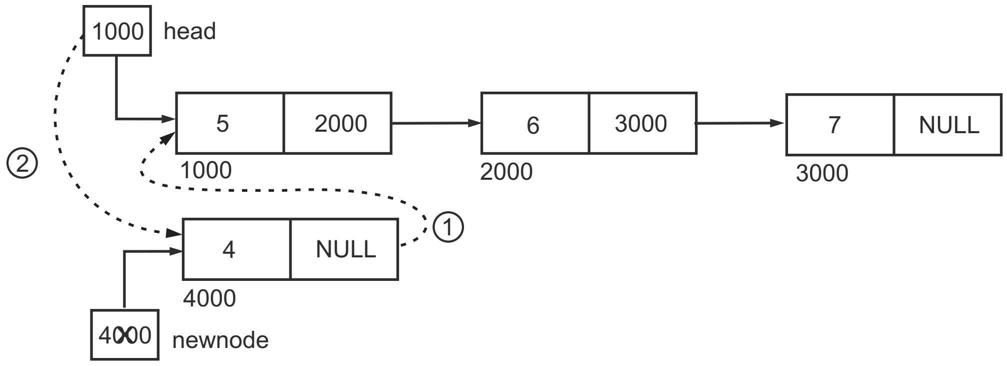


Fig. 3.23: (a) Before Insertion at the Beginning

- Using following steps, you get linked list as shown in Fig. 3.23 (b).
  - `newnode →next = head;`
  - `head = newnode`

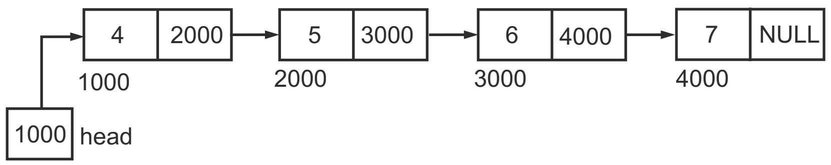


Fig. 3.23: (b) After Insertion at the Beginning

- **The 'C' function to insert node (at front/first) into singly linked list is written as:**

```
struct node * SLL_insert_front(struct node *head)
{
   struct node * newnode =NULL;
   newnode=(struct node *)malloc (sizeof(struct node));
   newnode->next=NULL;
   scanf("%d", &newnode->data);
   if(head==NULL)
      head=newnode;
   else
   {
      newnode->next = head;
      head=newnode;
   }
   return head;
}
```

2. **Inserting Node at End:**

- The dotted line represents link manipulation needed to add node at the end.

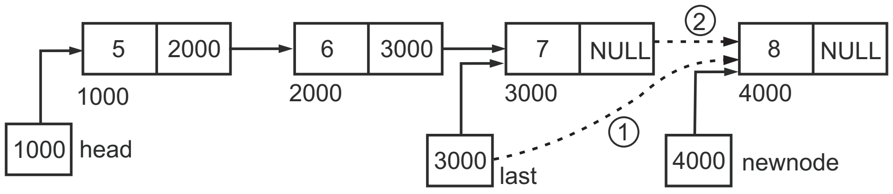


Fig. 3.24: (a) Before Insertion at End

- Using following steps, you get linked list as shown in Fig. 3.24 (b).
  - `last->next=newnode;`
  - `last = newnode;`

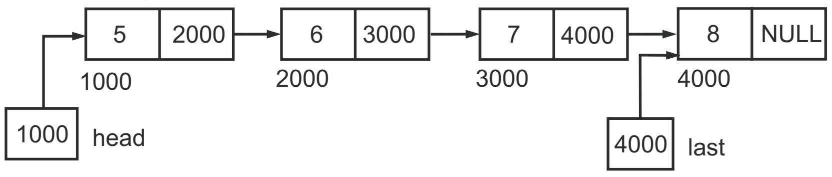


Fig. 3.24: (b) After Insertion at End

- **The 'C' function to insert node (at end) into singly linked list is written as:**

```
struct node * SLL_insert_end(struct node *head)
{
   struct node * newnode =NULL, *temp= NULL;
      newnode=(struct node *)malloc (sizeof(struct node));
      newnode->next=NULL;
      scanf("%d", &newnode->data);
```

```
        if(head==NULL)
           head = last = newnode;
        else
        {
           last->next  = newnode;
           last= newnode;
        }
     return head;
  }
```

3. Insertion of a Node at given position:

- Consider 4 nodes in the linked list and insert a newnode after third node in the linked list.

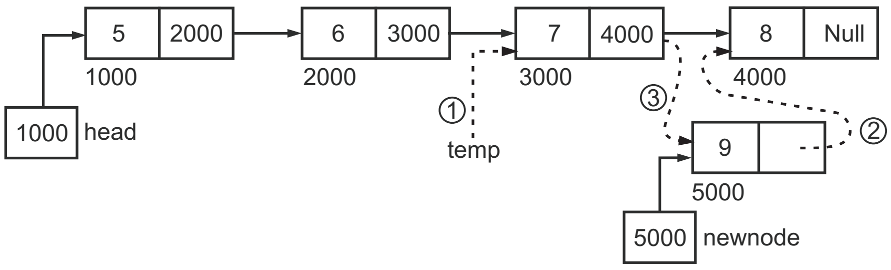


Fig. 3.25: (a) Before Insertion at Fourth Position

- The newnode is to be inserted after temp. The node which is successor of temp will now become successor of newnode.
- Using following steps, you get linked list as shown in Fig. 5.25 (b).
  - `Traverse temp pointer from head to till position-1.`
  - `newnode -> next = temp -> next;`
  - `temp->next=newnode;`

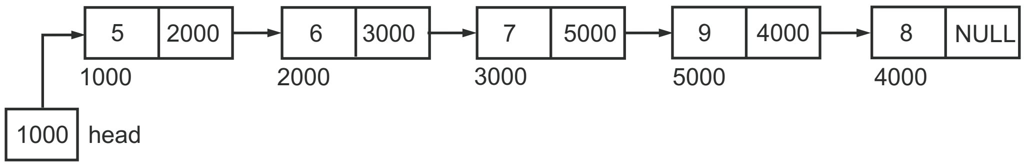


Fig. 3.25: (b) After Insertion at any Specified Node

- **The 'C' function to insert node (at middle) into singly linked list is written as:**

```
    struct node * SLL_insert_middle(struct node *head, int pos)
    {
       int i;
       struct node * newnode =NULL, *temp= head;
       newnode=(struct node *)malloc (sizeof(struct node));
       newnode->next=NULL;
       scanf("%d", &newnode->data);
```

```
        if (pos == 1)
        {
           newnode ->next = head;
           head = newnode;
        }
        else
        {
           for ( i = 1; i< pos-1; i++)
               temp = temp -> next;
               newnode ->next  = temp -> next;
               temp -> next = newnode;
        }
        return head;
    }
```

3.4.1.3 Deleting a Node from Linked List

- Deletion can also have various cases like insertion:
  1. Deletion at the front of the list (first position).
  2. Deletion at the end of the list (last position).
  3. Deletion at any position in the list (middle of list).
- Consider the following linked list:

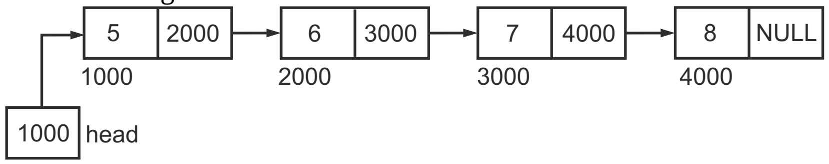


Fig. 3.26 (a): Input Linked List

1. Deletion from Beginning:

- If the node at first position is to be deleted, then we need to modify the head pointer.

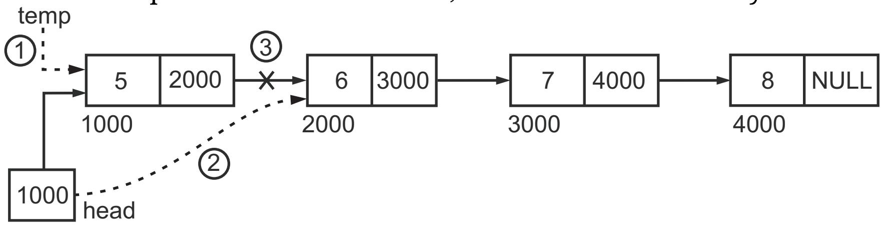


Fig. 3.26: (b) Before Deletion from front in SLL

- Using following steps, you will get linked list as shown in Fig. 3.26 (c).
  - `temp = head;`
  - `head = head -> next`
  - `free (temp);`

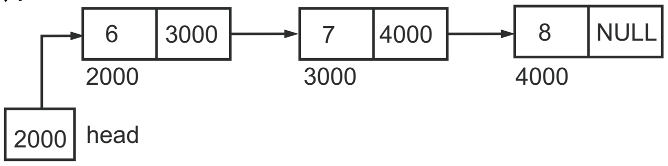


Fig. 3.26: (c) After deletion from front in SLL

- **The 'C' function to delete node (from front/beginning) from singly linked list is written as:**

```
struct node * SLL_delete_front(struct node *head)
{
   struct node * temp = NULL;
      printf(" Deleted element is : %d\n", head-> data);
      if(head -> next == NULL)
      {
         free (head)
         head = last = NULL;
      }
      else
      {
         temp = head;
         head = head ->next;
         temp -> next = NULL;
         free (temp);
      }
   return head;
}
```

2. Deletion from End:

- Let the last node pointed by 'last' pointer need to be deleted. Take pointer 'temp' to the node which is predecessor of 'last'.

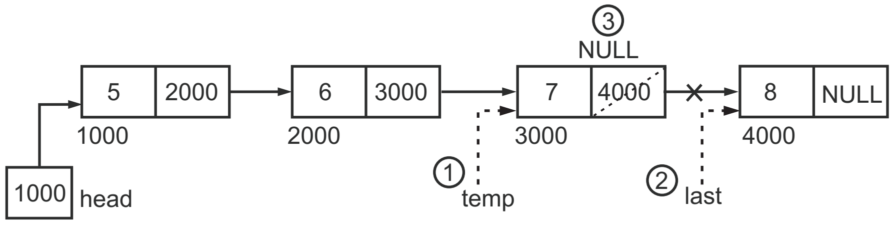


Fig. 3.27 (a): Before Deletion from End in SLL

- Using following steps, you will get linked list as shown in Fig. 3.27 (b).
  - `temp -> next = NULL;`
  - `free(last);`

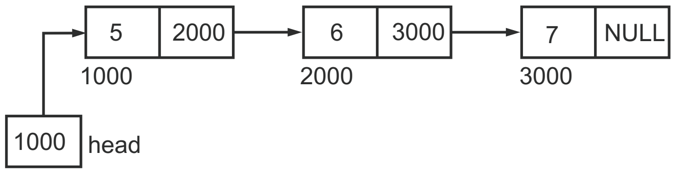


Fig. 3.27 (b): After Deletion from End in SLL

- **The ‘C’ function to delete node (from end) from singly linked list is written as:**

```
struct node * SLL_delete_end (struct node *head)
{
   struct node * temp = NULL;
      printf(" Deleted element is : %d\n ", head-> data);
      if(head -> next == NULL)
      {
         free (head)
         head = last = NULL;
      }
       else
       {
         temp = head;
          while ( temp -> next != last)
                temp = temp ->next;
         temp -> next = NULL;
         free(last);
         last = temp;
    }
   return head;
}
```

3. **Deletion from Specified Position:**

- We can use the following steps to delete a specific node from the single linked list.

```
Traverse pointer ‘q’ from head till (position-1)th node.
Set temp to q’s next;
q -> next = temp -> next;
temp->next=NULL;
free(temp);
```

- The node which is successor of temp will become successor of pointer ‘q’.

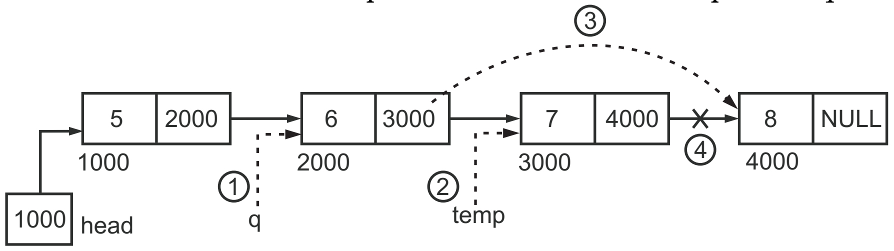


Fig. 3.28: (a) Before Deletion at specified position

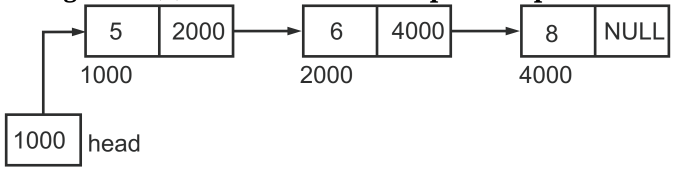


Fig. 3.28: (b) After Deletion at Specified Position

The ‘C’ function to delete node (from middle) from singly linked list is written as:

```
struct node * SLL_delete_middle(struct node *head, int pos)
{
    int i;
    struct node * q =NULL, *temp= head;
        for ( i = 1; i< pos-1; i++)
            temp = temp -> next;
        q = temp -> next;
        temp -> next = q -> next;
        q-> next = NULL;
            printf(" Deleted element is : %d\n", q-> data);
        free (q);
    return head;
}
```

Program 3.2: Menu driven program for single linked list (create, insert, delete, display).

```
#include <stdio.h>
#include <malloc.h>
#include <process.h>
void create();
void insert_at_beg();
void insert_at_end();
void insert_after_pos();
void del();
void search();
void display();
struct node
{
    int info;
    struct node *link;
}*start=NULL;
int data,item,n1,pos,i,m;
int main()
{
    int n;
    setbuf(stdout, NULL);
    printf("\n****Linked List*****\n");
    printf("\n1.Create\n2.Insert  at  Beginning\n3.Insert  at  End\n4.Insert
    After Position\n5.Delete\n6.Search\n7.Display\n8.Exit\n");
    while(1)
    {
        printf("\nEnter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at
        End 4.Insert after Pos. 5.Delete 6.Search  7.Display 8.Exit)\n");
        scanf("%d",&n);
        switch(n)
        {
        case 1:
            create();
            break;
        case 2:
            insert_at_beg();
            break;
        case 3:
            insert_at_end();
            break;
        case 4:
            insert_after_pos();
            break;
        case 5:
            del();
            break;
        case 6:
            search();
            break;
        case 7:
            display();
            break;
        case 8:
            exit(0);
        default:
            printf("\nWrong Choice !!\n");
        }
    }
    return 0;
}
void create()
{
    struct node *q, *tmp;
    printf("Enter element :\n");
    scanf("%d",&data);
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=NULL;
    if(start==NULL)
       start=tmp;
    else
    {       q=start;
          while(q->link!=NULL)
                q=q->link;
          q->link=tmp;
    }
}
void insert_at_beg()
{
    struct node *tmp;
    printf("\nEnter the element to be inserted :\n");
    scanf("%d",&data);
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=start;
    start=tmp;
    display();
}
void insert_at_end()
{
    struct node *tmp,*q;
    printf("\nEnter the element to be inserted :\n");
    scanf("%d",&data);
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=NULL;
    if(start==NULL)
       start=tmp;
    else
    {
       q=start;
       while(q->link!=NULL)
          q=q->link;
       q->link=tmp;
    }
    display();
}
```

```
void insert_after_pos()
{
   display();
   struct node *q,*tmp;
   int index;
   tmp=malloc(sizeof(struct node));
   printf("\nEnter the element to be inserted :\n");
   scanf("%d",&data);
   tmp->info=data;
   tmp->link=NULL;

   if(start==NULL)
   {
      start=tmp;
   }
   else
   {
      printf("Enter index after which element to be inserted :\n");
      scanf("%d",&index);
      q=start;
      for(i=0;i<index;i++)
      {
         q = q->link;
         if(q==NULL)
         {
               printf("There are  less  elements\n");
               return;
         }
      }
      tmp->link = q->link;
      q->link = tmp;
   }
   display();
}
void del()
{
   struct node *q,*tmp;
   printf("Enter the element to be deleted :\n");
   scanf("%d",&data);
```

```
    if(start->info==data)  //deletion of first node
    {
       tmp=start;
       start=start->link;
       free(tmp);
       display();
       return;
    }
    q=start;
    while(q->link->link!=NULL)     //deletion middle node
    {
       if(q->link->info==data)
       {
          tmp=q->link;
          q->link=tmp->link;
          free(tmp);
          display();
          return;
       }
       q=q->link;
    }
    if(q->link->info==data) //deletion of last node
    {
       tmp=q->link;
       q->link=NULL;
       free(tmp);
       display();
       return;
    }
    printf("\nElement not found \n");
 }
 void search()
 {
    struct node *tmp;
    int i=0;
    printf("\nEnter the element to be searched :");
    scanf("%d",&item);
    tmp=start;
    while(tmp!=NULL)
    {
        if(tmp->info==item)
        {
            printf("Element found at index: %d\n",i);
            return;
        }
        tmp=tmp->link;
        i++;
    }
    if(tmp->link==NULL)
        printf("Element not found \n");
}
void display()
{
    struct node *q;
    if(start==NULL)
        printf("List is empty!!\n");
    else
    {
        printf("**** Elements in Linked List ****\n");
        q=start;
        while(q!=NULL)
        {
            printf("%d\t",q->info);
            q=q->link;
        }
    }
}
```

Output:

```
****Linked List*****

1.Create
2.Insert at Beginning
3.Insert at End
4.Insert After Position
5.Delete
6.Search
7.Display
8.Exit
```

```
Enter  Your  Choice  :(1.Create  2.Insert  at  Beg.  3.Insert  at  End  4.Insert
after Pos. 5.Delete 6.Search  7.Display 8.Exit)
1
Enter element:
10
Enter  Your  Choice  :(1.Create  2.Insert  at  Beg.  3.Insert  at  End  4.Insert
after Pos. 5.Delete 6.Search  7.Display 8.Exit)
2
Enter the element to be inserted :
5
**** Elements in Linked List ****
5  10
Enter  Your  Choice  :(1.Create  2.Insert  at  Beg.  3.Insert  at  End  4.Insert
after Pos. 5.Delete 6.Search  7.Display 8.Exit)
3
Enter the element to be inserted :
15
**** Elements in Linked List ****
5  10 15
Enter  Your  Choice  :(1.Create  2.Insert  at  Beg.  3.Insert  at  End  4.Insert
after Pos. 5.Delete 6.Search  7.Display 8.Exit)
4
**** Elements in Linked List ****
5  10 15
Enter the element to be inserted :
20
Enter index after which element to be inserted :
2
**** Elements in Linked List ****
5  10 15 20
Enter  Your  Choice  :(1.Create  2.Insert  at  Beg.  3.Insert  at  End  4.Insert
after Pos. 5.Delete 6.Search  7.Display 8.Exit)
5
Enter the element to be deleted :
20
**** Elements in Linked List ****
5  10 15
Enter  Your  Choice  :(1.Create  2.Insert  at  Beg.  3.Insert  at  End  4.Insert
after Pos. 5.Delete 6.Search  7.Display 8.Exit)
6
Enter the element to be searched :15
Element found at index: 2
Enter  Your  Choice  :(1.Create  2.Insert  at  Beg.  3.Insert  at  End  4.Insert
after Pos. 5.Delete 6.Search  7.Display 8.Exit)
8
```

3.4.1.4 Traversing/Display a Linked List

- Traversing means all the elements in the list are visiting sequentially one by one. To traverse a linked list, start from the first node, using next field go to second node and so on.
- Traversing is the procedure of passing through (visiting) all the nodes of the linked list from the starting to end.

Algorithm:

1. Get the address of first node, say 'temp'.
2. If 'temp' is not null, repeat step (iii) and step (iv).
3. Process the data field of 'temp'. Process includes display, update etc.
4. Move to next node using 'next' field of 'temp'.
5. Stop.

- **The 'C' function to traversal a list in 'C' is written as:**

```
void display(struct node *head)
{
    struct node *temp=head;
    printf("\nLinked List contents are : \n");
    while(temp != NULL)
    {
       printf("%d\t",temp->data);
       temp=temp->next;
    }
}
```

Note:

- A linked list must always have at least one pointer pointing to first node of the list, which is called head pointer.

  For example: Insert a node with DATA (30) at the end, then traverse the list and the output will 20 → 40 → 30 which means we traversing the nodes from left to right (See Fig. 3.29).

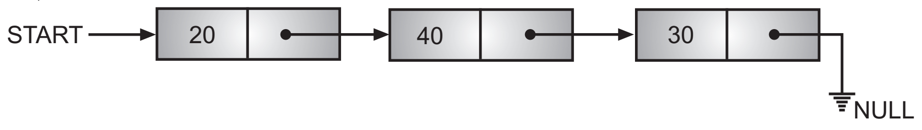


Fig. 3.29: Traversing a Singly Linked List

Program 3.3: Program for creation and display of static linked list.

```
#include <stdio.h>
void display();
struct node
{
   int info;
   struct node *link;
};
```

```
int main()
{
    struct node n4={40,NULL};
    struct node n3={30,&n4};
    struct node n2={20,&n3};
    struct node n1={10,&n2};
    struct node* start=&n1;
    setbuf(stdout, NULL);
    printf("** Elements of linked list **\n");
    struct node *q=start;
    while(q!=NULL)
    {
        printf("%d\t",q->info);
        q=q->link;
    }
}
```

Output:

```
** Elements of linked list **
    10  20 30  40
```

3.4.1.5 Reverse a Linked List [April 17]

- Reversing of the list means that last node becomes the first node and first node becomes the last.
- Example:

  Input: Head of following linked list,
  1->2->3->4->NULL
  Output: Linked list should be changed to,
  4->3->2->1->NULL

1. **The C function to reverse singly linked list is written as:**

```
void reverse (struct node *head)
{
    struct node *p=NULL, *q=NULL, *r=NULL;
    p = head;
    while (p != null)
    {
        r = q;
        q = p;
        p = p -> next;      /* pointing to successor */
        q -> next = r;      /* reverse */
    }
    head = q;
}
```

2. **The C function to print the list in reverse order is written as:**

```
void print_rev (struct node *head)
{
    if(head!=NULL)
    print_rev(head->link);
    printf("%d\t", head->data);
}
```

Program 3.4: Program to reverse a linked list.

```
#include <stdio.h>
#include <stdlib.h>
/* Link list node */
struct Node {
    int data;
    struct Node* next;
};
/* Function to reverse the linked list */
static void reverse(struct Node** head_ref)
{
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next = NULL;
    while (current != NULL) {
        // Store next
        next = current->next;

        // Reverse current node's pointer
        current->next = prev;

        // Move pointers one position ahead.
        prev = current;
        current = next;
    }
    *head_ref = prev;
}
/* Function to push a node */
void push(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}
```

```
/* Function to print linked list */
void printList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d  ", temp->data);
        temp = temp->next;
    }
}
/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 85);

    printf("Given linked list\n");
    printList(head);
    reverse(&head);
    printf("\nReversed Linked list \n");
    printList(head);
    getchar();
}
```

Output:

```
Given linked list
85 15 4 20
Reversed Linked list
20 4 15 85
```

- Time Complexity: O(n).
- Space Complexity: O(1).

3.4.1.6 Concatenation of Linked List

- Concatenation is the process of appending the second list to the end of the first list consisting of m nodes.
- When we concatenate two lists, the second list has n nodes, then the concatenated list will be having (m + n) nodes.

- C **function to two concatenate linked lists is written as:**

```
struct node *concatenate (struct node *X, struct node *Y)
{
    while(X->link!=NULL)
    {
        X=X->link;
    }
    X->link=Y;
    Return(X);
}
```

Here the X and Y are concatenated and X is pointer to first node of resultant list.

Program 3.5: Program for concatenation of list.

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *even = NULL;
struct node *odd = NULL;
struct node *list = NULL;
//Create Linked List
void insert(int data) {
   // Allocate memory for new node;
   struct node *link = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   link->data = data;
   link->next = NULL;
   if(data%2 == 0) {
      if(even == NULL) {
         even = link;
         return;
      } else {
         current = even;
         while(current->next != NULL)
         current = current->next;
         // Insert link at the end of the list
         current->next = link;
      }
   }
```

```
else {
      if(odd == NULL) {
         odd = link;
         return;
      } else {
         current = odd;
         while(current->next!=NULL)
            current = current->next;
         // Insert link at the end of the list
         current->next = link;
      }
   }
}
void display(struct node *head) {
   struct node *ptr = head;
   printf("[head] =>");
   while(ptr != NULL) {
      printf(" %d =>",ptr->data);
      ptr = ptr->next;
   }
   printf(" [null]\n");
}
void combine() {
   struct node *link;
   list = even;
   link = list;
   while(link->next!= NULL) {
      link = link->next;
   }
   link->next = odd;
}
int main() {
   int i;
   for(i = 1; i <= 10; i++)
      insert(i);
   printf("Even : ");
   display(even);
   printf("Odd  : ");
```

```
    display(odd);
    combine();
    printf("Combined List :\n");
    display(list);
    return 0;
}
```

Output:

```
Even : [head] => 2 => 4 => 6 => 8 => 10 => [null]
Odd  : [head] => 1 => 3 => 5 => 7 => 9 => [null]
Combined List :
[head] => 2 => 4 => 6 => 8 => 10 => 1 => 3 => 5 => 7 => 9 => [null]
```

3.4.1.7 Searching in Linked List

- Searching is performed in order to find the location of a particular element in the list.
- Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list elements, then the location of the element is returned from the function.

Algorithm:

```
Step 1 : SET ptr = head
Step 2 : Set i = 0
Step 3 : if ptr = NULL
         Write "List is Empty"
         Goto Step 8
         End of if
Step 4 : Repeat STEP 5 TO 7 Until ptr != NULL
Step 5 : if ptr -> data = item
         write i+1
         End of IF
Step 6 : i = i + 1
Step 7 : ptr = ptr -> Next
Step 8 : Stop
```

C function to searching element in singly linked list is written as:

```
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
```

```
        else
        {
            printf("\nEnter item which you want to search?\n");
            scanf("%d",&item);
            while (ptr!=NULL)
            {
                if(ptr->data == item)
                {
                    printf("item found at location %d ",i+1);
                    flag=0;
                }
                else
                {
                    flag=1;
                }
                i++;
                ptr = ptr -> next;
            }
            if(flag==1)
            {
                printf("Item not found\n");
            }
        }
    }
```

Program 3.6: Program to search an element in Linked list.

```
#include <stdio.h>
void display();
struct node
{
   int info;
   struct node *link;
};
struct node n4={40,NULL};
struct node n3={30,&n4};
struct node n2={20,&n3};
struct node n1={10,&n2};
struct node* start=&n1;
int main()
{
   int data;
   int flag=0;
   struct node *q;
   setbuf(stdout, NULL);
```

```
    printf("Linked List before deletion of specific node\n");
    display();
    printf("\nEnter the element to be searched :\n");
    scanf("%d",&data);
    q=start;
    while(q!=NULL)
    {
       if(q->info==data)
       {
                flag=1;
          printf("Element found");
          break;
       }
       q=q->link;
    }
    if(flag==0)
       printf("Element not found");
}
void display()
{
    struct node *q=start;
    while(q!=NULL)
    {
       printf("%d\t",q->info);
       q=q->link;
    }
}
```

Output:

```
Linked List before deletion of specific node
10 20 30 40
Enter the element to be searched :
30
Element found
```

3.4.1.8 Sort the Elements of Linked List

- We want to sort linked list in the ascending order.
- Function sortList() will sort the nodes of the list in ascending order.

Algorithm:

1. Define a node temp which will point to head.
2. Define another node next which will point to node next to temp.
3. Compare data of temp(temp->data) and index node(temp->next->data). If temp's data is greater than the index node's data then, swap the data between them.
4. temp will point to temp.next that is moving to the next element and node will point to node.next.
5. Continue this process until the entire list is sorted.

C function to sort the Linked list as follows:

```
void SortList()
{
   struct LinkedNode *node=NULL, *temp = NULL;
      int tempvar;     //temp variable to store node data
      node = start;   // head node is initialize to start
      while(node != NULL)
      {
         temp=node;    //temp node to hold node data and next link
         while (temp->next !=NULL)
                      //travel till the second last element
         {
            if(temp->data > temp->next->data)
                      // compare the data of the nodes
            {
            tempvar = temp->data;
               temp->data = temp->next->data;  // swap the data
            temp->next->data = tempvar;
            }
            temp = temp->next;    // move to the next element
         }
            node = node->next;    // move to the next node
      }
}
```

Program 3.7: Program for sorting a linked list in ascending order.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
   int data;
   struct node *next;
};
void main()
{
   int i;
   int num ;
   struct node *first, *nw, *pre, *new1, *count;
   clrscr();
   printf("\n Enter the number of node you want to create: ");
   scanf("%d", &num );
   first->next = NULL;
   nw = first;
```

```
    for (i = 0; i < num ; i++)
    {
       nw->next = (struct node* ) malloc(sizeof(struct node));
       nw = nw->next;
       printf("\n Enter the node: %d: ", i+1);
       scanf("%d", &nw->data);
       nw->next = NULL;
    }
    new1 = first;
    for( ; new1->next != NULL; new1 = new1->next)
    {
       for(count = new1->next; count != NULL; count = count->next)
       {
          if(new1->data > count->data)
          {
             int temp = new1->data;
             new1->data = count->data;
             count->data = temp;
          }
       }
    }
    nw = first->next;
    printf("\n After sorting the Linked list is as follows:\n");
    while (nw)
    {
       printf("%d\t", nw->data);
       nw = nw->next;
    }
    getch();
 }
```

Output:

```
Enter the number of the node you want to create:?
Enter the node: 1: 34
Enter the node: 2: 89
Enter the node: 3: 0
Enter the node: 4: 45
Enter the node: 5: 11
Enter the node: 6: 2
Enter the node: 7: 123
After sorting the linked list is as follows:
0     2     11     34     45     89     123
```

3.4.1.9 Merge the Elements of Linked List

- Merging is a process in which two or more lists are merged (combined) to form a new list.

The C function to Merge two sorted linked list using recursion as follows:

```
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
   struct Node* result = NULL;
   /* Base cases */
   if (a == NULL)
         return(b);
   else if (b==NULL)
         return(a);
   /* Pick either a or b, and recur */
   if (a->data <= b->data)
   {
      result = a;
      result->next = SortedMerge(a->next, b);
   }
   else
   {
      result = b;
      result->next = SortedMerge(a, b->next);
   }
   return(result);
}
```

Program 3.8: Program for merging linked list.

```
#include<stdio.h>
#include<stdlib.h>
struct node {
int data;
struct node *next;
};
void insert_node(struct node **,int);
void print(struct node *);
void merge(struct node **,struct node **,struct node **);
int main()
{
    struct node *ptr1,*ptr2,*ptr3=NULL;
    ptr1=ptr2=NULL;
    insert_node(&ptr1,23);
```

```
        insert_node(&ptr1,3);
        insert_node(&ptr1,78);
        insert_node(&ptr1,51);
        insert_node(&ptr1,90);
        print(ptr1);
        printf("\n");
        insert_node(&ptr2,15);
        insert_node(&ptr2,30);
        insert_node(&ptr2,88);
        print(ptr2);
        printf("\n");
        merge(&ptr1,&ptr2,&ptr3);
        print(ptr3);
        return 0;
    }
    void insert_node(struct node **head,int no)
    {
        struct node *p,*r;
        if(*head==NULL)
        {
            p=(struct node *)malloc(sizeof(struct node));
            p->data=no;
            p->next=NULL;
            *head=p;
        }
        else
        {
            p=*head;
            while(p->next!=NULL)
                p=p->next;
            p->next=(struct node *)malloc(sizeof(struct node));
            p=p->next;
            p->data=no;
            p->next=NULL;
        }
    }
    void print(struct node *head)
    {
        struct node *p;
        p=head;
        while(p!=NULL)
        {
            printf("%d ",p->data);
            p=p->next;
        }
        printf("\n");
    }
    void merge(struct node **ptr1,struct node **ptr2,struct node **ptr3)
    {
        struct node *temp;
        if(*ptr2==NULL && ptr1!=NULL)   // if second list dosent exist
        {
            *ptr3=*ptr1;
            return;
        }
        else if(*ptr1==NULL && ptr2!=NULL)    // if first list dosent exist
        {
            *ptr3=*ptr2;
            return ;
        }
        else if(*ptr1==NULL && ptr2==NULL)
        {
            return;
        }
        *ptr3=*ptr1;
        temp=*ptr1;
        while(temp->next!=NULL)
            temp=temp->next;
            temp->next=*ptr2;
    }
```

Output:

```
23 3 78 51 90
15 30 88
23 3 78 51 90 15 30 88
```

3.4.2 Operations on Doubly Linked List

- In singly linked list each node contains only one pointer which points to the address of next node in the list, so we can transverse a linked list only in one direction starting from first node.

- It is not possible to delete or insert ith node given only pointer to ith node.
- To overcome this problem, doubly linked list is used, in which each node contains two links, one to its predecessor and other to its successor, so traversal can take place at both the direction.
- Doubly linked list is also called two-way list. Extra storage is required to store both pointers.

Definition: [Oct. 16; April 18]

- A doubly linked list is a linear collection of data elements called nodes, where each node is having three field:
  1. A pointer field which contains an address of predecessor node in the list.
  2. The data field which contains actual data like integer, float, char string value etc.
  3. A pointer field which contains the address of successor node in the list.
- A structure of node in doubly link list is shown in Fig. 3.30.

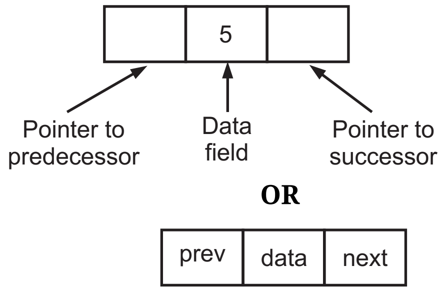


Fig. 3.30: A Node Structure

- The **node structure** for doubly link list (DLL):

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

- Fig. 3.31 shows example of doubly link list.

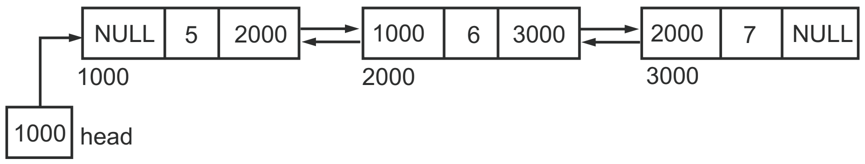


Fig. 3.31: Doubly linked list

3.4.2.1 Create a Doubly Linked List

- To create a linked list, we have to create a node one by one and append (insert at end) it to the list.

```
    initialize head = NULL;
```

Steps:

1. **Create a new node of Doubly linked list:** To create a node following steps are required:
   - Allocate memory for the new node.
   - Store data in the new node's data field.
   - Set prev and next field to NULL.
2. **Append:** Add newly created node at the end of doubly linked list.

Case 1:
```
/* if empty list */
   head = NULL
   head = newnode
   last = newnode
```
Case 2:
```
/* if doubly linked list is not empty */
   head != NULL
   set last node's next field to newnode
   set newnode's prev field to last
   last = newnode
```

Example: Let us consider { 5, 6, 7 } as data for linked list creation.

/* Assume 1000, 2000 and 3000 are addresses returned by malloc() function * /

Step 1 : `initialize head = NULL;`

Step 2 : **Creating node of Doubly Linked List (DLL):** Initially `head = NULL` i.e. list is empty.

```
newnode = (struct node *)malloc (sizeof (struct node));
newnode prev=new next = NULL;
newnode->data=5;
```

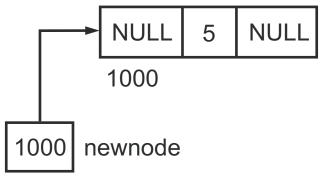


Fig. 3.32 (a)

Step 3 : **Append this node in the linked list.**

As head=NULL, case 1 is true.

`head=last=newnode;`

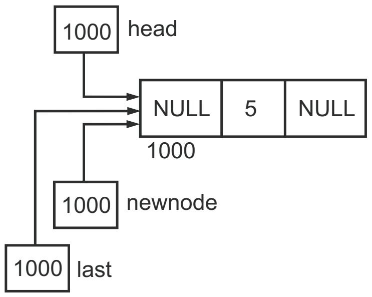


Fig. 3.32 (b)

Step 4 : Create a New Node: Consider node data as 6.

```
newnode=(struct node *)malloc (sizeof(struct node));
newnode->data=6
newnode->next=NULL;
```

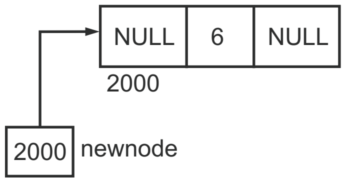


Fig. 3.32 (c)

Step 5 : Append this node in the linked list.

As head ≠ NULL, so case 2 is true.

```
last->next=newnode ;
newnode->prev=last;
last=newnode;
```

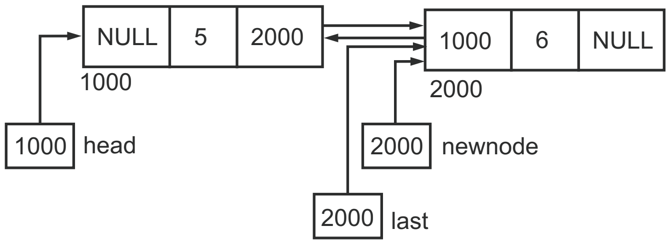


Fig. 3.32 (d)

Similarly insert node with data 7 using above steps. You will get:

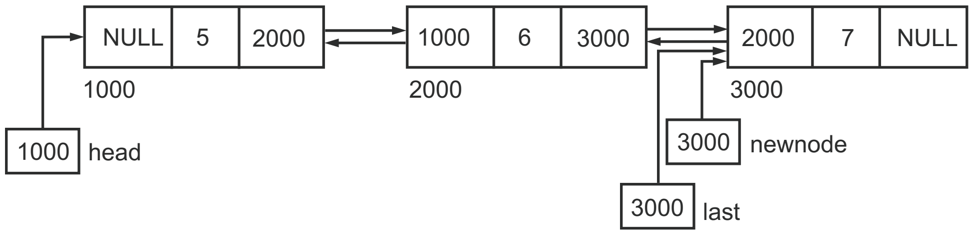


Fig. 3.32 (e)

Fig. 3.32 (a) (b) (c) (d) (e): Example of Create and Append Node in Doubly Linked List

- In similar way, we can insert 'n' nodes into the linked list.

The 'C' function to append a node into doubly linked list:

```
struct node * DLL_append (struct node *head)
{
    struct node * newnode =NULL,*temp=head;
    newnode=(struct node *)malloc (sizeof(struct node));
    newnode->prev = newnode->next=NULL;
    scanf("%d", &newnode->data);
    if(head==NULL)
        head = last = newnode;
    else
    {
        last ->next = newnode;
        newnode -> prev = last;
        last = newnode;
    }
    return head;
}
```

Program 3.9: Program to creating a Doubly Linked List (DLL).

```
#include<stdio.h>
#include<stdlib.h>
struct node  {
    int info;
    struct node* next;
    struct node* prev;
}*start=NULL;
int main()
{
    int data;
    setbuf(stdout, NULL);
    struct node *q,*tmp;
    printf("\nEnter the element to be inserted :\n");
    scanf("%d",&data);
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->prev=NULL;
    tmp->next=NULL;
    if(start == NULL)
        start = tmp;
    printf("**** Elements in Doubly Linked List ****\n");
    q=start;
    while(q!=NULL)
    {
        printf("%d\t",q->info);
        q=q->next;
    }
}
```

Output:

```
Enter the element to be inserted :
10
**** Elements in Doubly Linked List ****
10
```

3.4.2.2 Insertion of a Node in Doubly Linked List

- There are various positions where a node can be inserted in doubly linked list:
  1. Insert at the beginning of the list (first position).
  2. Insert at the end (last position).
  3. Insert at any position (middle of the list).
- Consider following DLL for insertion operation.

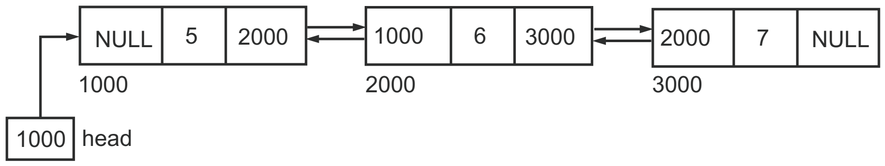


Fig. 3.33 (a): Input Doubly Linked List

1. To Insert Node at First Position: **[W-18]**

- The dotted line represents link manipulation needed to add node at the first position.

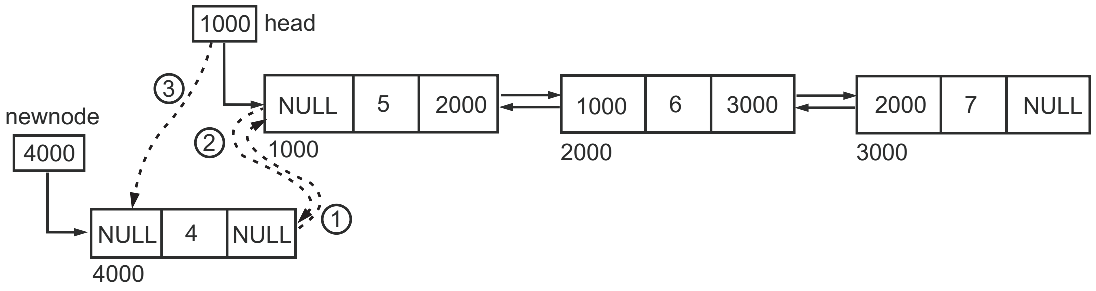


Fig. 3.33: (b) Before Insertion at the Beginning in DLL

- Using following steps, you get a linked list as shown in Fig. 3.33 (c).

```
newnode -> next = head;
head->prev=newnode;
head = newnode
```

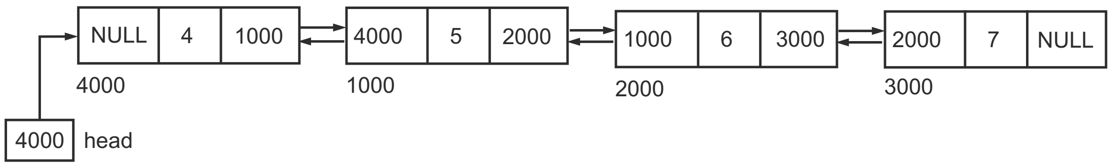


Fig. 3.33: (c) After Insertion at the Beginning in DLL

The 'C' function to insert node at beginning in doubly linked list is written as:

```
struct node * DLL_insert_front (struct node *head)
{
    struct node * newnode =NULL;
    newnode=(struct node *)malloc (sizeof(struct node));
    newnode->prev = newnode->next=NULL;
   scanf("%d", &newnode->data);
```

```
      if(head==NULL)
         head = last = newnode;
      else
      {
         newnode ->next = head;
         head -> prev = newnode;
         head = newnode;
       }
      return head;
   }
```

2. **Insertion of a node at end:**

- The dotted line represents link manipulation needed to add node at the end.

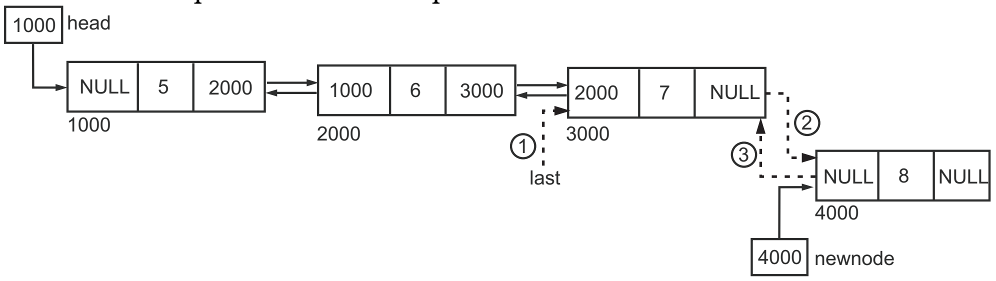


Fig. 3.34: (a) Before Insertion at end in DLL

- Using following steps, you get a linked list as shown in (b)
  - `last->next=newnode;`
  - `newnode->prev=last;`
  - `last = newnode`

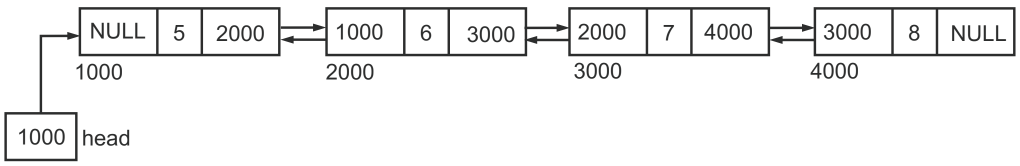


Fig. 3.34: (b) After Insertion at End in DLL

The 'C' function to insert node at end in doubly linked list is written as:

```
struct node * DLL_insert_end (struct node *head)
{
   struct node * newnode =NULL,*temp=head;
      newnode=(struct node *)malloc (sizeof(struct node));
      newnode->prev = newnode->next=NULL;
      scanf("%d", &newnode->data);
      if(head==NULL)
         head = last = newnode;
      else
      {
         last ->next = newnode;
         newnode -> prev = last;
         last = newnode;
      }
   return head;
}
```

3. Insertion of a Node at given Position:

- Consider 4 nodes in the linked list and insert a newnode after third node in the linked list.

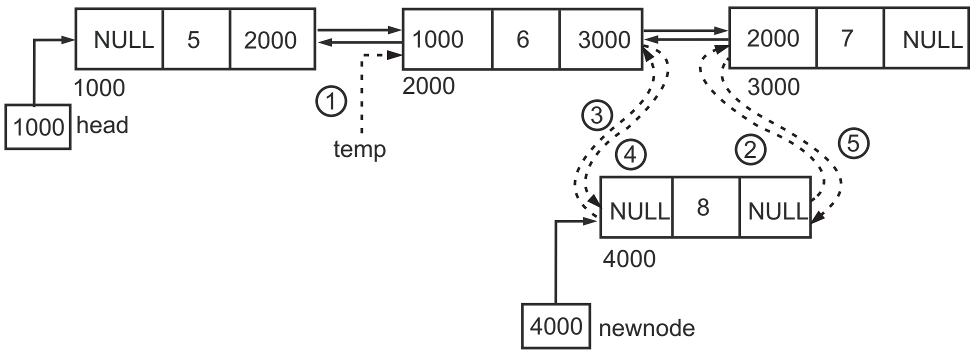


Fig. 3.35: (a) Before Insertion at Fourth Position in DLL

- The newnode is to be inserted after temp. The node which is successor of temp will now become successor of newnode.
- Using following steps, you get linked list as shown in (b).

```
Traverse temp pointer from head till (position-1)th node.
newnode -> next = temp -> next;
newnode->prev=temp;
temp->next=newnode;
newnode->next->prev=newnode;
```

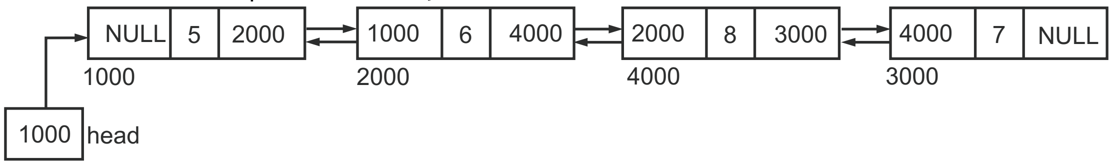


Fig. 3.35: (b) After Insertion at any Specified Node in DLL

The 'C' function to insert node at middle in doubly linked list is written as:

```
struct node * DLL_insert_middle(struct node *head, int pos)
{
    int i;
    struct node * newnode =NULL, *temp = NULL , *q = NULL;
    newnode=(struct node *)malloc (sizeof(struct node));
    newnode -> prev = newnode->next=NULL;
    scanf("%d", &newnode->data);
    if(pos == 1)
    {
        newnode ->next = head;
        head -> prev = newnode;
        head = newnode;
    }
```

```
        else
        {
            temp = head;
             for ( i = 1; i< pos-1; i++)
                temp = temp -> next;
             q = temp -> next;
             newnode ->next  = q;
             q -> prev = newnode;
             temp -> next = newnode;
             newnode -> prev = temp;
        }
        return head;
    }
```

3.4.2.3 Delete a node from Doubly Linked List

- Deletion can also have various cases like insertion:
  1. Deletion at the front of the list (first position).
  2. Deletion at the end of the list (last position).
  3. Deletion at any position in the list (middle of list).
- Consider following linked list:

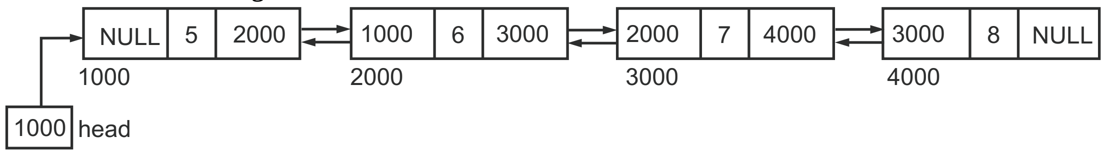


Fig. 3.36 (a): Input Doubly Linked List

1. Deletion from Beginning or First position:

- If the node at first position is to be deleted, then we need to modify the head pointer.

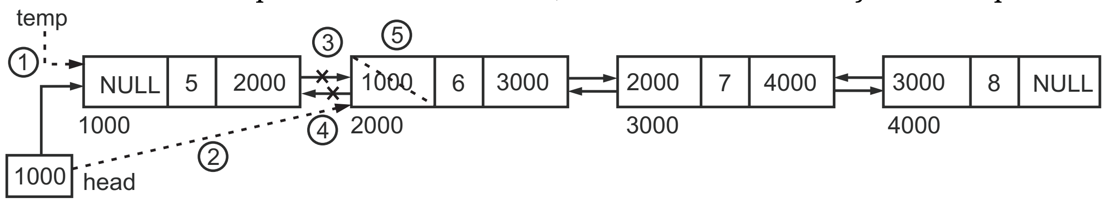


Fig. 3.36: (b) Before Deletion from Front in DLL

- Using following steps, you will get linked list as shown in Fig. 3.36 (c).

```
temp = head;
head = head -> next
temp -> next = NULL;
head -> prev = NULL;
free (temp);
```

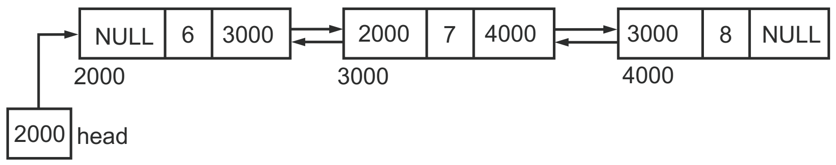


Fig. 3.36: (c) After Deletion from Front in DLL

The 'C' function to delete node (from front/beginning) from doubly linked list is written as:

```
struct node * delete_front(struct node *head)
{
   struct node * temp = NULL;
   printf("Deleted element is : %d\n", head-> data);
   if(head -> next == NULL)
   {
      free(head);
      head = last = NULL;
   }
   else
   {
      temp = head;
      head = head -> next;
      head -> prev = NULL;
      temp -> next = NULL;
      free(temp);
   }
   return head;
}
```

2. **Deletion at End:**

- Let the 'last' points to the last node to be deleted and 'temp' is the node which is predecessor of the 'last'.

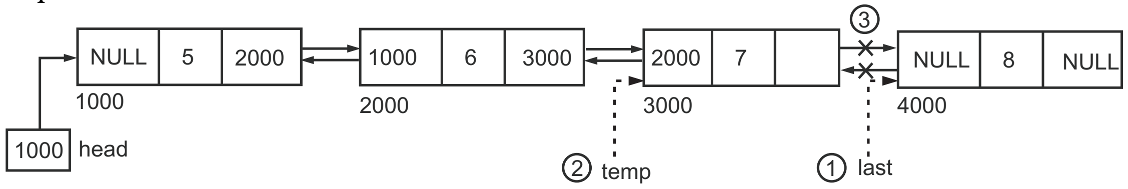


Fig. 3.37 (a): Before Deletion at End in DLL

  - `Temp -> next = NULL;`
  - `free(last);`

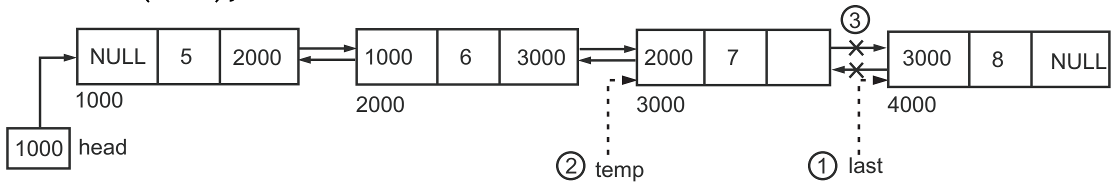


Fig. 3.37 (b): After deletion at end in DLL

The 'C' function to delete node (from end) from doubly linked list is written as:

```
struct node * DLL_delete_end(struct node *head)
{
   struct node * temp = NULL;
   printf("Deleted element is : %d\n", head-> data);
   if(head -> next == NULL)
   {
      free(head)
      head = last = NULL;
   }
   else
   {
      temp = last -> prev;
      temp -> next = NULL;
      free(last);
      last = temp;
   }
   return head;
}
```

3. Deletion from Specified Position:

- The node which is successor of temp will now become successor of pointer 'q'.

```
Traverse pointer 'q' from head to till position-1.
Set temp to q's next;
q -> next = temp -> next;
temp -> next = NULL;
free(temp);
```

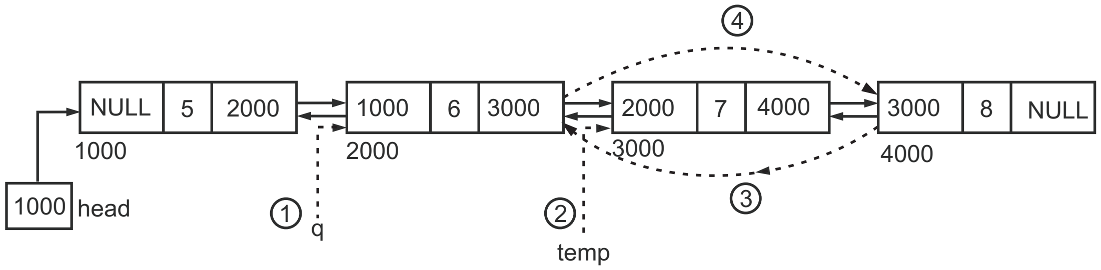


Fig. 3.38: (a) Before Deletion at specified position in DLL

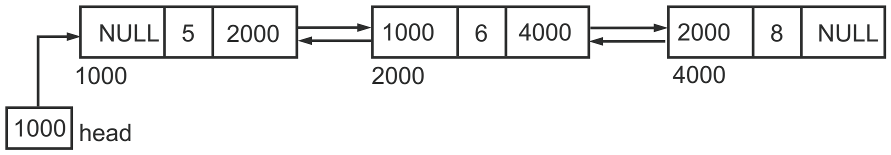


Fig. 3.38: (b) After Deletion at specified position in DLL

The 'C' function to delete node (from specific position) from doubly linked list is written as:

```
struct node * DLL_delete_middle(struct node *head, int pos)
{
   int i;
   struct node *temp = NULL , *q = NULL;
   temp = head;
   for (i = 1; i< pos-1; i++)
      temp = temp -> next;
   q = temp -> next;
   temp -> next  = q -> next;
   q -> next -> prev = temp;
   q -> next = q -> prev = NULL;
   printf("Deleted element is : %d\n", q -> data);
   free(q);
   return head;
}
```

3.4.2.4 Traversing a List

- Given a head pointer to the DLL, traversal is the same as in singly linked list. The advantage of DLL is the list can be traverse in both (forward and backward) directions, using next and previous (prev) pointers.
- **The 'C' function to traverse node in doubly linked list is written as:**

```
void display(struct node *head)
{
   struct node *temp=head;
   printf("\nLinked List contents are : \n");
   while(temp != NULL)
   {
      printf("%d\t",temp->data);
      temp=temp->next;
   }
}
```

Program 3.10: Menu driven program for doubly linked list (create, insert, delete, display).

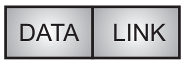
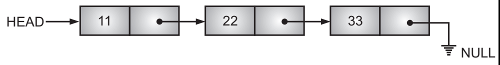
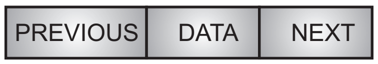
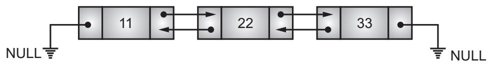
```
#include <stdio.h>
#include <malloc.h>
#include <process.h>
void create();
void insert_at_beg();
void insert_at_end();
void insert_after_pos();
void del();
void search();
void display();
struct node  {
   int info;
   struct node* next;
   struct node* prev;
}*start=NULL;
int data,item,n1,pos,i,m;
int main()
{
   int n;
   setbuf(stdout, NULL);
   printf("\n****Doubly Linked List*****\n");
   printf("\n1.Create\n2.Insert at Beginning\n3.Insert at
           End\n4.Insert After Position\n5.Delete\n6.Display\n7.Exit\n");
   while(1)
   {
      printf("\nEnter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End
                       4.Insert after Pos. 5.Delete 6.Display 7.Exit)\n");
      scanf("%d",&n);
      switch(n)
      {
      case 1:
         create();
         break;
      case 2:
         insert_at_beg();
         break;
      case 3:
         insert_at_end();
         break;
      case 4:
         insert_after_pos();
         break;
      case 5:
         del();
         break;
```

```
            case 6:
                display();
                break;
            case 7:
                exit(0);
            default:
                printf("\nWrong Choice !!\n");
            }
        }
        return 0;
    }
    void create()
    {
        int data;
        struct node *tmp;
        printf("\nEnter the first element to be inserted :\n");
        scanf("%d",&data);
        tmp=malloc(sizeof(struct node));
        tmp->info=data;
        tmp->prev=NULL;
        tmp->next=NULL;
        if(start == NULL)
            start = tmp;
        display();
    }
    void insert_at_beg()
    {
        int data;
        struct node *tmp;
        printf("\nEnter the element to be inserted :\n");
        scanf("%d",&data);
        tmp=malloc(sizeof(struct node));
        tmp->info=data;
        tmp->prev=NULL;
        tmp->next=NULL;
        if(start == NULL)
            start = tmp;
        else
        {
            start->prev = tmp;
            tmp->next = start;
            start = tmp;
        }
        display();
    }
    void insert_at_end()
    {
        int data;
        struct node *q,*tmp;
        printf("\nEnter the element to be inserted :\n");
        scanf("%d",&data);
        tmp=malloc(sizeof(struct node));
        tmp->info=data;
        tmp->prev=NULL;
        tmp->next=NULL;
        if(start == NULL)
            start = tmp;
        else
        {
            q=start;
            while(q->next != NULL)
                q = q->next; // Go To last Node
            q->next = tmp;
            tmp->prev = q;
        }
        display();
    }
    void insert_after_pos()
    {
        int data;
        struct node *q,*tmp;
        tmp=malloc(sizeof(struct node));
        printf("\nEnter the element to be inserted :\n");
        scanf("%d",&data);
        tmp->info=data;
        tmp->prev=NULL;
        tmp->next=NULL;
        if(start==NULL)
        {
            start=tmp;
        }
        else
        {
            printf("Enter index after which element to be inserted :\n");
            scanf("%d",&pos);
            q=start;
            for(i=0;i<pos;i++)
            {
                q = q->next;
            }
            tmp->next = q->next;
            if(q->next!=NULL)
            {
                q->next->prev=tmp;
            }
            q->next = tmp;
            tmp->prev=q;
            display();
        }
    }
    void del()
    {
        struct node *tmp,*q,*prev;
        printf("Enter the element to be deleted :\n");
        scanf("%d",&data);
        if(start->info==data)  //deletion of first node
        {
            tmp=start;
            if(tmp->next!=NULL)
            {
                start->next->prev=NULL;
            }
            start=start->next;
            free(tmp);
            display();
            return;
        }
        q=start;
        while(q->next->next!=NULL)  //deletion of middle node
        {
           if(q->next->info==data)
           {
              prev=q->next->prev;
              tmp=q->next;
              q->next=tmp->next;
              q->next->prev=prev;
              free(tmp);
              display();
              return;
           }
           q=q->next;
        }
        if(q->next->info==data)  //deletion at end
        {
           tmp=q->next;
           q->next=NULL;
           free(tmp);
           display();
           return;
        }
        printf("\nElement not found \n");
     }
     void  display()
     {       struct node *q;
     if(start==NULL)
        printf("List is empty!!\n");
     else
     {
        printf("**** Elements in Doubly Linked List ****\n");
        q=start;
        while(q!=NULL)
        {
           printf("%d\t",q->info);
           q=q->next;
        }
     }
     }
```

Output:

```
****Doubly Linked List*****
1.Create
2.Insert at Beginning
3.Insert at End
4.Insert After Position
5.Delete
6.Display
7.Exit
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete 6.Display 7.Exit)
1
Enter the first element to be inserted :
10
**** Elements in Doubly Linked List ****
10
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete 6.Display 7.Exit)
2
Enter the element to be inserted :
5
**** Elements in Doubly Linked List ****
5  10
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete 6.Display 7.Exit)
3
Enter the element to be inserted :
20
**** Elements in Doubly Linked List ****
5  10 20
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete 6.Display 7.Exit)
4
Enter the element to be inserted :
15
Enter index after which element to be inserted :
1
**** Elements in Doubly Linked List ****
5  10 15 20
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete 6.Display 7.Exit)
5
Enter the element to be deleted :
20
**** Elements in Doubly Linked List ****
5  10 15
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete 6.Display 7.Exit)
```

- Following table compares SLL and DLL:

Sr. No.	Singly Linked List (SLL)	Doubly Linked List (DLL)
1.	Singly linked list allows us to go one way direction.	Doubly linked list has two way directions next and previous.
2.	Singly linked list uses less memory per node (one pointer).	Doubly linked list uses more memory per node.
3.	Complexity of Insertion and Deletion at known position is O(n).	Complexity of Insertion and Deletion at known position is O(1).
4.	If we need to save memory in need to update node values frequently and searching is not required, we can use singly linked list.	If we need faster performance in searching and memory is not a limitation we use doubly linked list.
5.	In single list each node contains at least two parts: (i) INFO, and (ii) LINK.	In doubly linked list Each node contains at least three parts: (i) INFO, (ii) LINK to next node, and (iii) LINK to previous node.
6.	Singly linked list is unidirectional i.e., only one direction.	It is bidirectional.
7.	Diagram:  Element of a singly linked list  A singly linked list	Diagram:  Element of a doubly linked list  A doubly linked list
8.	Singly linked list can mostly be used for stacks.	They can be used to implement stacks, heaps, binary trees.

Data structure	Time Complexity								Space complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly-Linked list	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked list	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

3.4.3 Operations on Circular Linked List (CLL)

- So far we discuss the linked list are linear linked list. All elements of a linked list can be accessed by first setting up a pointer to the first node and then traversing the entire list.
- The last node's next field is set to null. Instead of that, store the address of first node in the last nodes next field. Such a linked list is called circular linked list.

Definition:

- A circular list is a linked list in which last node points to the head node.
- A circular list has neither a beginning nor end. From any node, it is possible to reach to any other node.
- Circular linked list could be either singly or doubly linked list. Time slicing and memory management are two applications of Circular Linked List (CLL).

Types of Circular Linked List:

- The two types of circular linked list are explained below:

1. Circular Singly Linked List:

- In single linked list, every node points to its next node in the sequence and the last node points NULL.
- But in circular singly linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

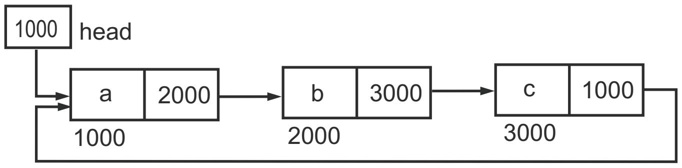


Fig. 3.39: Singly Circular Linked List (CLL)

- The head pointer points to the first node of the list. From last node we can access a first node.

2. Circular Doubly Linked List:

- Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node.
- The last node of the list contains the address of the first node of the list. The first node of the list also contains address of the last node in its previous pointer.

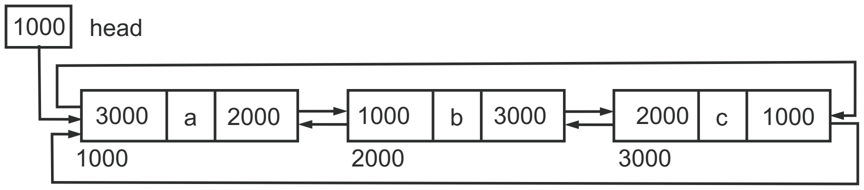


Fig. 3.40: Doubly Circular Linked List

- The last node's next link is set to the first node. The first node's previous link is set to the last node of the list.

3.4.3.1 Circular Singly Linked List

- A single linked circular list means the last node points to the first node and there is only one link between the nodes of the linked list.

1. Creating a Circular Singly Linked List:

- While traversing a list the problem is checking of end of the list, since null is not present.
- If we write while (node -> next != head) this would enter in the infinite loop. Hence, we detect the first node by creating the head node, which holds the address of the first node. We can also keep a counter, which is incremented when nodes are created and end of the node can be found.
- We can use the structure same as singly link list: **[April 18]**

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

Example:

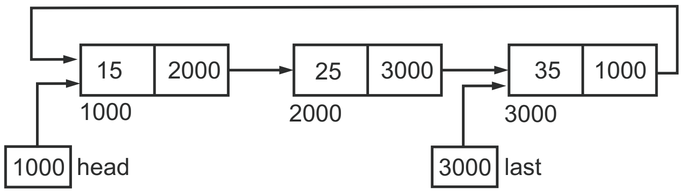


Fig. 3.41 (a): Example of Circular Singly Linked List

2. Insert the Element in Circular Singly Linked List:

- The following Fig. 3.41 (a) and Fig. 3.41 (b) show the insertion in circular singly linked list.

(i) Insertion at Beginning in CSLL:

- We can use the following steps to insert a new node at beginning of the circular linked list.

```
Step 1 : Create a newnode with given value.
Step 2 : Check whether list is Empty (head == NULL)
Step 3 : If it is Empty then,
           set head = newnode
           newnode ->next = head
Step 4 : If it is not Empty then,
           Set temp = head
Step 5 : Traverse the temp to its next node until it reaches to last node
           until (temp -> next == head)
Step 6 : set newnode -> next = head
           temp -> next = newnode
           head = newnode
```

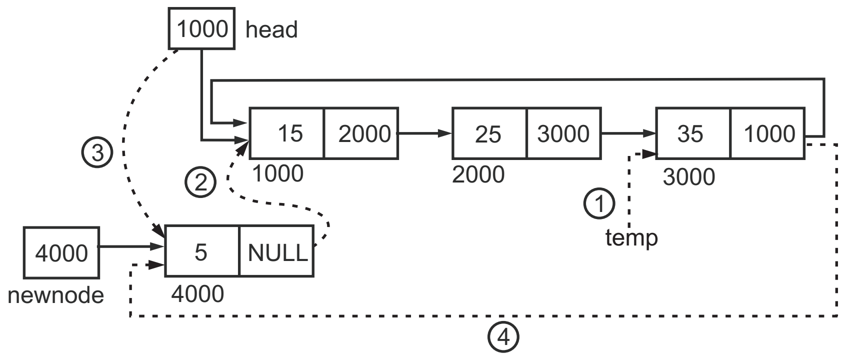


Fig. 3.41 (b): Insertion at front in CSLL

- Dotted line shows the situation after insertion.

(ii) Insertion at End in CSLL:

- We can use the following steps to insert a new node at end of the circular linked list:

  Step 1 : Create a newnode with given value.
  Step 2 : Check whether list is Empty `(head == NULL)`
  Step 3 : If it is Empty then,
  ```
  set head = newnode
  newnode ->next = head
  ```
  Step 4 : If it is not Empty then,
  ```
  Set temp = head
  ```
  Step 5 : Traverse the temp to its next node until it reaches to last node
  ```
  until (temp -> next == head)
  ```
  Step 6 :
  ```
  set temp-> next = newnode
    newnode -> next = head
  ```

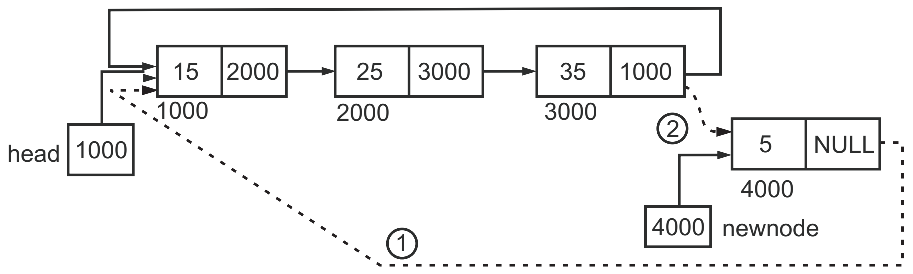


Fig. 3.41 (c): Insertion at end in CSLL

(iii) Insertion at given Position in CSLL:

- Inserting node at given position of circular singly linked list operation is same as inserting node at middle of singly linked list.
- **The 'C' function for insertion in circular singly linked list:**

```
struct node * CSLL_insert(struct node *head, int pos)
{
    int i;
    struct node *newnode=NULL,*temp=NULL;
    newnode=(struct node *)malloc(sizeof(struct node));
```

```
        if(newnode==NULL)
           printf("Memory allocation failed\n");
        else
        {
           printf("Enter node data\n");
           scanf("%d",&newnode->data);
           newnode->next=NULL;
           if(pos==1)
           {
              if(head==NULL)
              {
                 head=newnode;
                 head->next=head;
              }
              else
              {
                 temp=head;
                    while(temp->next!=head)
                       temp=temp->next;
                 newnode->next=head;
                 temp->next=newnode;
                 head=newnode;
              }
           }
           else
           {
              temp=head;
              for(i = 1;i<pos-1; i++)
                 temp=temp->next;
              newnode->next=temp->next;
              temp->next=newnode;
           }
        ++total_nodes;
     }
     return head;
  }
```

3. Deletion in Circular Singly Linked List (CSLL):

- In a circular singly linked list, the deletion operation can be performed in three ways those are as follows...
  (i) Deleting from Beginning of the list.
  (ii) Deleting from End of the list.
  (iii) Deleting a Specific Node.

1. Delete from beginning in CSLL:

- We can use the following steps to delete a node from beginning of the circular linked list:

  Step 1 : If list is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize both 'temp' and 'temp1' with head.

  Step 2 : If list is having only one node (`temp -> next == head`) then,
  ```
  set head = NULL
  delete temp
  ```
  Step 3 : If list has more than one node then,
  move the temp until it reaches to the last node (until temp-> next == head)
  ```
  set head = head -> next,
  temp-> next = head
  delete temp1
  ```
- The Fig. 3.42 shows deletion at front in circular singly, linked list.

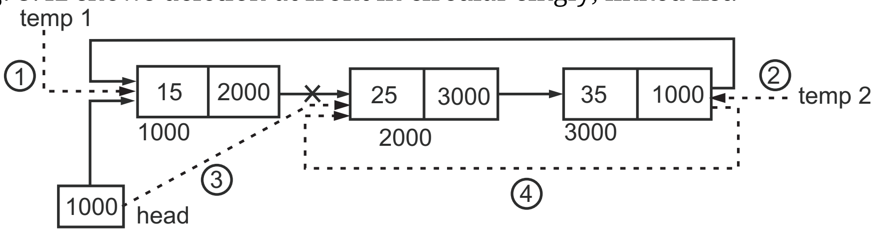


Fig. 3.42 (a): Deletion at beginning in CSLL

The 'C' function to delete node (from front) from circular singly linked list is written as:

```
struct node * CSLL_delete_front(struct node *head)
{
   struct node * temp = NULL, *temp1=NULL;
   printf("Deleted element is : %d\n", head-> data);
   if(head -> next == head)
   {
      free(head);
      head = NULL;
   }
   else
   {
      temp = temp1 = head;
      while( temp -> next != head )
         temp = temp -> next;
      head = head -> next;
      temp -> next = head;
      temp1 -> next = NULL;
      free (temp1);
   }
   return head;
}
```

2. Delete from end in CSLL:

- We can use the following steps to delete a node from end of the circular linked list.

  Step 1 : If list is Not Empty then, define two Node pointers 'temp' and 'temp1' and initialize 'temp' with head.

  Step 2 : If list has only one node (`temp -> next == head`) then, set `head = NULL` and delete temp. (Setting Empty list condition).

  Step 3 : If list has more than one node then, `set 'temp1 = temp'` and move temp to its next node. Repeat the same until temp reaches to the last node in the list. (`until temp -> next == head`).

  Step 4 : `Set temp1 -> next = head` and delete temp.

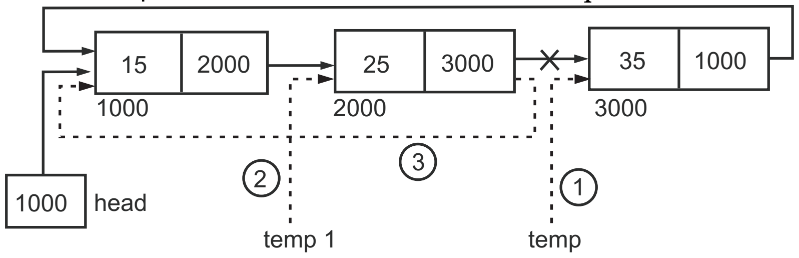


Fig. 3.42 (b): Deletion from end in CSLL

The 'C' function to delete node (from end) from circular singly linked list is written as:

```
struct node * CSLL_delete_end(struct node *head)
{
   struct node * temp = head, *temp1=NULL;
   if(head -> next == head)
   {
      free (head);
      head = NULL;
   }
   else
   {
      temp1 = temp;
      temp = temp -> next;
      while( temp -> next != head )
      {
         temp = temp -> next;
         temp1 = temp1 -> next;
      }
      temp1 -> next = head;
      temp -> next =  NULL;
      free (temp);
   }
   printf("Deleted element is : %d\n", temp-> data);
   return head;
}
```

3. Delete from specific position in CSLL:

- Deleting node at given position of circular linked list operation is same as deleting node at middle of singly linked list. So we can refer Fig. 3.42 (a) and Fig. 3.42 (b) for this operation.

  The 'C' function for deletion from middle in circular singly linked list:

```
struct node * CSLL_delete_middle(struct node *head, int pos)
{
   int i;
   struct node *temp=NULL,*q=NULL;
   temp=head;
   if(pos==1)
   {
      if(head->next==head)
      {
         printf("Deleted element is : %d\n",head->data);
         free(head);
         head=NULL;
         return head;
      }
      else
      {
         temp=q=head;
         while(q->next!=head)
         q=q->next;
         q->next=head->next;
         head=head->next;
      }
   }
   else
   {
      q=head;
      for(i = 1;i<pos-1; i++)
           q=q->next;
      temp=q->next;
      q->next=temp->next;
   }
   printf("Deleted element is : %d\n",temp->data);
   temp->next=NULL;
   free(temp);
   return head;
}
```

Traversing / Display Circular Singly Linked List:

- Traversal is a technique of visiting each and every node. We traverse a circular singly linked list until we reach the same node where we started.

  The 'C' function to traversal a list in 'C' is written as:

```
void display(struct node *head)
{
   struct node *temp = head;
   printf("\nLinked List contents are : \n");
   //start from the beginning
   if(head != NULL)
   {
      while(temp->next != temp)
      {
         printf("(%d,%d) ", temp -> key, temp -> data);
         temp = temp ->next;
      }
   }
   printf(" ]");
}
```

3.4.3.2 Circular Doubly Linked List

- In circular doubly linked list last node's next link is set to the first node of the list and the first node's previous link is set to the last node of the list. This gives access to the last node directly from first node.

1. **Insertion in Circular Doubly Linked List:**
   (i) **Insertion at Beginning in CDLL:** We can use the following steps to insert a node at end circular doubly linked list.
   - `Newnode -> next=head;`
   - `Newnode -> prev = head->prev;`
   - `head -> prev = newnode;`
   - `temp -> next = newnode;`
   - `newnode = head;`

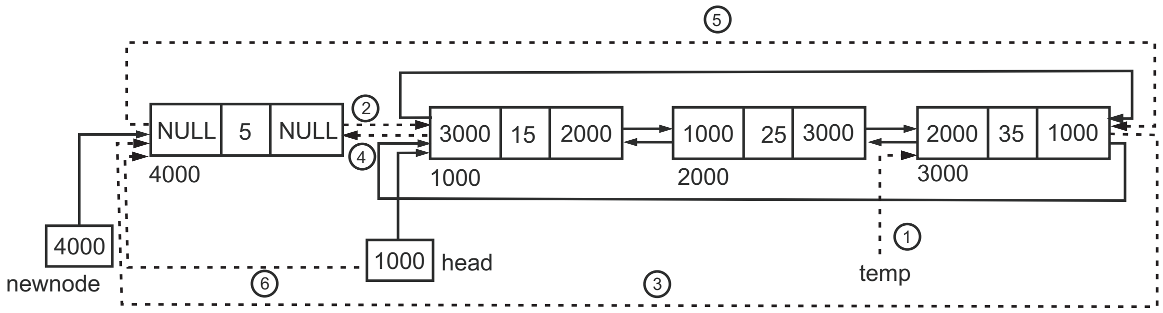


Fig. 3.43 (a): Before insertion at front in CDLL

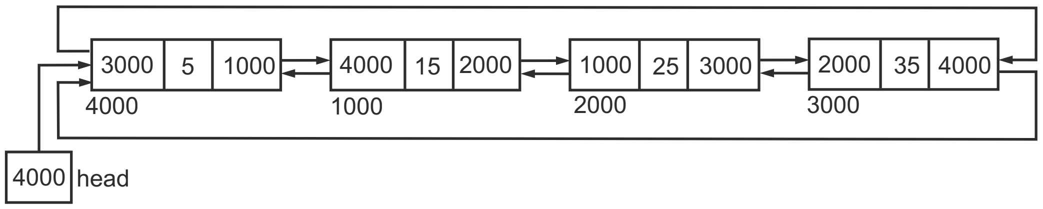


Fig. 3.43 (b): After Insertion at Front in CDLL

- **The 'C' function to insert node at beginning in circular doubly linked list is written as:**

```
struct node * CDLL_insert_front (struct node *head)
{
    struct node * newnode =NULL;
    newnode=(struct node *)malloc (sizeof(struct node));
    newnode->prev = newnode->next=NULL;
    scanf("%d", &newnode->data);
    if(head==NULL)
    {
        head = newnode;
        head -> prev = head -> next = head;
    }
    else
    {
        temp = head ->prev;
        newnode ->next = head;
        head -> prev = newnode;
        temp - > next = newnode;
        newnode -> prev = temp;
        head = newnode;
    }
    return head;
}
```

(ii) Insertion at End in CDLL:

- We can use the following steps to insert a node at the end of circular doubly linked list:
  - `newnode -> next = head;`
  - `temp -> next = newnode;`
  - `newnode -> prev = head;`
  - `head->prev = newnode;`

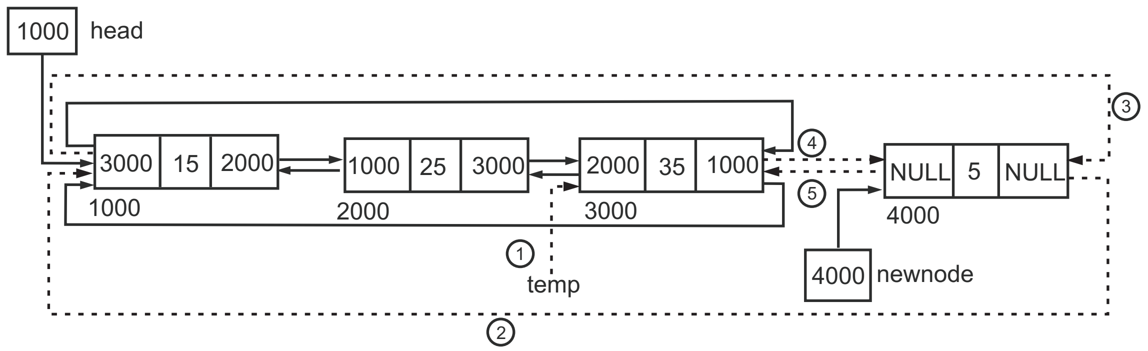


Fig. 3.43 (c): Insertion at End in CDLL

- **The 'C' function to insert node at end in circular doubly linked list is written as:**

```
struct node * CDLL_insert_end (struct node *head)
{
    struct node * newnode =NULL;
    newnode=(struct node *)malloc (sizeof(struct node));
    newnode->prev = newnode->next=NULL;
    scanf("%d", &newnode->data);
    if(head==NULL)
    {
       head = newnode;
       head -> prev = head -> next = head;
    }
    else
    {
       temp = head -> prev;
       newnode ->next = head;
       head -> prev = newnode;
       temp - > next = newnode;
       newnode -> prev = temp;
    }
    return head;
}
```

(iii) Insertion at Middle in CDLL:

- Inserting node at given position of circular doubly linked list operation is same as inserting node at middle of doubly linked list. So we can refer to Fig. 3.43(a) and 3.43 (b) for this operation.

2. **Deletion in circular doubly linked list:**

- In a double linked list, the deletion operation can be performed in three ways as follows:
  1. Deleting from Beginning of the list.
  2. Deleting from End of the list.
  3. Deleting a Specific Node.

(i) Deletion from beginning in CDLL:

- We can use the following steps to delete a node from beginning of the double linked list:

Step 1 : If list is not Empty then,
```
    define a node pointer 'temp', 'temp1'
    initialize temp with head and temp1 with head->prev.
```
Step 2 : Check whether list is having only one node (`temp → previous = temp → next`)

Step 3 : If it is TRUE, then set head to NULL and delete temp (Setting Empty list conditions)

Step 4 : If it is FALSE, then,
Move head to its next node i.e `head = head -> next`
```
temp1 → next = head
head → prev = temp1
delete temp.
```

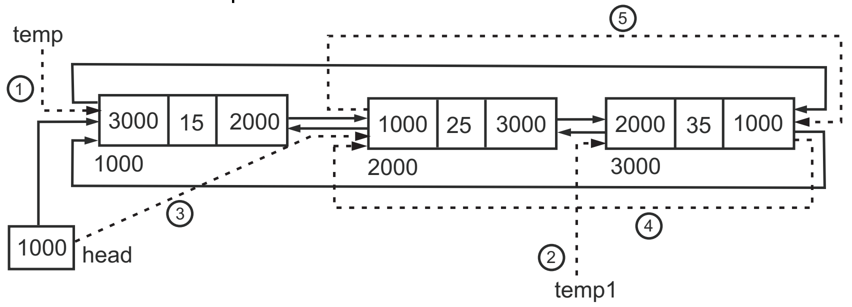


Fig. 3.44 (a): Deletion from beginning in CDLL

- **The 'C' function to delete node (from front) from circular doubly linked list is written as:**

```
struct node * CDLL_delete_front(struct node *head)
{
   struct node * temp = NULL, * temp1= NULL;
   printf("Deleted element is : %d\n", head-> data);
   if(head -> next == head)
   {
      free (head);
      head = last = NULL;
   }
   else
   {
      temp = head;
      temp1 = head -> prev;
      head = head -> next;
      head -> prev = temp1;
      temp1 -> next = head;
      temp -> prev = temp -> next = NULL;
      free (temp);
   }
   return head;
}
```

(ii) Deletion from end in CDLL:

- We can use the following steps to delete a node from end of the double linked list.

Step 1 : If list is not Empty then,
```
define a node pointer 'temp' and ‘temp1’
```
Step 2 : if list has only one node then
delete head and assign NULL to head. (Setting Empty list condition)

Step 3 : If list has more than one node then,
```
initialize temp with head-> prev and temp1 with temp -> prev.
     set temp1 -> next = head
head -> prev = temp1
```
Step 4 : Assign NULL to temp -> prev and temp -> next and delete temp.

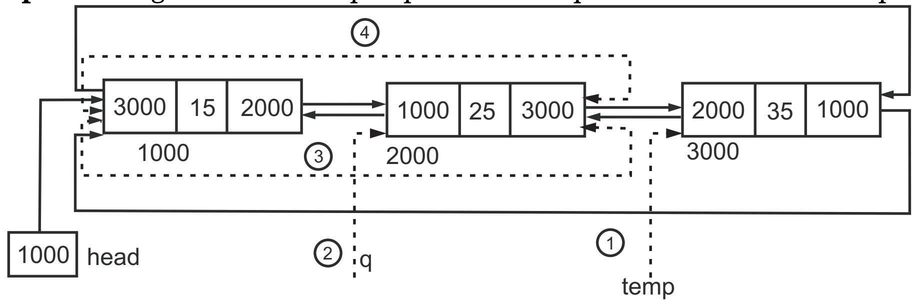


Fig. 3.44 (b): Deletion from end in CDLL

- **The ‘C’ function to delete node (from end) from circular doubly linked list is written as:**

```
struct node * CDLL_delete_end(struct node *head)
{
   struct node * temp = head -> prev , *q = temp -> prev;
   if(head -> next == head)
   {
      printf("Deleted element is : %d\n", head-> data);
      free(head)
      head = NULL;
   }
   else
   {
      temp = head -> prev ;
      q = temp -> prev;
      q -> next = head;
      head -> prev = q;
      temp -> next = temp -> prev = NULL;
      printf("Deleted element is : %d\n", temp-> data);
      free(temp);
   }
   return head;
}
```

(iii) Deleting a Specific Node in CDLL:

- Deleting node at given position of circular doubly linked list operation is same as deleting node at middle of doubly linked list. So we can refer Fig. 3.44 (a) and Fig. 3.44 (b) for this operation.

Program 3.11: Menu driven program for circular single linked list (create, insert, delete, search, display).

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<stdlib.h>
void create();
void insert_at_beg();
void insert_at_end();
void insert_after_pos();
void del_at_beg();
void del_after_pos();
void del_at_end();
void search();
void display();
struct node
{
   int info;
   struct node *link;
} *last;
int data,pos;
int main()
{
   int ch;
   last=NULL;
   setbuf(stdout, NULL);
   printf("\n****Circular Linked List*****\n");
   printf("\n1.Create\n2.Insert at Beginning\n 3.Insert at End\n 4.Insert
   After Position\n 5.Delete at Beginning\n 6.Delete at End\n 7.Delete
   after Position\n 8.Search\n 9.Display\n10.Exit\n");
   while(1)
   {
   printf("\nEnter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End
   4.Insert after Pos. 5.Delete at Beginning 6.Delete at end 7.Delete after
   Pos. 8.Search 9.Display 10.Exit)\n");
   scanf("%d",&ch);
```

```
    switch(ch)
    {
    case 1:
       create();
       break;
    case 2:
       insert_at_beg();
       break;
    case 3:
       insert_at_end();
       break;
    case 4:
       insert_after_pos();
       break;
    case 5:
       del_at_beg();
       break;
    case 6:
       del_at_end();
       break;
    case 7:
       del_after_pos();
       break;
    case 8:
       search();
       break;
    case 9:
       display();
       break;
    case 10:
       exit(0);
    default:
       printf("\nWrong Choice !!\n");
    }
    }
    return 0;
}
```

```
void create()
{
    struct node *tmp;
    printf("Enter element :\n");
    scanf("%d",&data);
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    if(last==NULL)
    {
        last=tmp;
        tmp->link=last;
    }
    else
    {
        tmp->link=last->link;
        last->link=tmp;
        last=tmp;
    }
}
void insert_at_beg()
{
    struct node *tmp;
    printf("Enter the element to be inserted :\n");
    scanf("%d",&data);
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=last->link;
    last->link=tmp;
    display();
}
void insert_at_end()
{
    struct node *tmp;
    printf("Enter element :\n");
    scanf("%d",&data);
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=last->link;
    last->link=tmp;
    last=tmp;
    display();
}
```

```
void insert_after_pos()
{
   struct node *tmp, *q;
   printf("Enter the elements : ");
   scanf("%d",&data);
   printf("Enter index after which element to be inserted : ");
   scanf("%d",&pos);
   int i;
   q=last->link;
   for(i=0;i<pos;i++)
   {
      q=q->link;
      if(q==last->link)
      {
         printf("There are  less  elements\n");
         return;
      }
   }
   tmp=malloc(sizeof(struct node));
   tmp->link=q->link;
   tmp->info=data;
   q->link=tmp;
   if(q==last)
      last=tmp;
   display();
}
void del_at_beg()
{
   struct node *q;
   if(last->link==last)  //single node in the list
   {
      last=NULL;
      free(last);
      printf("List is empty!!");
      return;
   }
   if(last==NULL)
   {
      printf("List is empty");
   }
```

```
    q=last->link;
    last->link=q->link;
    free(q);
    printf("First element deleted!!\n");
    display();
    return;
}
void del_at_end()
{
    struct node *q;
    q=last->link;
    if(last==NULL)
    {
        printf("List is empty");
    }
    if(last->link==last)  //single node in the list
    {
        last=NULL;
        free(last);
        printf("List is empty!!");
        return;
    }
    while(q->link!=last)//traverse upto Last pointer
    {
        q=q->link;
    }
    q->link=last->link;
    free(last);
    last=q;
    printf("Last element deleted!!\n");
    display();
    return;
}
void del_after_pos()
{
    struct node *tmp, *q;
    int i,pos;
    printf("Enter index after which element to be deleted : ");
    scanf("%d",&pos);
    q=last->link;
    for(i=0;i<pos;i++)
    {
       q=q->link;
       if(q==last)
       {
          printf("There are  less  elements\n");
          return;
       }
    }
    if(q->link==last)
    {
       q->link=last->link;
       free(last);
       last=q;
       display();
       return;
    }
    tmp=q->link;
    q->link=tmp->link;
    free(tmp);
    display();
    return;
 }
 void display()
 {
    struct node *q;
    if(last==NULL)
    {
       printf("List is empty!!\n");
       return;
    }
    printf("**** Elements in Circular Linked List ****\n");
    q=last->link;
    while(q!=last)
    {
       printf("%d\t",q->info);
       q=q->link;
    }
    printf("%d\t",last->info);
 }
```

```
void search()
{
    struct node *tmp;
    int i=0,item;
    printf("\nEnter the element to be searched :");
    scanf("%d",&item);
    tmp=last->link;
    while(tmp!=last)
    {
        if(tmp->info==item)
        {
            printf("\n Element %d is found at index : %d \n",item,i);
            return;
        }
        tmp=tmp->link;
        ++i;
    }
    if(last->info==item)
        printf("\nElement %d is found at index : %d \n",item,i);
    else
        printf("\nElement not found \n");
}
```

Output:

```
****Circular Linked List*****
1.Create
2.Insert at Beginning
3.Insert at End
4.Insert After Position
5.Delete at Beginning
6.Delete at End
7.Delete after Position
8.Search
9.Display
10.Exit
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete at Beginning 6.Delete at end 7.Delete after Pos.
8.Search 9.Display 10.Exit)
1
Enter element :
10
```

```
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete at Beginning 6.Delete at end 7.Delete after Pos.
8.Search 9.Display 10.Exit)
2
Enter the element to be inserted :
5
**** Elements in Circular Linked List ****
5  10
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete at Beginning 6.Delete at end 7.Delete after Pos.
8.Search 9.Display 10.Exit)
3
Enter element :
15
**** Elements in Circular Linked List ****
5  10 15
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete at Beginning 6.Delete at end 7.Delete after Pos.
8.Search 9.Display 10.Exit)
4
Enter the elements : 20
Enter index after which element to be inserted : 2
**** Elements in Circular Linked List ****
5  10 15 20
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete at Beginning 6.Delete at end 7.Delete after Pos.
8.Search 9.Display 10.Exit)
5
First element deleted!!
**** Elements in Circular Linked List ****
10 15 20
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete at Beginning 6.Delete at end 7.Delete after Pos.
8.Search 9.Display 10.Exit)
6
Last element deleted!!
**** Elements in Circular Linked List ****
10 15
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete at Beginning 6.Delete at end 7.Delete after Pos.
8.Search 9.Display 10.Exit)
7
```

```
Enter index after which element to be deleted : 0
**** Elements in Circular Linked List ****
10
Enter Your Choice :(1.Create 2.Insert at Beg. 3.Insert at End 4.Insert
after Pos. 5.Delete at Beginning 6.Delete at end 7.Delete after Pos.
8.Search 9.Display 10.Exit)
```

- Following table differentiate between singly, doubly and circularly linked lists.

Parameters	Singly Linked List	Doubly Linked List	Circular Linked List
Concept	One way direction.	Two way direction.	One way direction in a circular manner.
Has head	Yes	Yes	No-because tail will refer to first node.
Has tail	Yes	Yes	Yes
No of Node	1-next node	2-next node and previous node	1-next node
insert()	O(n)	O(1)	O(n)
delete()	O(n)	O(1)	O(1)

3.5 APPLICATIONS OF LINKED LIST [Oct. 16]

- The applications of linked list are as follows:
  1. Polynomial manipulation.
  2. Garbage collection.
  3. Linked dictionary.
  4. Sorting and Merging linked lists.
  5. Searching in the list.
- Let us discuss few of them:
- **Garbage Collection:** Operating system uses free space management for allocation of blocks. In order to re-use the memory after deallocation of block, operating system uses linked list to point the free block in the list. For good memory utilization, operating system periodically collects all the free blocks and insert into free pool(list of free blocks). Any technique that does this collection is called garbage collection.
- **Linked Dictionary:** While compiling a program the linked dictionary has an important role. In compiler, the organization and maintenance of dictionary holding names and their corresponding values are maintained in the linked dictionary. It is also called symbol table.

  Using linked list, the symbols are easily inserted. To search a symbol, list is traverse.
- **Searching:** In order to search a particular number, linked list is traverse. When the given number matches with some number in list, then search is successful. Occurrence, of a particular number in a list for number of times can also be found.

3.5.1 Polynomial Representation

- Linked lists are used to represent and manipulate polynomials. In a linked list repro sentation of polynomial each term of polynomial is considered as a Node.
- In a polynomial representation each Node contains three fields :
  1. Coefficient field.
  2. Exponent field.
  3. Link field.

  Here, we give a structure of Node used for polynomial representation.

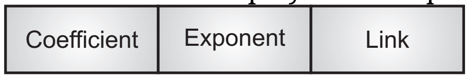
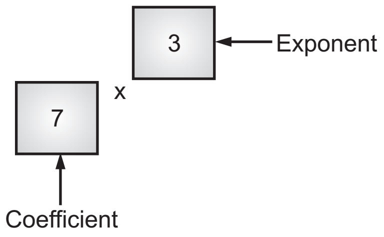


Fig. 3.45: Structures of Node used for Polynomial Representation

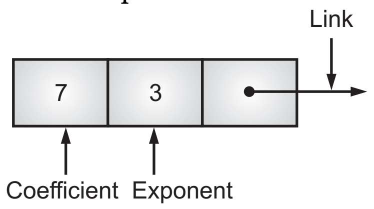
- Coefficient field is used to hold the value of coefficient.
- Exponent field is used to contains the value of exponent.
- Link field contains the address of the next term used in the polynomial.

Example: Represent the polynomial $f(x) = 7x^3 + 2x^2 + 3x + 1$ by using linked list.

Ans.: Given polynomial: $f(x) = 7x^3 + 2x^2 + 3x + 1$. Let us consider first term of polynomial as:



In the form of Node above term is represented.



In the same fashion we represent the whole polynomial as:

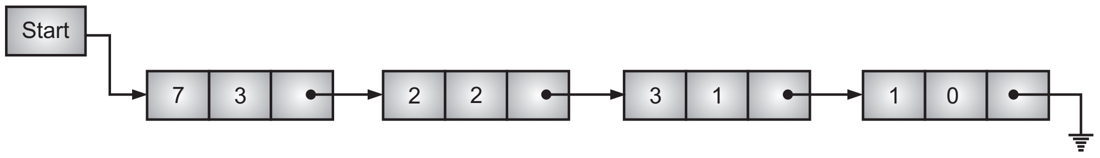


Fig. 3.46: Representation using Linked List

Example: Consider a polynomial $f(x) = 3x^5 + 4x^3 - 9x^2 - 8$. It can be represented in memory using a singly linked list with four nodes. Note that while representing a polynomial; it is assumed that the exponent of cach successor node is less than that of precedessor node.

Ans.: Fig. 3.47 shows the representation of the polynomial f(x) using a singly linked list.

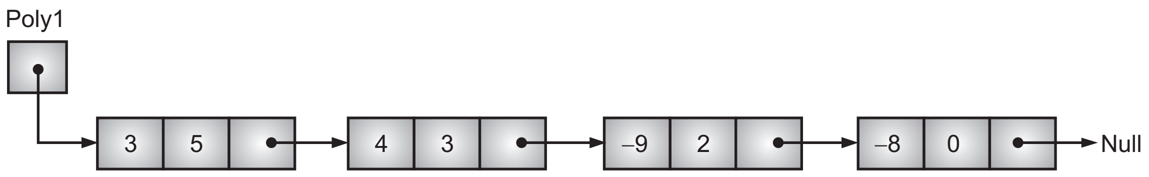


Fig. 3.47: A Linked List Representing Polynomial f(x)

Example: Represent the polynomial using linked list $f(x) = 3x^2 + 2x + 5$.

Ans.: Given polynomial:

$f(x) = 3x^2 + 2x + 5$

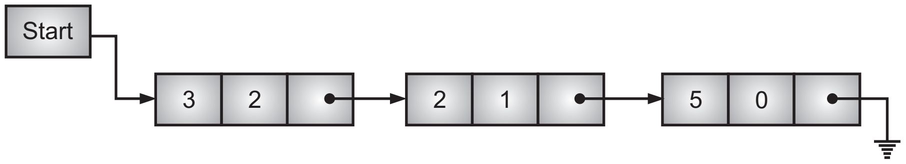


Fig. 3.48: Representation of Polynomial using Linked List

3.5.2 Polynomial Manipulations

- The manipulations of symbolic polynomials are an application of the list processing. The polynomial is, $A(X) = a_1 x^{em} + a_2x^{e(m-1)} + \ldots + a_nx^{e1}$, where $e_m > e_{m-1} > \ldots > e_1 \geq 0$ and $a_1 \ldots a_n$ are coefficients.
- A node of the linked list will represent each term. A node having three fields represents the coefficient, exponent of a term and a pointer to the next item.

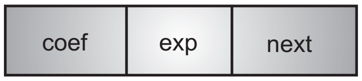


Fig. 3.49: A Node for Polynomial

- **Example:** The polynomial, say $A = 5 X^7 + 3X^5 + 4X^2 + 6$ is stored as,

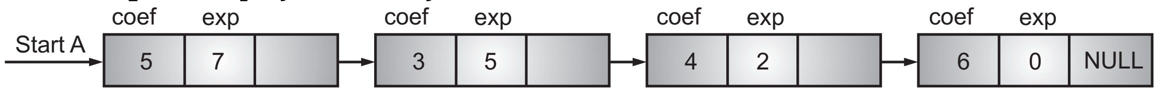


Fig. 3.50

- The polynomial $B = 3X^5 + X^2 - 2X$ will be represent as,

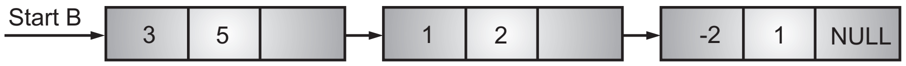


Fig. 3.51

- Thus, the 'C' declaration of the node is,

```
struct node
{
    int coef, exp;
    struct node * next;
}
```

- The singly linked list is used to represent polynomial. Create function is same as that of in singly linked list, only difference is that polynomial node has two data fields.

- Function for creation:

```
struct node * createpoly(struct node *head)
{
 int i,n,c;
 struct node *temp=head, *newnode=NULL, *last=NULL;
        printf("\n enter highest exponent for the polynomial: ");
 scanf("%d",&n);
        for(i=n;i>=0;i--)
 {
   printf("\nEnter coefficient for the term with exponent %d\t",i);
   scanf("%d", &c);
   if (c == 0)
       continue;
   newnode = (struct node *) malloc(sizeof(struct node));
           newnode->next=NULL;
   newnode->coef=c;
           newnode->exp=i;

   if(head==NULL)
               head=last=newnode;
           else
           {
               last->next=newnode;
               last=newnode;
           }
       }
   return head;
 }
```

3.5.3 Polynomial Evaluation

- The function traversal of singly linked list can be used with few modifications for polynomial evaluation.

```
/* function for evaluating polynomial */
double evaluate (struct node *head, double val)
{
    double result = 0;
    struct node *temp=head;
    while(temp != NULL)
    {
       result +=(temp -> coef) * pow(val, temp -> exp);
       temp = temp -> next;
    }
     return result;
}
/* pow() is library function use to compute a^b */
```

- To call above functions main() is as follows:

```
int main()
{
    int x;
    double ans;
    struct node *head=NULL;
    head = createpoly(head);
    printf("\n Enter value of variable x : ");
    scanf("%d",&x);
    ans = evaluate (head,x);
    printf("\n Ans of polynomial evaluation is = %f\n",ans);
}
```

3.5.4 Polynomial Addition

- Consider two polynomials,
  $A = 5x^7 + 6x^5 + 2x^2 + 2$
  $B = 6x^5 + x^2 - 2x$
- Two terms of a polynomial can be added if their exponents are same.
- The result after addition of above two polynomials is,
  $C = 5x^7 + 12x^5 + 3x^2 - 2x + 2$
- The two polynomials A and B are represented with two linked lists with pointers head1 and head2 pointing to first node of each polynomial.

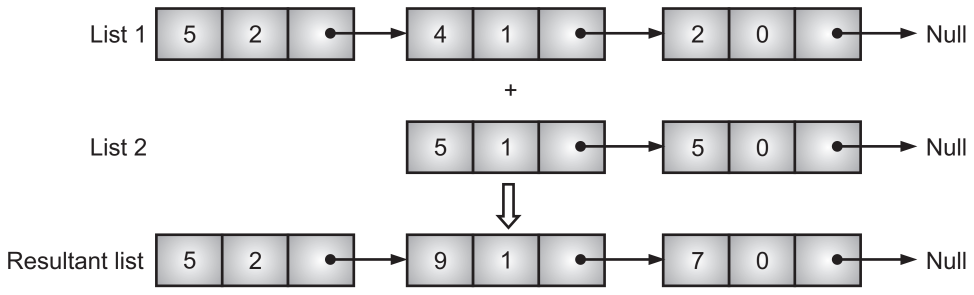


Fig. 3.52: Example of Addition of Polynomial

- The polynomial C will be represented with linked list with head3 pointer pointing to first node.

```
struct node *addPoly(struct node * head1, struct node *head2)
{
    struct node * p1=head1, *p2=head2, *newnode=NULL, *head3=NULL, *last=NULL;
    if(head1 == NULL || head2==NULL)
    {
        if(head1==NULL)
            head3= head2;
                else
            head3=head1;
    }
    else
    {
         while (p1 != NULL && p2 != NULL)
         {
            newnode=(struct node *) malloc(sizeof(struct node));
            newnode -> next = NULL;
            if (p1 -> exp == p2 -> exp)
            {
                newnode -> exp = p1 -> exp; // or p2->exp
               newnode -> coef = p1 -> coef + p2 -> coef;
               p1 = p1 -> next;
                p2 = p2 -> next;
            }
            else
            {
                if(p1 -> exp > p2 -> exp)
                {
                    newnode -> exp = p1 -> exp;
                   newnode -> coef = p1 -> coef;
                   p1 = p1 -> next;
                }
               else
                {
                   newnode -> exp = p2 -> exp;
                   newnode -> coef = p2 -> coef;
                   p2 = p2 -> next;
                }
            }
            if(head3==NULL)
               head3=last=newnode;
               else
            {
               last->next=newnode;
               last=newnode;
            }
         }//while
      while (p1 != NULL)
      {
         newnode=(struct node *) malloc(sizeof(struct node));
         newnode -> next = NULL;
         newnode -> exp = p1 -> exp;
         newnode -> coef = p1 -> coef;
         p1 = p1 -> next;
         last->next=newnode;
         last=newnode;
      }
      while (p2 != NULL)
      {
         newnode=(struct node *) malloc(sizeof(struct node));
         newnode -> next = NULL;
         newnode -> exp = p2 -> exp;
         newnode -> coef = p2 -> coef;
         p2 = p2 -> next;
         last->next=newnode;
         last=newnode;
      }
   }
   return head3;
   }
```

- **Function to Display polynomial:**

```
   void display(struct node *head)
   {
      struct node *temp=head;
      while(temp != NULL)
      {
         printf("(%dx^%d)+",temp->coef,temp->exp);
         temp=temp->next;
      }
   }
```

- **To call createpoly(), display() and addPoly() functions main() is as follows:**

```
int main()
{
    struct node *head1=NULL, *head2=NULL, *head3=NULL;
    head1 = createpoly(head1);
    printf("\n\nPolynomial1 is :\t");
    display(head1);
    head2 = createpoly(head2);
    printf("\n\nPolynomial2 is :\t");
    display(head2);
    head3 = addPoly(head1,head2);
    printf("\n\nAddition of Polynomials is :\t");
    display(head3);
}
```

3.6 GENERALIZED LINKED LIST [April 15, 17]

- In this section we will study GLL with its basic concepts.

3.6.1 Definition

- A generalized list, G, is a finite sequence of n ≥ 0 elements, $\alpha_1, \alpha_2, \ldots, \alpha_n$, which we write as G = $(\alpha_1, \alpha_2, \ldots \alpha_n)$ where α_i are either atom or list. The elements $\alpha_{i,}$ $1 \le i \le n$ which are not atoms are said to be the sublists of G, where n is the length of list.

  Example:

  1. A = () : Empty or null list
  2. B = (a, b()) : List of three element, first is a second is b and third is empty list.
  3. C = (a, (b, c), d): List contain 3 elements; first is a, second is list(b, c) and third is d, length = 3.
  4. D = (a, D) : A list of length 2 which is recursive as it includes itself and it can also be written as D = (a, (a, (a, - -))).

Pictorial Representation of example 3:

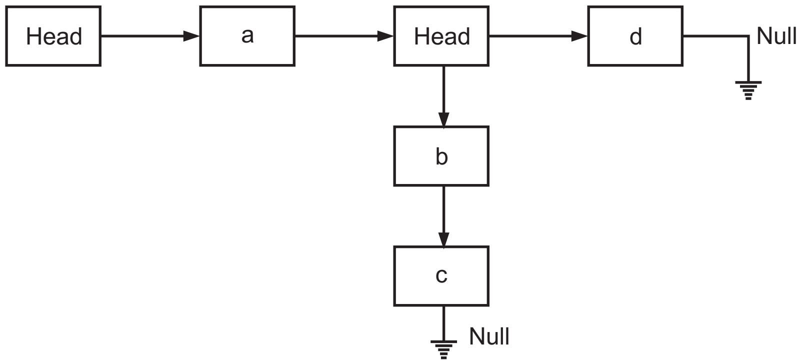


Fig. 3.53: Pictorial Representation of Example 3

3.6.2 Concept and Representation

- A generalized linked list A, is defined as, "a finite sequence of n>=0 elements, a_1, a_2, a_3, …, a_n, such that ai are either atoms or the list of atoms. Thus A = (a_1, a_2, a_3, … a_n)".
  Where, n is total number of nodes in the list.
- Now to represent such a list of atoms we will have certain assumptions about the node structure

Flag	Data	Down pointer	Next pointer

Flag = 1 means down pointer exists.
 = 0 means next pointer exists.
Data means the atom.
Down pointer is address of node which is down of the current node.
Next pointer is the address of the node which is attached as the next node.

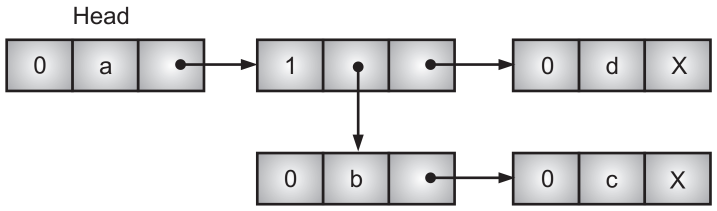
- With this typical node structure let us represent the generalized linked list for the above. Let us take few example to learn how actually the generalized linked list can be shown,

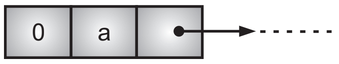
Typical 'C' structure of GLL:

```
typedef struct node
{
   chart c;                     /* Data */
   int ind;                     /* Flag */
   struct node *next,*down;     /* next and down pointer */
}gll;
```

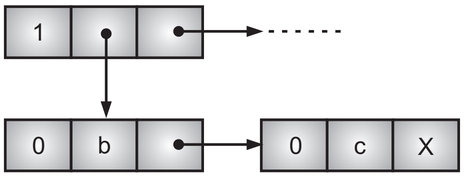
Example of GLL [List Representation]

1. (a, (b, c), d)



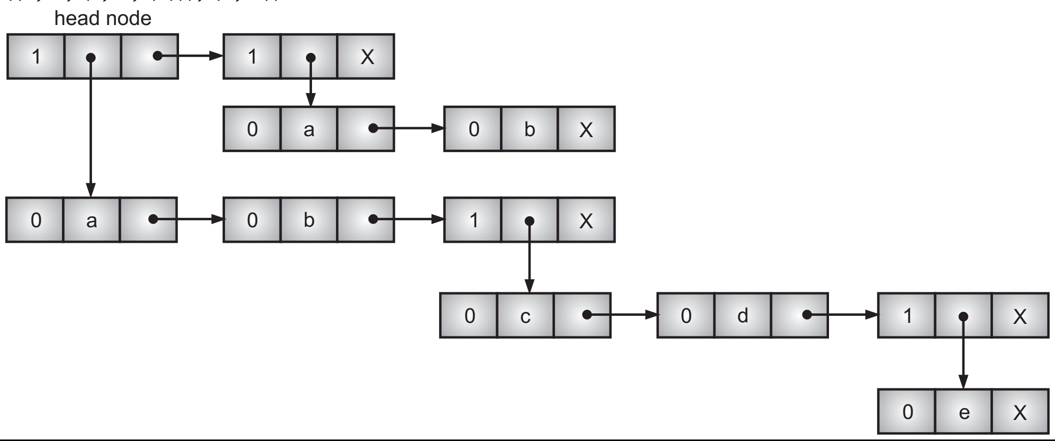
In above example the head node is 

In this case the first field is 0, it indicates that the second field is variable. If first field is 1 means the second field is a down pointer, means some list is starting.



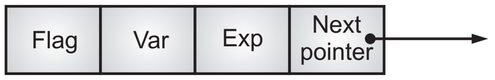
The above figure indicates (b, c). In this way the generalized linked list can be built. The X in the next pointer field indicates NULL value.

2. ((a, b, (c, d, (e))), (a, b))



3.6.3 Multiple-Variable Polynomial Representation Using Generalized Linked List (GLL)

- The above representation can be used to represent a multi-variable polynomial. Let us see the typical node structure of it.

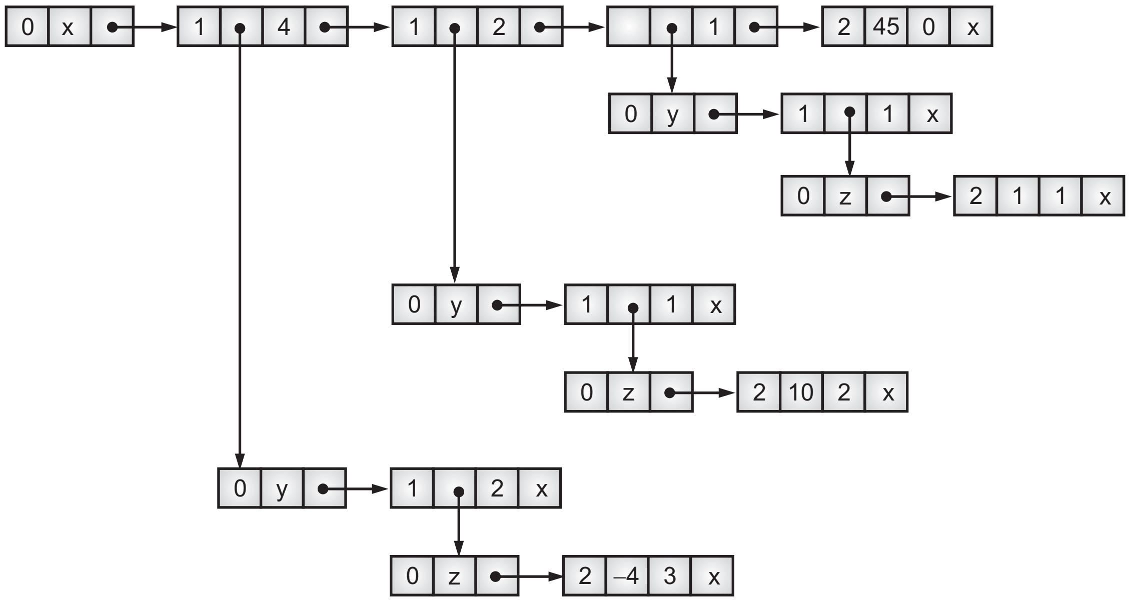


Let us see each field one by one Flag = 0 means variable is present.

Flag = 1 : Means down pointer is present.

Flag = 2 : Means coefficient and exponent is present.

Example: $4x^4 y^2 z^3 + 10x^2 + yz^2 + 7xyz + 45$



PRACTICE QUESTIONS

Q.I Multiple Choice Questions:

1. Which is a series of linearly arranged finite elements of same type?
   (a) list (b) array
   (c) stack (d) All of these
2. In a circularly linked list organization, insertion of a record involves the modification of
   (a) 0 pointer (b) 1 pointer
   (c) 2 pointers (d) 3 pointers
3. The concatenation of two lists is to be performed in O (1) time. Which of the following implementation of a list should be used?
   (a) Singly linked list (b) Doubly linked list
   (c) Circular doubly linked list (d) Array implementation of list.
4. Which of the following operations is performed more efficiently by doubly linked list than by linear linked list?
   (a) Deleting nodes whose location is given
   (b) Searching an unsorted list for a given item.
   (c) Inserting a node after the node with a given location.
   (d) Traversing the list to process each node.
5. Consider the linked list of n elements. What is the time taken to insert an element after an element pointed by some pointer?
   (a) O(1) (b) O(log2n)
   (c) O(n) (d) O(n log2n)
6. ______ is an ordered collection of data in which each element contains the location of the next element known as?
   (a) Array (b) Record
   (c) linked list (d) node
7. What does an empty linked list consist?
   (a) a variable (b) two nodes
   (c) data and a link (d) a null head pointer
8. In the last node of the circular linked list the link field contains?
   (a) Null (b) Pointer to next node
   (c) Pointer to first node (d) Pointer to data item
9. In a circular linked list:
   (a) Components are all linked together in some sequential manner.
   (b) There is no beginning and no end.
   (c) Components are arranged hierarchically.
   (d) Forward and backward traversal within the list is permitted.
10. What is the time complexity to count the number of elements in the linked list?
    (a) O(1) (b) O(n)
    (c) O(log n) (d) O(n2)
11. The way to represent a linear list is to expand each node to contain a link or pointer to the next node this representation is called as _____.
    (a) singly linked list (b) doubly linked list
    (c) circular linked list (d) none of these

12. Which list is a variation of Linked list in which the first element points to the last element and the last element points to the first element?
    (a) singly linked list (b) doubly linked list
    (c) circular linked list (d) none of these
13. The list with no nodes on it is called as _____.
    (a) singly list (b) null list
    (c) circular linked list (d) none of these
14. Which operation on a list means to visit every element or node in the list beginning from first to last?
    (a) insertion (b) deletion
    (c) traversing (d) none of these
15. Which is used for static implementation of linked list?
    (a) arrays (b) pointers
    (c) stacks (d) none of these
16. Which operation is used to print each and every node's information?
    (a) create (b) delete
    (c) display (d) search

Answers

1. (a)	2. (c)	3. (c)	4. (a)	5. (a)	6. (c)	7. (d)	8. (c)	9. (b)	10. (b)
11. (a)	12. (c)	13. (b)	14. (c)	15. (a)	16. (c)				

Q.II Fill in the Blanks:

1. Linked list data structure is used for sorting data in the term of a ______.
2. Linked list is called ______ data structure where amount of memory required can be changed during its use. Linked list can grow or shrink during the execution of a program.
3. Linked list is a linear collection of ______ data items.
4. Single linked list is a ______ of elements in which every element has link to its next element in the sequence.
5. ______ linked list is also called as two-way linked list.
6. A ______ linked list is a dynamic data structure. It may grow or shrink growing or shrinking depends on the operations made on list.
7. A linked list is a ______ data structure, in which the elements are not stored at contiguous memory locations.
8. A ______ is a collection of two sub-elements or parts. A data part that stores the element and a next part that stores the link to the next node.
9. ______ is a process to search the first occurrence of the given item in the list and to return the address of the node.
10. Insertion is a process to ______ a new element in the linked list.
11. A linked list in which the pointer of the last node points to the first node of the list is called ______ linked list.
12. The pointer list is called an ______ pointer to the list (because it is not contained within a list node), whereas the other pointers in the list are internal pointers (because they are contained within list nodes).
13. A doubly linked ______ list means the last node points to the first node and there are two links (one for pointing to the predecessor node, another for pointing to the next node) between the nodes of the linked list.
14. Dynamic implementation of linked list uses ______.

Answers

1. list	2. dynamic	3. similar	4. sequence	5. doubly
6. singly	7. linear	8. node	9. searching	10. add
11. circular	12. external	13. circular	14. pointers	

Q.III State True or False:

1. A linked list is a linear data structure.
2. Static implementation of linked list uses pointers.
3. A linked list is a linear collection of data items called nodes, where the linear order is given by means of pointers.
4. In the single link list each node has two fields one for data and other is for link reference of the next node.
5. To create a linked list, we will create the nodes one by one and add them to the end of the list.
6. A circular linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields (two link fields (references to the previous and to the next node in the sequence of nodes) and one data field).
7. Linked List is a dynamic data structure, which can be used with a data collection that grows or shrinks during program execution.
8. The sort operation helps to find an element in the linked list.
9. Deletion is a process to remove an existing node from the linked list.
10. Doubly linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.
11. Generalized list made up of number of components, some are the data elements (atoms) and others are generalized list (list includes list).
12. A linked list in which the pointer of the last node points to the first node of the list is called circular linked list.
13. A doubly linked list contains a pointer to the next node as well as the previous node. This ensures that the list can be traversed in both directions.
14. A singly linked circular list means the last node points to the first node and there is only one link between the nodes of the linked list.
15. A linked list is a dynamic data structure where each element (called a node) is made up of two items - the data and a reference (or pointer) which points to the next node.

Answers

1. (T)	2. (F)	3. (T)	4. (T)	5. (T)	6. (F)	7. (T)	8. (F)
9. (T)	10. (T)	11. (T)	12. (T)	13. (T)	14. (T)	15. (T)	

Q.IV Answer the following Questions:

(A) Short Answer Questions:

1. What is list?
2. What is linked list?
3. Define the term node.
4. List types of linked list.

5. What are the operations performed by linked list?
6. Define doubly linked list.
7. What is circular linked list?
8. What is polynomial?
9. Define generalized linked list.
10. How to search a node in linked list?
11. Define null list.
12. What is meant by external pointer?

(B) Long Answer Questions:

1. Define linked list. How to create it? Explain with example.
2. What are the fields of a node in a linked list? Describe with diagram.
3. Give node structure for doubly linked list. Write advantages of doubly linked list over singly list.
4. Describe different types of linked list. Also state their advantages and disadvantages.
5. Explain any two operations of singly linked list.
6. Write a 'C' function to add a node at the beginning of singly linked list.
7. Define doubly linked list? How to create it? Explain with example.
8. How to search element in singly linked list? Explain with example.
9. Write a short note on circular linked list.
10. Create two linked lists to represent the following polynomials:
    $3x^2y + 9xy^3 + 15xy + 3$
    $13\ x^3y^2 + 7x^2y + 22\ xy + 9y^3$
    Write a function add() to add these polynomials and print the resulting linked list.
11. With the help of example describe how to create a generalized linked list.
12. Write a 'C' function to delete an element from singular linked list.
13. How to insert and delete elements in circular linked list?
14. Write a 'C' function to delete last node from linked list.
15. Write a C function to insert node in circular doubly linked list.
16. Write an algorithm to traverse a singly linked list.
17. Write an algorithm and program to reversing a singly linked list.
18. Write a 'C' function to delete node from circular linked list at any position.
19. Write an algorithm to merging a singly linked list.
20. Write a program to sorting a singly linked list.
21. Write a 'C' function to insert a node in circular doubly linked list.
22. Write an algorithm for traversing a linked list and also give its function.
23. List the applications of linked list and explain any one in detail.
24. Write a program that reads the name, age and salary of 10 persons and maintains them in a linked list sorted by name.

UNIVERSITY QUESTIONS AND ANSWERS

April 2015

1. "The elements of a linked list are stored sequentially". State true or false. **[1 M]**

Ans. Refer to Section 3.1.

2. Write a 'C' function to delete an odd position element from a doubly linked list. **[5 M]**

Ans. Refer to Section 3.4.2.

3. Write a node structure for generalized linked list. Draw GLL for ((a, (b, (c)), d, e). **[3 M]**

Ans. Refer to Section 3.6.

April 2016

1. 'Linked list can only be traversed sequentially'. State True/False. **[1 M]**

Ans. Refer to Section 3.4.1.4.

2. Write a 'C' function to perform addition of elements at event position in a doubly link list of integers. **[5 M]**

Ans. Refer to Section 3.4.2.

October 2016

1. Write node structure of doubly linked list. **[1 M]**

Ans. Refer to Section 3.4.2.

2. What are applications of Linked List? **[5 M]**

Ans. Refer to Section 3.5.

3. Write a 'C' function to create a Doubly Linked List. **[5 M]**

Ans. Refer to Section 3.4.2.1.

April 2017

1. What is circular linked list? **[1 M]**

Ans. Refer to Section 3.4.3.

2. Write a 'C' function to reverse singly linked list. **[5 M]**

Ans. Refer to Section 3.4.1.5, Point (1).

3. What is generalized linked list? Explain with example. **[3 M]**

Ans. Refer to Section 3.6.

October 2017

1. Define node structure of a singly linked list. **[1 M]**

Ans. Refer to Section 3.3, Point (1).

April 2018

1. Write node structure for a Doubly Circular Linked List. **[1 M]**

Ans. Refer to Section 3.4.2.

2. "A Linked List can only be traversal sequentially". State True/False. **[1 M]**

Ans. Refer to Section 3.4.1.4.

3. Write a 'C' function to display even element (data) in a single linked list of integer. **[5 M]**

Ans. Refer to Section 3.4.1.

■■■

CHAPTER 4

Stack

Objectives ...

- To study Basic Concepts in Stack
- To learn Static and Link Implementations of Stacks
- To study different Operations and Applications of Stacks
- To study Expression Evaluation and Conversions (Prefix, Infix and Postfix)

4.0 INTRODUCTION

- A stack is an ordered collection of items in which insertion and deletions are allowed only at one end called top of the stack. Stack is most essential linear data structure.
- A stack is a non-primitive linear data structure. As all the deletion and insertion in a stack is done from top of the stack, the last added element will be the first to be removed from the stack. That is the reason why stack is also called Last-In-First-Out (LIFO) type of list.
- In Fig. 4.1 illustrates the stack of books that we keep in the order. Whenever, we want to remove a book the removal operation is made from the top or new books can be added at the TOP.

Fig. 4.1: Stack of Books

4.1 DEFINITION OF STACK

- A stack is an ordered collection of homogeneous data element where the insertion and deletion operations take place at only one end called as TOP of the stack.
- Stack is a set of elements in a Last-In-Fist-Out (LIFO) technique. The insertion or adding the element onto the stack is called 'push'. The deletion or removing the element from the stack is called 'pop'.
- The position of the stack where the operations are performed is called Top Of Stack (TOS) of the stack.
- Stack can also be defined as Abstract Data Type (ADT).
- **Example:** In Fig. 4.2 shows the stack of numbers 1, 2, 3, 4, 5 where element 1 is at the bottom of the stack and element 5 is at the top of the stack.

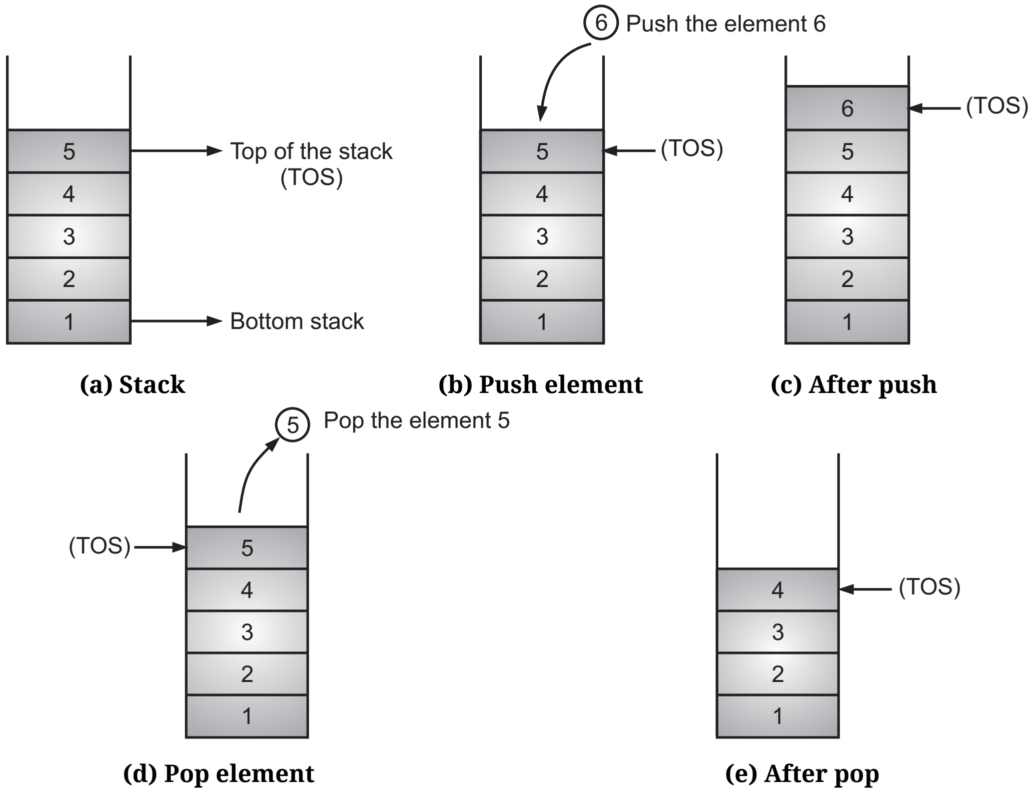
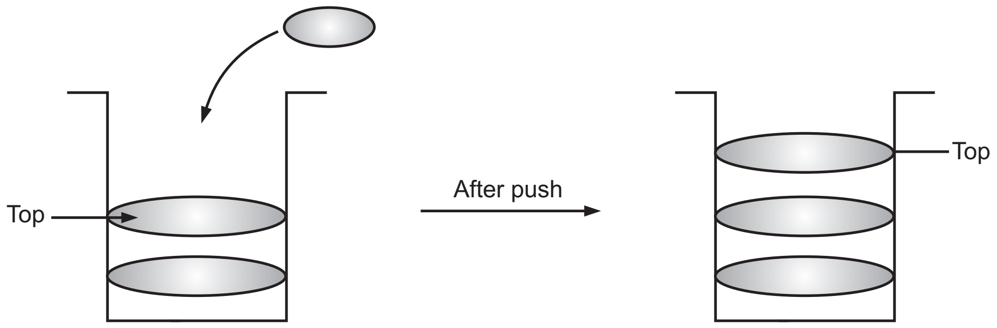


Fig. 4.2: Stack with push and pop Operations

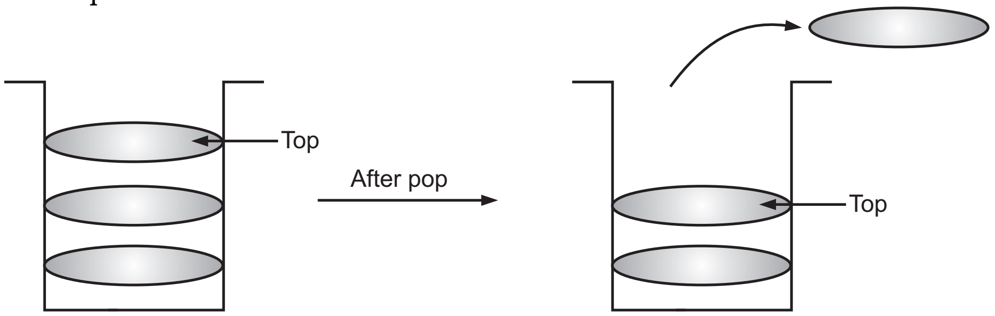
4.2 OPERATIONS ON STACK [Oct. 17; April 18]

- Various operations on stack data structure are given below:
  1. **init(s):** To initialize stack 's'.
  2. **push(s, x):** push operation adds an element on the top of the stack. Top always points to the element which is recently pushed into the stack.

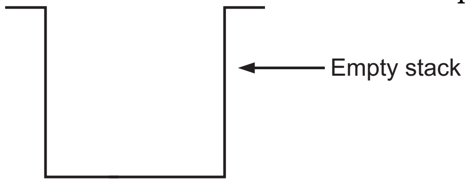


We must ensure whether there is a space to accommodate new element before every push operation.

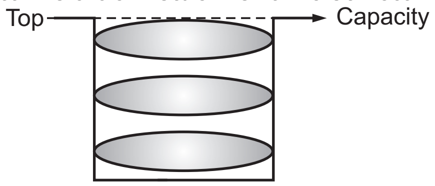
3. **pop(s):** pop removes an element, which is at the top of the stack. After removing a topmost element, the size of stack is reduce by 1 and the element below it becomes the top.



4. **IsEmpty(s):** Returns true if stack is empty else return false. When there is no element present on the stack then stack is said to be empty.



5. **IsFull(s):** When top of the stack reached to stack capacity then stack is said to be full. This function returns true if stack is full else return false.



6. **peek():** This operation on stack get the top data element of the stack, without removing it. A peek() operation may give access to the top without modifying the stack.

Note: The time complexity of push, pop, isEmpty, Isfull in the worst-case run-time complexity can be O(1). However, with respect to the representations (static array and a reasonable linked list) these operations take constant time. So, the size and isEmpty are constant-time operations. push and pop are also O(1) because they only work with one end of the data structure - the top of the stack.

4.3 REPRESENTATIONS OF STACK

- Stack data structure can be implemented using static and dynamic data structure.
  1. Static implementation using arrays, and
  2. Dynamic implementation using linked list.

4.3.1 Implementation of Stack using Array (Static Implementation)

- An array is used to store ordered list of elements. Stack is an ordered list of elements. Hence, it is easy to manage a stack when represented using an array.

- There are limitations imposed on the stack because array is static data structure. Once, we declare the size it cannot be modified at runtime. Therefore, we specify maximum size of array which leads to poor memory utilization.
- The Fig. 4.3 shows the array implementation of stack. When stack is implemented using arrays. One of the either of the side of array can be considered as the top (upper) side and other side as bottom (lower) side. Let stack [n] be the one dimensional array; so stack size = n and max = n elements.

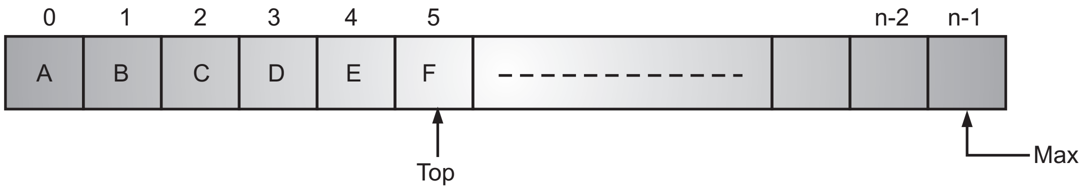


Fig. 4.3: Stack using Array

- Initially, elements are stored in the stack from the first location. The first element is stored in 0th location i.e. stack[0], the 2nd element is stored in 1st location at stack[1] and so on; nth element is stored at stack[n – 1].
- The top pointer points to the top of the stack, Here, top = 5. This implementation is suitable only when we exactly know the number of elements to be stored; otherwise memory is wastage.
- The following operations are performed on the stack:
  1. Creating a stack.
  2. Checking stack empty or not.
  3. Checking stack full or not.
  4. Insert (push) an element in the stack.
  5. Delete (pop) an element from the stack.
  6. Access the top element.
  7. Status to identify the present position of stack.
- Let us discuss the above operations on the stack ADT. The structure stack is defined as follows:

```
struct stack
{
     int data[10];
     int top;
};
```

1. init(Stack): To initialize stack S.

- In this function top is initialized to -1 to indicate the stack is empty.

```
struct stack s1;
s1.top = – 1;
```

- Here, we created stack named s1 in which we can insert at the most 10 integer values.

2. IsEmpty(Stack):

- This operation takes stack as an argument, checks whether it is empty or not and returns 1 if empty otherwise 0.

- To check whether stack is empty or not, we compare value of 'top' with value '–1'.

```
IsEmpty (stack)
   if (top == –1)
       return 1;
   else
      return 0;
```

3. IsFull(Stack):

- This operation takes stack as an argument, checks whether it is full or not and returns 1 if full otherwise 0.
- To check whether stack is empty or not, we compare value of 'top' with value '–1'.

```
IsFull (stack)
   if (top == size-1)
       return 1;
   else
      return 0;
```

4. push(Stack, element):

- The element is pushed into the stack only if the stack is not full. After push operation, the top is incremented by 1.

```
if (top = = n – 1)
   printf ("stack overflow \n");
else
   {
    top++;                        }  ⇒ stack[++top] = element;
    stack[top] = element;         }
   }
```

5. pop(Stack):

- An element can be popped from the stack only if stack is not empty.
- In such a case the topmost element is removed first and then top is decremented.

```
if (top ! = –1)
   return(stack[top--]);
else
   printf("stack underflow\n");
```

- All the operations above can be defined in terms of functions create(), IsEmpty(), Top(), push(), pop(), IsFull(). It is better if we logically group the stack. The structure stack has been defining as follows:

```
#define max 20
struct stack
   {
    int data[max];
    int top;
   };
struct stack p, q, r;
```

- This defines three stacks p, q, r.

- The pictorial representation of push and pop operations for the following function calls.

```
push(&p,15);
push(&p,10);
pop(&p);
push(&p,5);
pop(&p);
pop(&p);
pop(&p);
```

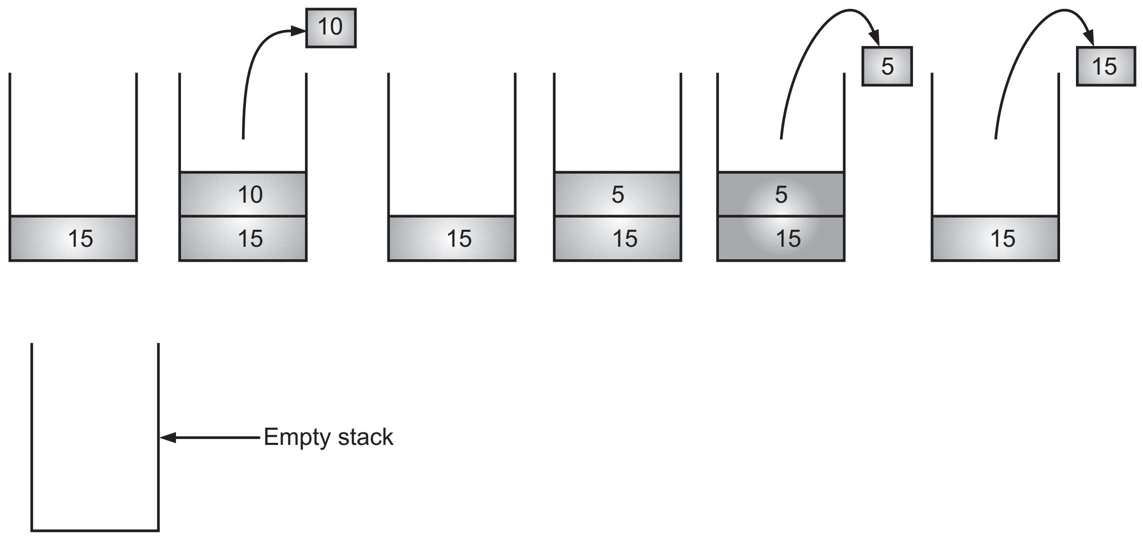


Fig. 4.4: Operations on Stack

Program 4.1: Implementation of static stack (using arrays).

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#define MAX 5    //Maximum number of elements that can be stored
int top=-1, stack[MAX];
void push();
void pop();
void display();
int main()
{
    int ch;
    setbuf(stdout, NULL); // turn off buffering of stdout
    printf("\n*** Stack Menu ***");
    printf("\n1.Push\n2.Pop\n3.Display\n4.Exit");
    while(1)    //infinite loop
    {
        printf("\nEnter your choice(1-4):");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: push();
                    break;
            case 2: pop();
                    break;
            case 3: display();
                    break;
            case 4: exit(0);
            default: printf("\nWrong Choice!!");
        }
    }
    return 0;
}
void push()
{
    int val;

    if(top==MAX-1)
           printf("\nStack is full!!");
    else
    {
        printf("\nEnter element to push:");
        scanf("%d",&val);
        top=top+1;
        stack[top]=val;       //element inserted in stack at top position
    }
}
void pop()
{
    if(top==-1)
      printf("\nStack is Empty!!");
    else
    {
        printf("\nDeleted element is %d",stack[top]); //delete top element
        top=top-1;
    }
}
```

```
void display()
{
    int i;
    if(top==-1)
            printf("\nStack is Empty!!");
    else
    {
        printf("\n**** Elements in Stack*****.\n");
        for(i=top;i>=0;i--)
            printf("%d\n",stack[i]);
    }
}
```

Output:

```
*** Stack Menu ***
1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):1
Enter element to push:10
Enter your choice(1-4):1
Enter element to push:20
Enter your choice(1-4):1
Enter element to push:30
Enter your choice(1-4):3
**** Elements in Stack*****.
30
20
10
Enter your choice(1-4):2
Deleted element is 30
Enter your choice(1-4):
```

Program 4.2: Implementation of static stack with operations like push, pop etc.

```
/* "ststack.h"*/ /*header file*/
#define MAX 10
struct stack
{
   int a[MAX];
   int top;
};
```

```
void inits(struct stack*);
void pushs(struct stack*,int);
int pops(struct stack*);
int isfulls(struct stack*);
int isemptys(struct stack*);
int peeks(struct stack*);
/*program*/
#include <stdio.h>
#include "ststack.h"
int main(void)
{
   struct stack s;
   int opt,x,n,e;
   inits(&s);
   do
   {
      printf("1.Push\n2.Pop\n3.Peek\n4.isEmpty\n5.isfull\n6.Exit\n");
      scanf("%d",&opt);
      switch(opt)
      {
      Case 1: printf("enter element to push\n");
              scanf("%d",&x);
              pushs(&s,x);
              break;
      Case 2: e=pops(&s);
              if(e!=0)
              {
                 printf("deleted element is %d\n",e);
              }
              else
              printf("stack is underflow\n");
              break;
      Case 3: e=peeks(&s);
              printf("element on top is %d\n",e);
              break;
      Case 4: e=isemptys(&s);
              if(e==1)
              printf("list is empty\n");
              else
              printf("list is not empty\n");
              break;
```

```
        Case 5: e=isfulls(&s);
                if(e==1)
                printf("list is full\n");
                else
                printf("list is not full\n");
                break;
      }
   }while(opt!=6);
   return 0;
   }
   void inits(struct stack *ps)
   {
      ps->top=-1;
   }
   void pushs(struct stack *ps,int x)
   {
      if((isfulls(ps))==1)
      {
         printf("stack overflow\n");
         return;
      }
      else
      {
         ps->a[++ps->top]=x;
      }
   }
   int pops(struct stack *ps)
   {
      if((isemptys(ps))==1)
          return 0;
      else
      {
         return ps->a[ps->top--];
      }
   }
   int peeks(struct stack *ps)
   {
      return ps->a[ps->top];
   }
```

```
int isemptys(struct stack *ps)
{
   if(ps->top==-1)
     return 1;
   else
     return 0;
}
int isfulls(struct stack *ps)
{
   if(ps->top==MAX-1)
        return 1;
   else
        return 0;
}
```

Output:

```
1.Push
2.Pop
3.Peek
4.isEmpty
5.isfull
6.Exit
4
list is empty
1.Push
2.Pop
3.Peek
4.isEmpty
5.isfull
6.Exit
5
list is not full
1.Push
2.Pop
3.Peek
4.isEmpty
5.isfull
6.Exit
1
enter element to push
4
```

```
1.Push
2.Pop
3.Peek
4.isEmpty
5.isfull
6.Exit
1
enter element to push
7
1.Push
2.Pop
3.Peek
4.isEmpty
5.isfull
6.Exit
3
element on top is 7
1.Push
2.Pop
3.Peek
4.isEmpty
5.isfull
6.Exit
2
deleted element is 7
1.Push
2.Pop
3.Peek
4.isEmpty
5.isfull
6.Exit
3
element on top is 4
1.Push
2.Pop
3.Peek
4.isEmpty
5.isfull
6.Exit
```

4.3.2 Implementation of Stack using Linked List (Dynamic Implementation)

- Array implementation discussed in above section has certain limitations such as, stack cannot grow or shrink during execution of program.
- Therefore, either much of the space is wasted, if not used, or there is shortage of space if needed.
- In linked representation, we can add or remove the element from any position of the list. But stack allow us to add and remove elements from only one end i.e. Top to the stack.
- In linked representation, we need not have prior knowledge of the number of elements.
- When elements are removed, the memory is freed. Each element of the stack will be represented as a node of the list. We can add or remove node only at one end.
- The first node is considered to be the top of the stack. The last node is bottom of stack and set to null.
- **Example:** The Fig. 4.5 shows the pictorial representation of stack in link list containing three elements (C, B, A).
- The structure defined to represent dynamic stack of integers is as follows:

```
#define MAX 20
struct node
{
   int data[MAX];
   struct node *next;
    };
```

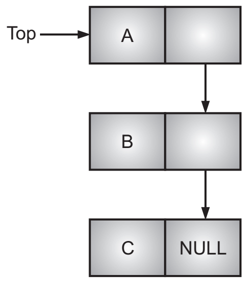
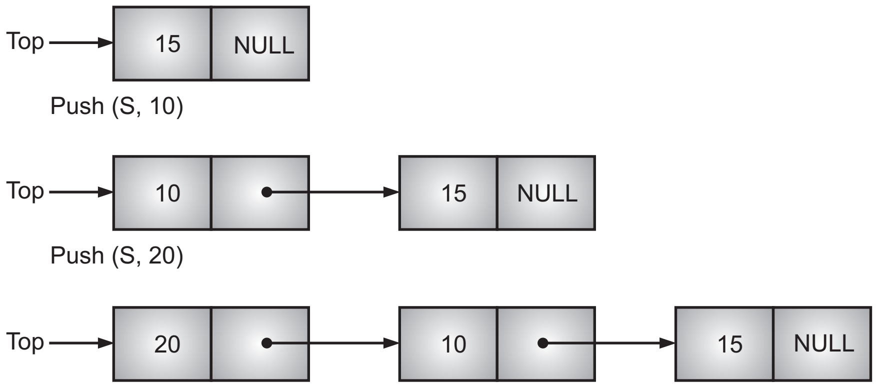


Fig. 4.5: Stack as a Linked List Data Structure

Operations on Stack with respect to Link List Representation:

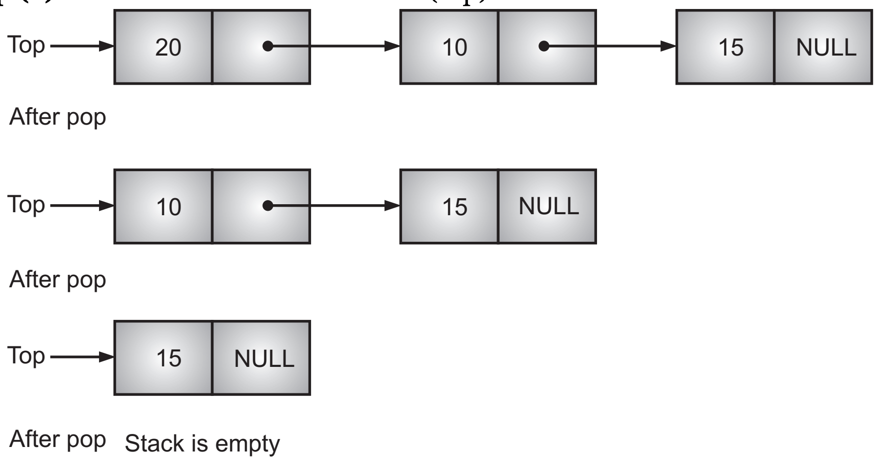
- Various operations on stack are listed below:
  1. **create:** `struct node *s, *TOP=NULL;`
  2. **push:** It inserts an element onto top of the stack, e.g. push(S, 15).



- Here, stack grows at the 'top' end and the element is always pushed at top.
  The function can be written as,

```
void push (struct node *top, int item)
{
   struct node *p=NULL;
   p = (struct node *)malloc(sizeof(struct node));
   if (p==NULL)
         printf("Memory allocation failed\n");
   else
   {
      p→data=item;
      p→next=top;
            top=p;
   }
}
```

3. **pop (S):** It removes the first element (top).



Here, all operations are same as that of linked list. Only the difference is that new element is pushed only at the top and pop from top.

Program 4.3: Implementation of dynamic stack (using linked list).

```
#include<stdio.h>
struct node
{
   int data;
   struct node *next;
};
/* add the elements */
void push (struct node *top, int item)
{
   struct node *p=NULL;
   p = (struct node *)malloc(sizeof(struct node));
```

```
    if (p==NULL)
         printf(“Memory allocation failed\n”);
    else
       {
          p→data=item;
          p→next=top;
          top=p;
       }
}
/* pop item from the stack */
int pop(struct node *top)
{
    struct node *p=NULL;
    int x;
    if(top==NULL)
    {
       printf("stack is empty \n");
       return -1;
    }
    p=top;
    x=top→data;
    top=top→next;
    free(p);
    return x;
}
void main()
{
    struct node *top=NULL;
       push(top,15);
       push(top,10);
       printf("%d\n", pop(top))
       push(top,20)
       printf("%d\n", pop(top));
       printf("%d\n", pop(top));
       pop(top);
}
```

Output:

```
10
20
15
Stack is empty
```

Program 4.4: Implementation of dynamic stack with all operations.

```
/* dystack.h*/ /* header file*/
struct stack
{
   int info;
   struct stack *next;
};
void init(struct stack**);
void push(struct stack**,int);
struct stack * pop(struct stack**);
void peek(struct stack*);
void isEmpty(struct stack*);
void display(struct stack);
/* program */
#include <stdio.h>
#include <stdlib.h>
#include "dystack.h"
int main(void)
{
   struct stack *s,*e;
   int opt,x,n;
   init(&s);
   do
   {
      printf("1.Push\n2.Pop\n3.Peek\n4.isEmpty\n5.Exit\n");
      scanf("%d",&opt);
      switch(opt)
      {
      case 1:  printf("enter element to push\n");
               scanf("%d",&x);
               push(&s,x);
               break;
      case 2:  e=pop(&s);
               if(e!=NULL)
               {
                  free(e);
                  peek(s);
               }
               else
               printf("stack is underflow\n");
               break;
```

```
        case 3: peek(s);
                break;
        case 4: isEmpty(s);
                break;
    }
}
while(opt!=5);
return 0;
}
void init(struct stack **s)
{
    *s=NULL;
}
void push(struct stack **s,int x)
{
    struct stack *p;
    p=(struct stack *)malloc(sizeof(struct stack));
    p->info=x;p->next=NULL;
    if(*s==NULL){ *s=p; }
    else
    {
        p->next=*s;
        *s=p;
    }
}
display(struct node **l)
{
    struct node *p;
    if(*l==NULL) { printf("List is Empty\n");}
    else
    {
        p=*l;
        while(p!=NULL)
        {
            printf("%d\t",p->info);
            p=p->next;
        }
    }
}
```

```
struct stack * pop(struct stack **s)
{
   struct stack *p;
   if(*s==NULL) return NULL;
   else
   {
      p=*s;
      *s=(*s)->next;
      p->next=NULL;
      return p;
   }
}
void peek(struct stack *s)
{
   if(s==NULL) printf("stack underflow\n");
   else        printf("%d\n",s->info);
}
void isEmpty(struct stack *s)
{
   if(s==NULL) printf("List is Empty\n");
   else        printf("List is not Empty\n");
}
```

Output:

```
1.Push
2.Pop
3.Peek
4.isEmpty
5.Exit
4
List is Empty
1.Push
2.Pop
3.Peek
4.isEmpty
5.Exit
1
enter element to push
4
```

```
1.Push
2.Pop
3.Peek
4.isEmpty
5.Exit
1
enter element to push
9
1.Push
2.Pop
3.Peek
4.isEmpty
5.Exit
3
9
1.Push
2.Pop
3.Peek
4.isEmpty
5.Exit
2
4
1.Push
2.Pop
3.Peek
4.isEmpty
5.Exit
3
4
1.Push
2.Pop
3.Peek
4.isEmpty
5.Exit
4
List is not Empty
1.Push
2.Pop
3.Peek
4.isEmpty
5.Exit
```

4.3.3 Comparison between Static and Dynamic Implementations of Stack

- Following table lists the difference between two different representations of stack in array and linked list.

Sr. No.	Stack Representation using Array	Stack Representation using Linked List
1.	It is called static implementation of stack.	It is called dynamic implementation of stack.
2.	This method does not provide flexibility because as the size of stack has to be declared then there will be no charge in size after program design.	This method does provide flexibility as the size of list can grow infinitely.
3.	Memory space is wasted (because of storing less number of elements in the array than the maximum size of array).	Memory space is not wasted, because memory space is allocated only when the user wants to push an element into the stack.
4.	Insertion and deletion operation in a stack using an array involve more data movement.	The first advantage of this method is that insertion and deletion operations do not involve more data movement, (just the change of pointers).
5.	Less space is used for each data element (no pointer is used).	More space is used for each node (because pointer fields are involved).

4.4 APPLICATIONS OF STACK [Oct. 17]

- Stack data structure is used in many applications. Some of them are listed below:
  1. **Expression Conversion**: An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.
  2. **Expression Evaluation:** A stack can be used to evaluate prefix, postfix and infix expression.
  3. **Syntax Parsing**: Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.
  4. **String Reversal:** Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack. [April 17]
  5. **Parenthesis Checking:** Stack is used to check the proper opening and closing of parenthesis.
  6. **Backtracking**: Backtracking is one of the algorithm designing technique. For that purpose, we dive into some way, if that way is not efficient, we come back to the previous state and go into some other paths. To get back from current state, we need to store the previous state. For that purpose, we need stack. Some examples of backtracking are find the solution for Knight Tour problem or N-Queen Problem etc.
  7. **Function Call**: Stack is used to keep information about the active functions or subroutines.

4.4.1 Function Call and Recursion [April 15, 17]

- A function is a self-contained block of code that executes a certain task.

Function Call:

- In order to use function in the program we use function call to access the function. The function that calls the function is referred to as calling function.

Processing of Function (Subprogram) Calls:

- Stacks are frequently used to keep track of the order of execution of the functions called by a main program.
- The stacks only have pointers to the main program and the functions and not the function themselves, which help in maintaining the order of functions required for completion of processing. Fig. 4.6 shows this.

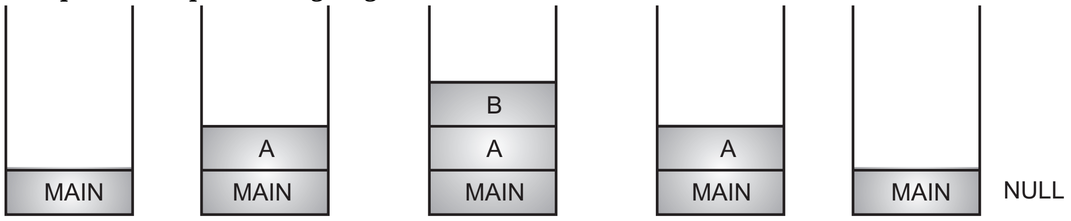


Fig. 4.6: Function Calls during Program Execution

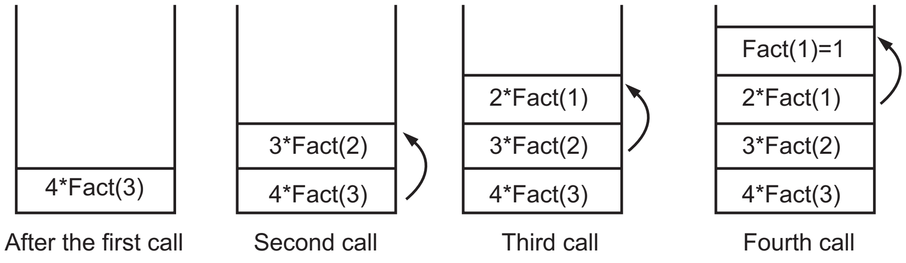
- Here, initially MAIN is pushed in the stack. At a certain point during execution MAIN calls A, which in turn at some stage calls B.
- Now A can start execution when B has been executed, after it A is completely executed and then MAIN is completely executed. After execution of any function the stack is POPped ultimately leaving an empty or a NULL stack.

Recursion:

- A procedure is called recursive, if the procedure is defined by itself. For example, the problem of factorial can be solved using a recursive procedure.
- Recursion is useful in developing algorithms for specific problems. The stacks are used to implement recursive procedures.
- In the recursion, the procedure is called itself directly or indirectly. Directly means procedure called itself repeatedly. Indirect means a procedure calls another procedure.
- Hence, the procedures or functions are executed repeatedly, every time a new value is passed to the recursive function till the condition is satisfied.
- If the recursive procedure call itself, then current values of parameters must be saved, since they will be used again when program is reactivated.
- Consider, the example of finding factorial of given number. We write this function recursively as,

```
int fact (int x)
{
  if (x = = 0)
   return(1);
  else return (x * fact(x –1));
}
```

- Let see how factorial function works by considering x=4. When function call occur, previous variables gets stored in stack.



- Returning values from base case to caller function.

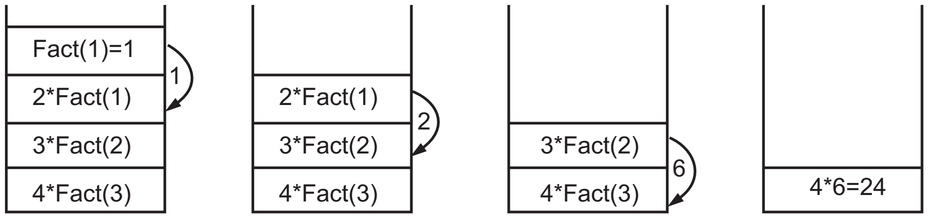


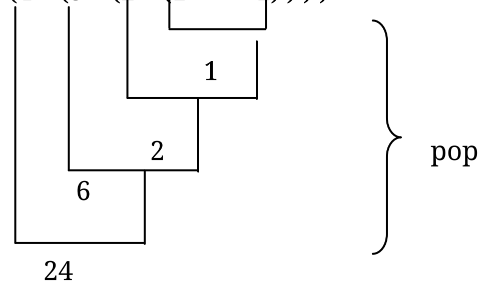
Fig. 4.7: Factorial of Number using Recursion

- It is same as:

  (4 * (3 * (2 * fact (1))))

  (4 * (3 * (2 * (1 * fact(0))))) push

  (4 * (3 * (2 * (1 * 1))))



Rules of Recursion:

1. There must be terminating condition to stop recursion.
2. When function is called recursively the copy of the stack is maintaining.

Memory Allocation to different Function Calls in Recursion:

- When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, so the memory for a called function is allocated on top of the memory allocated for calling the function.
- Remember, a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Program 4.5: Program to calculate factorial of a given number using recursion in C.

```
#include<stdio.h>
#include<conio.h>
int fact(int);
int main()
{
    int num,f;
    setbuf(stdout, NULL);
    printf("Enter any Number\n");
    scanf("%d",&num);
    f=fact(num);
    printf("Factorial of a number %d is %d",num,f);
    return 0;
}
int fact(int f)
{
    if (f==1)
        return f;
    else
        return f * fact(f-1);
}
```

Output:

```
Enter any Number : 5
Factorial of a number 5 is 120
```

4.4.2 String Reversal [April 15, 16]

- A simple application of stacks is reversing strings. This can be achieved very easily by reading the input string character by character and push that onto stack, till end of the string is reached.
- Once, all the characters of the string are pushed onto the stack, they are popped one by one. Since, the character last pushed in comes out first, subsequent POP operations result in reversal of the string.

Algorithm for String Reversal:

Let str[] is character array/string to be reversed, stack[] is character to array to store string,

```
1. Set i = 0
2. While(i < length_of_str)
       PUSH str[i] onto the stack[i]
       Set i + 1
   End While
```

```
3. Set i = TOP
4. While(i >= 0)
       POP the TOP element of the stack[i] and store it in str[i]
       Set i = i - 1
   End While
5. Print "The reversed string is: ", str
6. Exit
```

- Fig. 4.8 shows an example of reverse a string using stack.

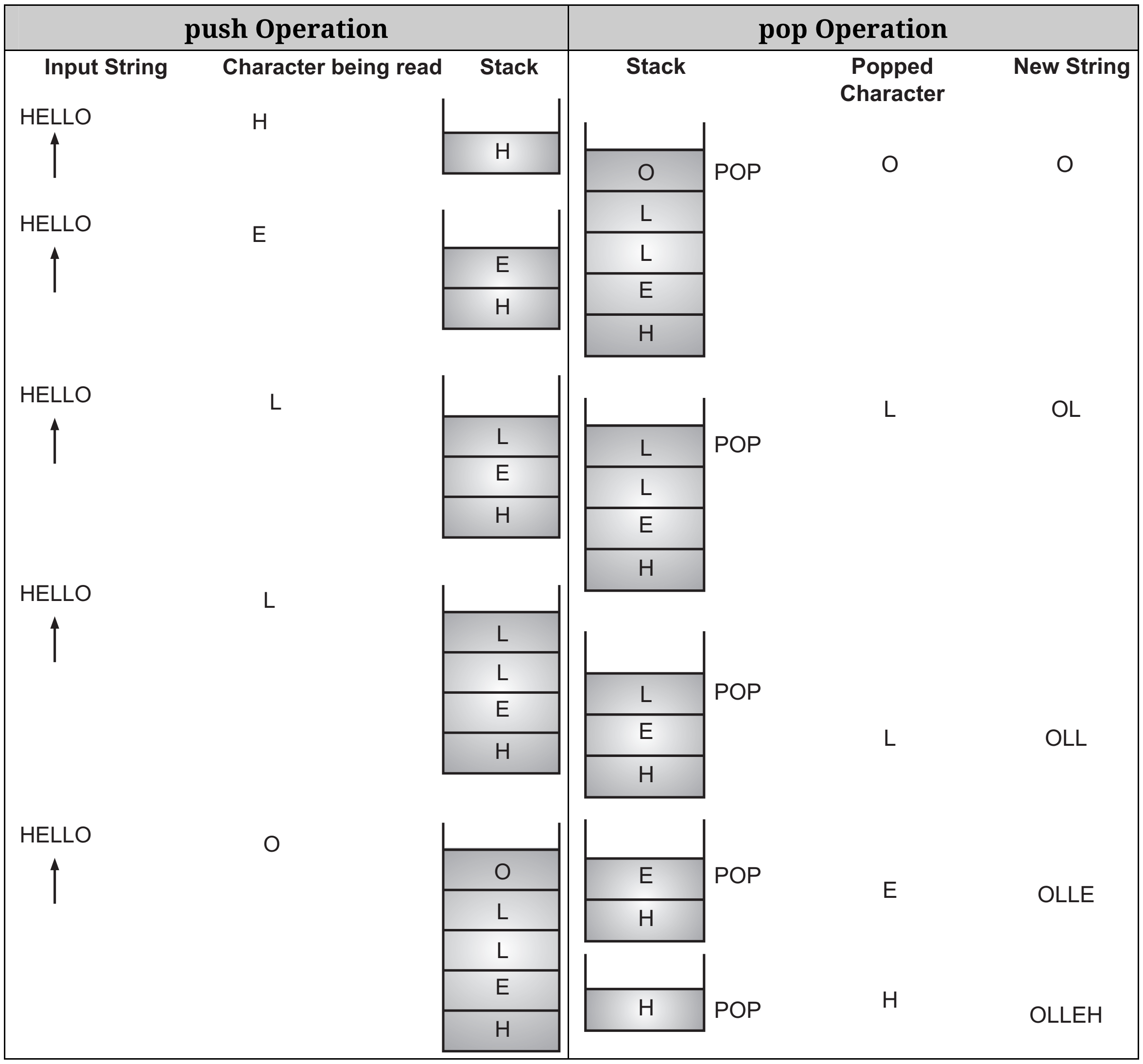


Fig. 4.8

Program 4.6: Program to reverse a string using stack.

```
#include<stdio.h>
#include<string.h>
#include "cstack.h"
int main()
{
   struct stack s;
   char ch[MAX], e;
   int n=0,i;
   initc(&s);
   printf("enter a string to push\n");
   gets(ch);
   while(ch[n]!='\0')
   {
      n++;
   }
   printf("total char is %d\n",n);
   pushc(&s,ch,n);
   while(!isemptyc(&s))
   {
      e=popc(&s);
      printf("%c",e);
   }
   printf("\n");
   return 0;
}
void initc(struct stack *cs)
{
   cs->top=-1;
}
void pushc(struct stack *cs,char ch[],int n)
{
   int i;
   if(isfullc(cs))
   {
      printf("stack overflow\n");
      return;
   }
```

```
    else
    {
       for(i=0;i<=n;i++)
       cs->a[++cs->top]=ch[i];
    }
 }
 char popc(struct stack *cs)
 {
    if(isemptyc(cs))
    {
       return;
    }
    else
    {
       return cs->a[cs->top--];
    }
 }
 int isemptyc(struct stack *cs)
 {
    if(cs->top==-1)
      return 1;
    else
      return 0;
 }
 int isfullc(struct stack *cs)
 {
    if(cs->top==MAX-1)
        return 1;
    else
        return 0;
 }
```

Output:

```
enter a string to push
hello
total char is 5
olleh
```

4.4.3 Palindrome Checking

- A palindrome is nothing but any number or a string which remains unaltered/ unchanged when reversed.

  Example: 12321, RACECAR.
- A palindrome is a sequence of symbols such as a word, phrase verse, or sentence, and that reads the same way from either direction, forward or backward. Examples of single word palindromes include "pop," "level," and "radar."
- In order to check if the word is a palindrome, the first and the last characters are compared, then the second and the second last characters are compared and so on until all the characters have been compared.
- During the process, as soon as two symbols do not match, then the word is not a palindrome.

Program 4.7: Program to find palindrome or not.

```
/*cstack.h*/ /*Header File*/
#define MAX 30
struct stack
{
   char a[MAX];
   int top;
};
void initc(struct stack *);
void pushc(struct stack *,char[],int);
char popc(struct stack *);
int isfullc(struct stack *);
int isempty(struct stack *);
/*Program*/
#include <stdio.h>
#include <string.h>
#include "cstack.h"
int reverse(struct stack *,char[],int);
int main()
{
   struct stack s;
   char ch[MAX],e;
   int n=0,i,p;
   initc(&s);
   printf("enter a string to push\n");
   gets(ch);
```

```
    while(ch[n]!='\0')
    {
       n++;
    }
    printf("total char is %d\n",n);
    pushc(&s,ch,n);
    p=reverse(&s,ch,n);
    if(p!=0)
      printf("it is palandrome\n");
     else
        printf("not a palandrome\n");
    return 0;
}
void initc(struct stack *cs)
{
    cs->top=-1;
}
void pushc(struct stack *cs,char ch[],int n)
{
    int i;
    if(isfullc(cs))
    {
       printf("stack overflow\n");
       return;
    }
    else
    {
       for(i=0;i<=n;i++)
       cs->a[++cs->top]=ch[i];
    }
}
char popc(struct stack *cs)
{
    if(isemptyc(cs))
    {
       return;
    }
    else
    {
        return cs->a[cs->top--];
    }
}
int isemptyc(struct stack *cs)
{
    if(cs->top==-1)
         return 1;
    else
         return 0;
}
int isfullc(struct stack *cs)
{
    if(cs->top==MAX-1)
          return 1;
    else
          return 0;
}
int reverse (struct stack *cs,char ch[],int n)
{
    int i=0;
    if(isemptyc(cs))
    {
        return 0;
    }
    cs->top=cs->top-1;
    while(cs->top!='\0' && i<n)
    {
        if(cs->a[cs->top]==ch[i])
         {
            i++;
            cs->top=cs->top-1;
            continue;
        }
        else
        return 0;
    }
    return 1;

}
```

Output:

```
enter a string to push
radar
total char is 5
it is palandrome
```

4.4.4 Expression Types (Infix, Prefix and Postfix)

- An expression is defined as, "a number of operands or data items combined using several operators". One of the prominent applications of stacks is to evaluate the expression.
- Infix, Prefix and Postfix notations are introduced by Jan Lukasiewicz. We consider simple arithmetic expressions with binary operators like +, -, *, /, ^ (power), and parentheses. Operands are single digit numbers (0,1, …,9) or alphabets (A-Z or a-z). Expression may contains "$" or "↑" instead of "^".
- The arithmetic expression written in following forms:
  1. **Infix:** If the operator symbols are placed between the operands, then the expression is in infix notation. For example, (A + B) * C.
  2. **Prefix:** In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.
  3. **Postfix:** If the operator symbols are placed after its operands, then the expression is in Postfix Notation. For example, AB + C *. This notation is also called as Reverse polish notation.
- Stacks are frequently used in evaluation of arithmetic expressions. An arithmetic expression consists of operators and operands. The expression is evaluated using precedence and associativity.

Operator Precedence and Associativity:

Priority (Precedence)	Operators	Associativity	Example
Lowest	+, −	left to right	x − y + z = ((x − y) + z)
Middle	*, /	left to right	x * y/z = ((x * y)/z)
Highest	^ or ↑ or $	right to left	x ^ y ^ z = (x ^ (y ^ z))

Precedence determines order of evaluation: 1+2*3^4 = 1+(2*(3^4)).

Precedence may be changed by parentheses: ((1+2)*3)^4.

1. For example, a * b * c. This expression is evaluate from left to right, since * is left associative operator.
2. For example, a ↑ b ↑ c; this expression is evaluate from right to left since ↑ is right associative operator.
3. Similarly, a + b * c.

- In this expression * has higher precedence than +, hence b * c is evaluated first.

4.4.4.1 Expression Conversion and Evaluation (Prefix, Postfix and Infix Expressions)

- Expression conversion is the most important application of stacks. The way to write arithmetic expression is known as a notation. The process of placing the operator at a particular place in an expression is called polish notation.

- Polish notation, is a symbolic logic invented by Polish mathematician Jan Lukasiewicz in the 1920's. A Polish mathematician found a way to represent the same expression called polish notation.
- Polish notation, also known as Polish prefix notation or simply prefix notation, is a form of notation for logic, arithmetic, and algebra.
- Polish notations are of Infix notation (a normal arithmetic expression is normally called as infix expression like (A+B), Prefix notation (prefix expression by keeping operators as prefix like, (+AB) and postfix notation (we use the reverse way of the above expression for our evaluation. The representation is called Reverse Polish Notation (RPN) or postfix expression like (AB+).

4.4.4.2 Conversion of an Infix Expression to a Postfix Expression

[April 17, 18]

- There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than operands.
- The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared.

Input: Infix expression.

Output: Postfix expression.

Read infix expression from left to right and character by character.

For each character 'ch' in the input string:

```
1. If 'ch' is an operand, then
       output 'ch'.
2. If 'ch' is an operator, then following action should be taken:
   o  Check whether there is any operator already present in stack or not.
   o  if stack is empty, then push operator onto stack.
   o  if present, then check whether priority of 'ch' is greater than priority of
      topmost stack operator.
   o  if priority of 'ch' is greater then push 'ch' onto stack.
   o  else pop and print operator until one of lower priority operator is encountered
      or the stack becomes empty and finally push ch onto stack.
3. if 'ch' is a left parentheses
       push 'ch' into the stack.
4. if 'ch' is a right parentheses
       pop and output tokens until a left parentheses is popped (but do not output left
       parenthesis).
```

- For better understanding, let us consider following example P / Q – R * S + T.

Input Character	Action	Stack	Output
P	Print P	Empty	P
/	Push	/	P
Q	Print Q	/	PQ
–	Pop	–	PQ/
R	Print R	–	PQ/R
*	Push	–*	PQ/R
S	Print S	–*	PQ/RS
+	Pop	–	PQ/RS*
	Pop	Empty	PQ/RS*–
	Push	+	PQ/RS*–
T	Print T	+	PQ/RS*–T
End of input	Pop till empty	Empty	PQ/RS*–T+

- So the postfix expression for given infix expression is : PQ/RS*-T+.

Program 4.8: Program to convert infix expression to postfix expression.

```
#include<stdio.h>
#include<ctype.h>
struct stack
{
   char data[20];
   int top;
};
/* initialization of the stack */
void init(struct stack *s)
{
   s->top=-1;
}
/* checking stack empty or not */
int isempty(struct stack *s)
{
   if(s->top==-1)
      return 1;
   else
      return 0;
}
```

```
/* insert element onto the stack */
void push(struct stack *s,char x)
{
    s->top++;
    s->data[s->top] = x;
}
/* delete element from the stack */
char pop(struct stack *s)
{
    if(s->top == -1)
        return -1;
    else
        return s->data[s->top--];
}
/* finding precedence of an operator */
int priority(char x)
{
    if(x == '(')
        return 0;
    else if(x == '+' || x == '-')
        return 1;
    else if(x == '*' || x == '/')
        return 2;
    else if(x == '^' || x == '$')
        return 3;
    else
        return -1;
}
int main()
{
    struct stack s1;
    char exp[20],x;
    int i;
    init(&s1);
    printf("Enter the infix expression:: ");
    scanf("%s",exp);
    i=0;
```

```
        while(exp[i] != '\0')
        {
            if(isalnum(exp[i]))
                printf("%c",exp[i]);
            else if(exp[i] == '(')
                push(&s1,exp[i]);
            else if(exp[i] == ')')
            {
                while((x = pop(&s1)) != '(')
                    printf("%c", x);
            }
            else
            {
                while(priority(s1.data[s1.top]) >= priority(exp[i]))
                    printf("%c",pop(&s1));
                push(&s1,exp[i]);
            }
            i++;
        }
        while(!isempty(&s1))
        {
            printf("%c",pop(&s1));
        }
    return 0;
    }
```

Output:

```
    Enter the infix expression:: A+B*(C-D)/E
    ABCD-*E/+
```

4.4.4.3 Conversion of an Infix Expression to Prefix Expression

[April 15, 16]

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

  Step 1: Reverse the infix expression.

  Step 2: If expression contains parenthesis then reverse the direction of parenthesis.

  Step 3: Convert the expression into postfix expression.

  (Refer to 4.4.1.1 infix to postfix conversion algorithm)

  Step 4: Reverse the output string.

- **Example:** Consider the infix expression A+B/(B-D).

  Step 1: Reverse the given infix expression

  (D-B)/B+A

  Step 2: Convert reversed string into postfix expression.

Input character	Action	Stack	Output
(	Push ‘ (‘	(	–
D	Print D	(	D
-	Push ‘ - ‘	(–	D
B	Print B	(–	DB
)	Pop ‘ - ‘ and Print	(	DB-
	Pop ‘ (‘and Print	Empty	DB-
/	Push ‘ / ’	/	DB-
B	Print B	/	DB-B
+	Pop ‘ / ‘and Print	Empty	DB-B/
	Push	+	DB-B/
A	Print A	+	DB-B/A
End of input	Pop till stack empty	Empty	DB-B/A+

  Step 3: Reverse output string so that we get prefix string.

  Output string is DB-B/A+

  Prefix expression = +A/B-BD

Program 4.9: Program to convert infix expression to prefix.

```
# include<stdio.h>
# include<string.h>
# include<ctype.h>
# define MAX 20
struct stack
{
char data[MAX];
int top;
};
/* initialization of stack */
void init(struct stack *s)
{
   s->top=-1;
}
```

```
/*checking stack empty or not */
int isempty(struct stack *s)
{
   if(s->top==-1)
         return 1;
   else
         return 0;
}
/* insert element onto the stack */
void push(struct stack *s,char x)
{
    s->top++;
    s->data[s->top] = x;
}
/* delete element from the stack */
char pop(struct stack *s)
{
    if(s->top == -1)
        return -1;
    else
        return s->data[s->top--];
}
/* finding precedence of an operator */
int priority(char x)
{
    if(x == ')')
        return 0;
    else if(x == '+' || x == '-')
        return 1;
    else if(x == '*' || x == '/')
        return 2;
    else if(x == '^' || x == '$')
        return 3;
    else
   return -1;
}
/* convert infix expression into prefix expression */
void convert(struct stack *s,char exp[],char pre[])
{
int i=0,j=0,k=0,l;
char x;
strrev(exp);
```

```
        while(exp[i] != '\0')
        {
            if(isalnum(exp[i]))
                pre[j++]=exp[i];

            else if(exp[i] == ')')
                push(s,exp[i]);
            else if(exp[i] == '(')
            {
                while((x = pop(s)) != ')')
                    pre[j++]=x;
            }
            else
            {
             while(priority(s->data[s->top]) > priority(exp[i]))
                    pre[j++]=pop(s);

             push(s,exp[i]);
            }
            i++;
        }
        while(!isempty(s))
        {
            pre[j++]=pop(s);
        }
    pre[j]='\0';
    strrev(pre);
    }
    int main()
    {
    struct stack s1;
    char infix[20],prefix[20];
    init(&s1);
    printf("Enter the infix expression:\n");
    scanf("%s",infix);
    convert(&s1,infix,prefix);
    printf("The prefix is: %s”,prefix);
    return 0;
    }
```

Output:

```
    Enter the infix expression:
       a * b + c * d
    The prefix is: + * ab * cd
```

4.4.4.4 Conversion of Postfix Expression to an Infix Expression

- Stacks can be used to convert given postfix expression to corresponding infix expression. The algorithm is given below:

```
Input:     Postfix expression.
Output:    Infix expression.
Step 1 :   Iterate the given expression from left to right, one character at a time
Step 2 :   If a character is operand, push it to stack.
Step 3 :   If a character is an operator,
           3.1  Pop operand from the stack, say it's as s1.
           3.2  Pop operand from the stack, say it's as s2.
           3.3  Perform (s2 operator s1) and push it to stack.
Step 4 :   Once the expression iteration is completed, initialize the result string and
           pop out from the stack and add it to the result.
Step 5 :   Return the result.
```

Example 3: Consider the Postfix expression ABC/-AK/L-* convert it into infix.

Postfix Expression: A B C / - A K / L - *

Token	Action	Stack	Notes
A	Push **A** to stack	[A]	
B	Push **B** to stack	[A, B]	
C	Push **C** to stack	[A, B, C]	
/	Pop **C** from stack	[A, B]	Pop two operands from stack, C and B. Perform B/C and push (B/C) to stack
	Pop **B** from stack	[A]	
	Push **(B/C)** from stack	[A, (B/C)]	
-	Pop **(B/C)** from stack	[A]	Pop two operands from stack, (B/C) and A. Perform A−(B/C) and push (A−(B/C)) to stack
	Pop **A** from stack	[ ]	
	Push **(A−(B/C))** to stack	[((A−(B/C))]	
A	Push **A** to stack	[((A−(B/C)), A]	
K	Push **K** to stack	[((A−(B/C)), A, K]	
/	Pop **K** to stack	[((A−(B/C)), A]	Pop two operands from stack, K and A. Perform A/K and push (A/K) to stack
	Pop **A** to stack	[((A−(B/C))]	
	Push **(A/K)** to stack	[(A−(B/C)), (A/K)]	
L	Push **L** to stack	[(A−(B/C)), (A/K), L]	
−	Pop **L** from stack	[(A−(B/C)), (A/K)]	Pop two operands from stack, L and (A/K). Perform (A/K)−L and push ((A/K)−L) to stack
	Pop **(A/K)** from stack	[((A−(B/C))]	
	Push **((A/K)−L)** to stack	[(A−(B/C)), ((A/K)−L)]	
*	Pop **((A/K)−L)** from stack	[((A−(B/C))]	Pop two operands from stack, (A/K−L) and A−(B/C). Perform (A−(B/C))*((A/K)−L) and push (A−(B/C))*((A/K)−L) to stack
	Pop **((A−(B/C))** from stack	[]	
	Push **((A−(B/C))*((A/K)−L))** **to** stack	[((A−(B/C))*((A−K)−L))]	
Infix Expression: ((A−(B/C))*((A/K)−L))			

Program 4.10: Program to convert postfix expression to infix expression.

```
#include <stdio.h>
#include <stdlib.h>
int top = 10;
struct node
{
   char ch;
   struct node *next;
   struct node *prev;
}  *stack[11];
typedef struct node node;
void push(node *str)
{
   if (top <= 0)
   printf("Stack is Full ");
   else
   {
      stack[top] = str;
      top--;
   }
}
node *pop()
{
   node *exp;
   if (top >= 10)
      printf("Stack is Empty ");
   else
      exp = stack[++top];
   return exp;
}
void convert(char exp[])
{
   node *op1, *op2;
   node *temp;
   int i;
   for (i=0; exp[i]!='\0';i++)
   if (exp[i] >= 'a'&& exp[i] <= 'z'|| exp[i] >= 'A' && exp[i] <= 'Z')
   {
        temp = (node*)malloc(sizeof(node));
        temp->ch = exp[i];
        temp->next = NULL;
        temp->prev = NULL;
        push(temp);
    }
    else if (exp[i] == '+' || exp[i] == '-' || exp[i] == '*'
                                        || exp[i] == '/' || exp[i] == '^')
    {
        op1 = pop();
        op2 = pop();
        temp = (node*)malloc(sizeof(node));
        temp->ch = exp[i];
        temp->next = op1;
        temp->prev = op2;
        push(temp);
    }
}
void display(node *temp)
{
    if (temp != NULL)
    {
        display(temp->prev);
        printf("%c", temp->ch);
        display(temp->next);
    }
}
void main()
{
    char exp[50];
    printf("Enter the postfix expression:");
    scanf("%s", exp);
    convert(exp);
    printf("\nThe Equivalant Infix expression is:");
    display(pop());
    printf("\n\n");
}
```

Output:

```
Enter the postfix expression:ABC/-AK/L-*
The Equivalant Infix expression is:A-B/C*A/K-L
```

4.4.4.5 Evaluation of Postfix Expression [April 15; Oct. 16]

- After converting infix to postfix, we need postfix evaluation algorithm to find the correct answer. Here also we have to use the stack data structure to solve the postfix expressions.
- From the postfix expression, when some operands are found, pushed them in the stack. When some operator is found, two items are popped from the stack and the operation is performed in correct sequence.
- After that, the result is also pushed in the stack for future use. After completing the whole expression, the final result is also stored in the stack top.

Algorithm for Evaluation of Postfix Expression:

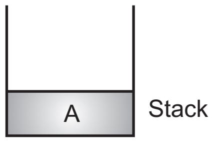
```
Read postfix expression character by character from left to right.
1.  if (read character = operand) then
        push the element in the stack
2.  if (read character = operator) then
    (a)  pop 2 operands from the stack.
         y=pop() from the stack.
         x=pop() from the stack.
    (b)  Evaluate them by applying the operator i.e. " x operator y ".
    (c)  push the result onto stack.
3.  if string is ended, pop the result.
```

Example: Evaluate the following postfix expression.

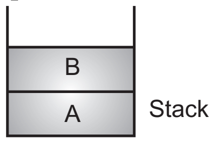
AB + C - (Let A = 3, B = 4, C = 2)

Ans. Read given postfix string from left to right.

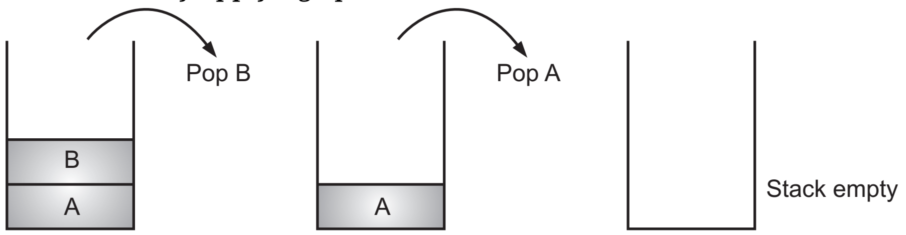
1. First element is operand A, push A onto stack.



2. Second element is operand B, push B onto the stack.

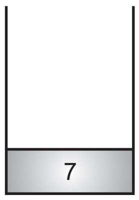


3. The third element is operator '+', pop 2 elements from stack (i.e. A and B) and then evaluate them by applying operator ‘ + ’.

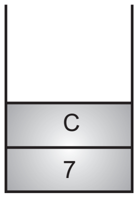


Evaluating, A + B = 3 + 4 = 7.

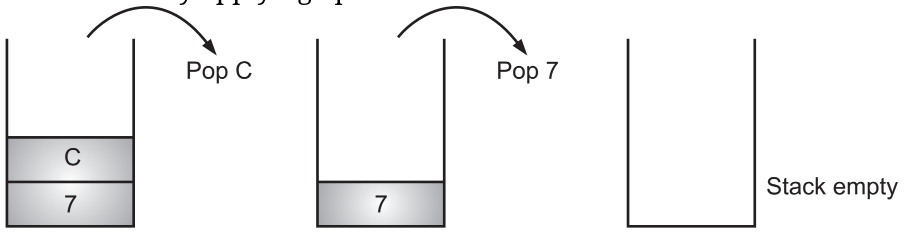
4. Push the result onto the stack.



5. Next element in operand C, push onto the stack.

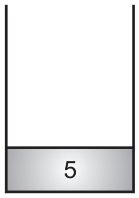


6. Next element is operator ' * ', pop 2 elements from stack (i.e. C and 7) and then evaluate them by applying operator ‘ * ’.

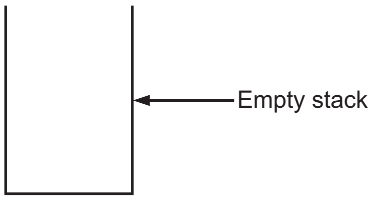


Evaluating = 7 - C
= 7 - 2
= 5

7. Push result 5 onto the stack.



8. End of expression, therefore pop result from stack.



Result of given postfix expression is 5.

Program 4.11: Program to evaluate a postfix expression.

```
#include<stdio.h>
#include<ctype.h>
int stack[20];
int top = -1;
```

```
void push(int x)
{
   stack[++top] = x;
}
int pop()
{
   return stack[top--];
}
int main()
{
   char exp[20];
   setbuf(stdout, NULL);  // turn off buffering of stdout
   char *temp;
   int n1,n2,n3,num;
   printf("Enter postfix expression without space :: ");
   scanf("%s",exp);
   temp = exp;
   while(*temp != '\0')
   {
       if(isdigit(*temp))
       {
          num = *temp - 48;        //convert exp. string to integer
          push(num);
       }
       else
       {
          n1 = pop(); //operand1
          n2 = pop(); //operand2
          switch(*temp)
          {
             case '+':
             {
                n3 = n2 + n1;
                break;
             }
             case '-':
             {
                n3 = n2 - n1;
                    break;
             }
             case '*':
             {
                n3 = n2 * n1;
                    break;
             }
```

```
                case '/':
                {
                   n3 = n2 / n1;
                       break;
                }
            }
            push(n3);
        }
         temp++;
    }
    printf("\nThe result of postfix expression %s = %d\n\n",exp,pop());
    return 0;
}
```

Output:

```
Enter postfix expression without space :: 636+5*9/-
The result of postfix expression 636+5*9/- = 1
```

4.4.4.6 Evaluation of Prefix Expression

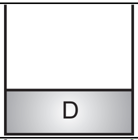
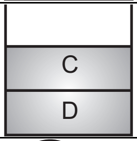
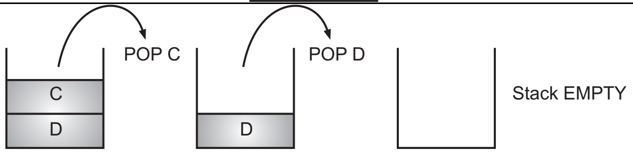
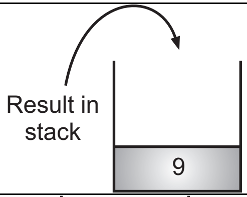
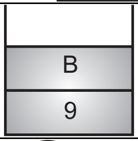
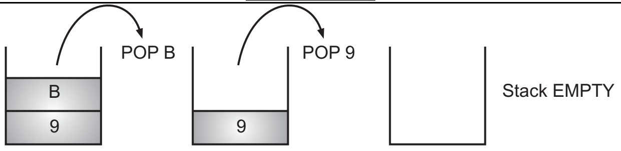
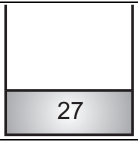
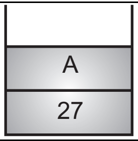
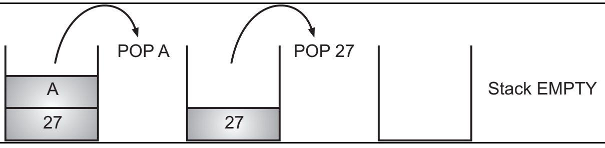
- With prefix notation, we can evaluate each expression in a single scan of the expression. A stack can be conveniently used for the evaluation of prefix expressions.
- To evaluate prefix expression, first reverse the given infix expression. Then read the expression from left to right. Each encountered element is examined. If the element is an operand, then the element is pushed on to the stack.
- If the element is an operator, then the two operands are popped from the stack and the desired operation is done.
- The result of the operations is again pushed onto the stack. This process is repeated till the end of the expression is encountered. The final result is popped from the stack.
- **Example:** Consider following prefix expressions with values:
  say (-*+ABCD), let A=4, B=3, C=2, D=5. After putting value in expression we will get (-*+4325).
  Reverse the expression and follow the above mentioned steps.

Symbol Scanned	Stack	Action
5	5	PUSH into stack.
2	5,2	PUSH into stack.
3	5,2,3	PUSH into stack.
4	5,2,3,4	PUSH into stack.
+	5,2,7	POP two elements and perform + operation and PUSH intermediate result in stack.
*	5,14	POP two elements and perform * operation and PUSH intermediate result in stack.
-	-9	POP two elements and perform - operation and PUSH intermediate result in stack.

Example: Take another example, evaluate the prefix expression:

+ A * B + CD if A = 2 B = 3 C = 4 D = 5

Starting from right to left +A * B + CD. When the values are inserted the expression becomes +2*3+45.

Step 1 : First element its operand D, PUSH D into the stack.	
Step 2 : Second element is also C, PUSH C also into the stack.	
Step 3 : The third element '+' is an operator, POP two elements from the stack, i.e., C and D and evaluate the expression Evaluating = D + C = 4 + 5 = 9	
Step 4: PUSH the result, i.e., 9 into the stack.	
Step 5 : Next element B is operand; PUSH it into the stack.	
Step 6: Next * is an operator, POP two operands from the stack, i.e. B and 9 and evaluate the expression. Evaluating = 9*B = 9*3 = 27	
Step 7 : PUSH the result 27 into the stack.	
Step 8: Next A is an operand, PUSH it into stack.	
Step 9: Next + is an operator, POP two elements and evaluate. Evaluating = 27 + A = 27 + 2 = 29	
Step 10 : End of expression, thus, the result is 29.	

Program 4.12: Program for evaluation of prefix expression.

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
int stack[20];
int top = -1;
void push(int x)
{
   stack[++top] = x;
}
int pop()
{
   return stack[top--];
}
int main()
{
   char exp[20], revexp[20];
   setbuf(stdout, NULL);   // turn off buffering of stdout
   char *temp=NULL;
   int n1, n2, n3, num;
   printf("Enter prefix expression without space :: ");
   scanf("%s",exp);
   strcpy(revexp,strrev(exp));  //convert prefix to postfix
        temp=revexp;
        while(*temp != '\0')
        {
          if(isdigit(*temp))
          {
             num = *temp - 48;//convert exp. string to integer
             push(num);
          }
          else
          {
          n1 = pop();
          n2 = pop();
```

```
                switch(*temp)
                {
                case '+':
                {
                   n3 = n2 + n1;
                   break;
                }
                case '-':
                {
                   n3 = n2 - n1;
                   break;
                }
                case '*':
                {
                   n3 = n2 * n1;
                   break;
                }
                case '/':
                {
                   n3 = n2 / n1;
                   break;
                }
             }
             push(n3);
          }
       temp++;
       }
       printf("\nThe result of prefix expression %s  =  %d\n\n",exp,pop());
       return 0;
       }
```

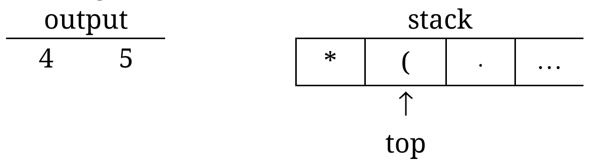
Output:

```
   Enter prefix expression without space :: -*+4325
   The result of prefix expression 5234+*- = -9
```

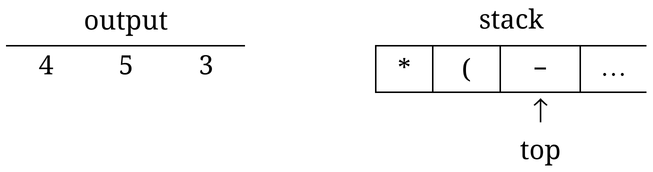
Evaluation of Infix Expression:

- To evaluate an infix expression, many compilers convert this infix expression into its equivalent postfix form first and then evaluate it, which generates the code for evaluating the transformed postfix expression. If we examine an infix expression little carefully we see that parentheses must be used to indicate the priorities of the operators involved in the expression.

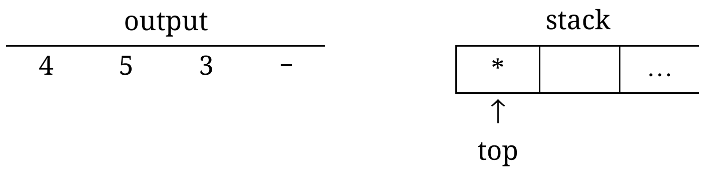
Consider the infix expression 4 * (5 − 3). The infix expression is scanned in left-to-right order. When an operand is found, it is sent to the output. Initially, for the above input 4 is encountered and is sent to output immediately Next, the * operator is found. At this point, another operand is expected after * on which it must be applied. So it must be stored and hence * is pushed on to a stack of operators. Note that before pushing this operator the stack was empty. In general, when an operator is encountered it must be checked against the top stack element. The operator is pushed to the stack if either the stack is empty or if the operator is having a higher priority than the stack top element. In our case * is pushed to the stack. Next an open parenthesis '(' is encountered and is pushed to the stack. Then the operator 5 is found and it is sent to the output. At this stage the output and stack look like the following:



Now the operator '−' is encountered. It is pushed to the stack since the stack top symbol '(' is assumed to have lower priority than any other operator. The operand 3 is found next and it is sent to output directly. Now the output and stack take the following form.



Finally, the right parenthesis ')' is encountered. When a ')' is encountered, the symbols are popped from the stack and sent to output until a '(' is found in stack top. This '(' is popped from the stack but not sent to output. After doing this the output and stack become



There is no other symbol left in the input now. At this stage, the operators are popped from stack and sent to output until the stack is empty. Hence the output will look like and the stack is empty. The output now shows the RPN (Reverse Polish Notation) expression for the given infix expressions.

output

4 5 3 − *

4.4.5 Backtracking Strategy (4 Queens Problem)

- Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

- Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

4 Queens Problem:

- The 4 queens problem can be solved using a stack and backtracking logic.

Step 1 : Only one queen can be placed in any row, we begin by placing the first queen in row 1, column 1. This location is then pushed into a stack, giving the position shown in step 1.

After placing a queen in the first row, we look for a position in the second row. Position 2,1 is not possible because the queen in the first row is guarding this location on the vertical.

Step 2 : Similarly, position 2,2 is guarded on the diagonal. We therefore place a queen in the third column in row 2 and push this location into the stack. This position is shown in step 2.

We now try to locate a position in row 3, but none are possible. The first column is guarded by the queen in row 1, and the other three positions are guarded by the queen in row 2.

Step 3 : At this point we must backtrack to the second row by popping the stack and continue looking for a position for the second-row queen. Because column 4 is not guarded, we place a queen there and push its location into the stack.

Step 4 : Looking again at row 3, we see that the first column is still guarded by the queen in row 1 but that we can place a queen in the second column. We do so and push this location into the stack (step 4). When we try to place a queen in row 4, however, we find that all positions are guarded. Column 1 is guarded by the queen in row 1 and the queen in row 3. Column 2 is guarded by the queen in row 2 and the queen in row 3. Column 3 is guarded by the queen in row 3, and column 4 is guarded by both the queen in row 1 and the queen in row 2.

Step 5 : We therefore backtrack to the queen in row 3 and try to find another place for her. Because the queen in row 2 is guarding both column 3 and column 4, there is no position for a queen in row 3. Once again we backtrack by popping the stack and find that the queen in row 2 has nowhere else to go, so we backtrack to the queen in row 1 and move her to column 2. This position is shown in step 5.

Step 6 : Analyzing row 2, we see that the only possible position for a queen is column 4 because the queen in row 1 is guarding the first three positions. We therefore place the queen in this location and push the location into the stack (step 6).

Step 7 : Column 1 in the third row is unguarded, so we place a queen there. Moving to row 4, we find that the first two positions are guarded, the first by the queen in row 3 and the second by all three queens.

Step 8 : The third column is unguarded, however, so we can place the fourth queen in this column for a solution to the problem.

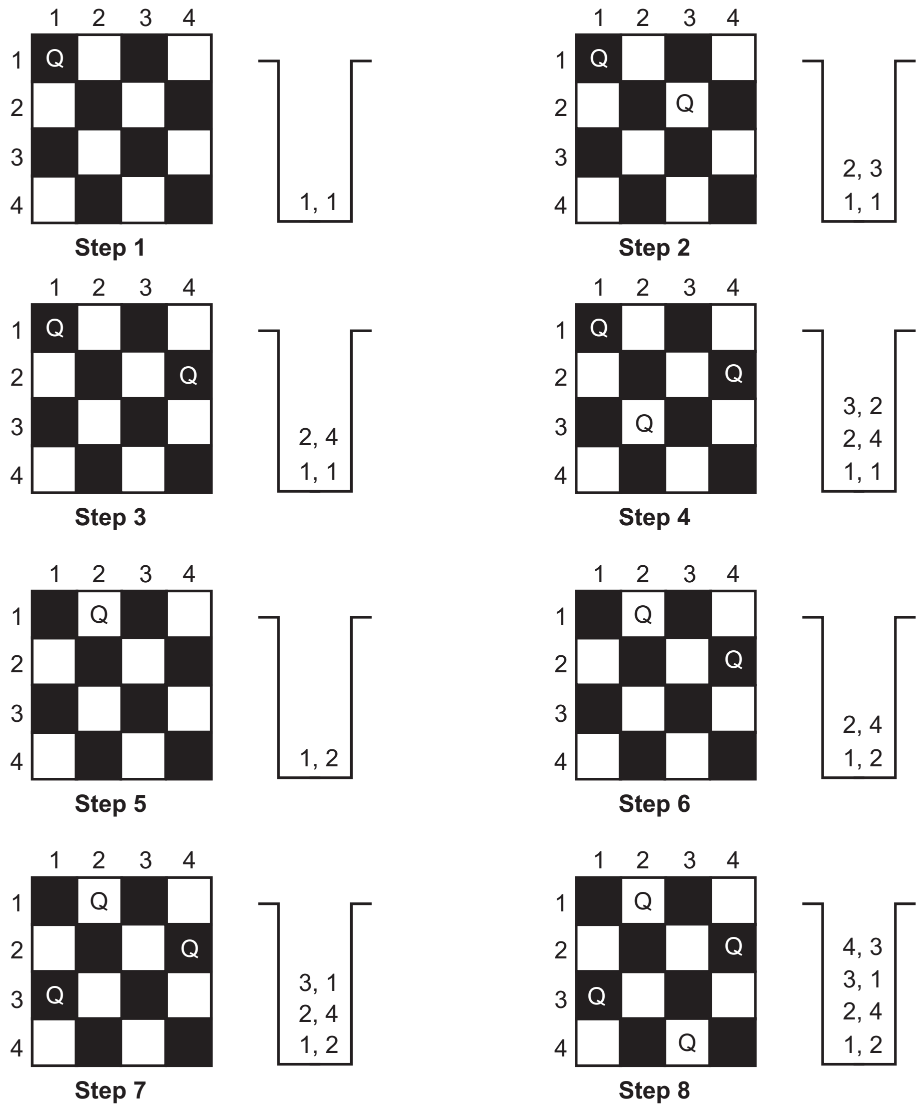


Fig. 4.9

- The following program of 4-queens problem uses recursion and recursion is implemented using stack.

Program 4.13: Program to solve 4 Queen Problem using backtracking.

```
#define N 4
#include <stdbool.h>
#include <stdio.h>
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;
    return true;
}
bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1))
                return true;
            board[i][col] = 0; // BACKTRACK
        }
    }
    return false;
}
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };
    if (solveNQUtil(board, 0) == false) {
        printf("Solution does not exist");
        return false;
    }
    printSolution(board);
    return true;
}
// driver program to test above function
int main()
{
    solveNQ();
    return 0;
}
```

Output:

```
 0  0  1  0
 1  0  0  0
 0  0  0  1
 0  1  0  0
```

Examples:

Example 1: Convert following infix expression into reverse polish notation.

1. P+Q*R–S

   Ans.: PQR*+S–
2. (A+B^C)*D+E^F

   Ans.: ABC^+D*EF^+
3. A/B$C+D*E–A*C

   Ans.: ABC$/DE*+AC*-

Example 2: Evaluate following expression.

1. 9 6 + 3 / 4 2 7 + – *

   Ans.: -25
2. 9 8 + 7 4 – *

   Ans.: 51
3. 6 2 3 + – 3 8 2 / + * 2 $ 3 +

   Ans.: 52
4. E D C – B A + * / where A=5, B=3, C=9, D=2, E=4

   Ans.: 14

Example 3: Trace the output of following code segment.

```
initstack(s);
push(s,7);
push(s,4);
i=pop(s)+1;
while(i)
   push(s,i--);
while(!isempty(s))
   printf("%d\t",pop(s));
```

Ans.: `1   2   3   4   5   7`

PRACTICE QUESTIONS

Q.I Multiple Choice Questions:

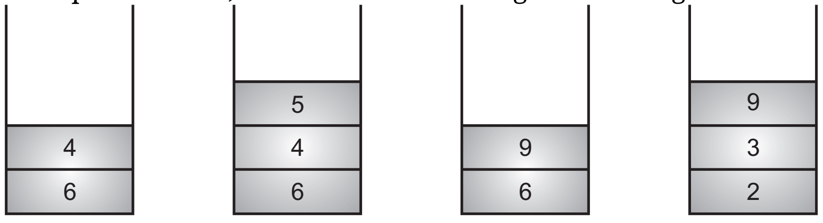
1. A stack is _____.
   (a) Linear data structure (b) Non-linear data structure
   (c) Built-in data structure (c) none of this
2. A stack is linear data structure in which data is stored and retrieved in a _____.
   (a) Last out First in (LOFI) (b) Last in First out (LIFO)
   (c) First in First out (FIFO) (d) Last out Last in (LOLI)
3. Elements are added at which position of the stack?
   (a) Bottom (b) Middle
   (c) Top (d) None of these
4. In a stack, if a user tries to remove an element from empty stack it is called _____.
   (a) Underflow (b) Empty collection
   (c) Overflow (d) Garbage Collection
5. Pushing an element into stack already having five elements and stack size of 5, then stack becomes
   (a) Overflow (b) Crash
   (c) Underflow (d) User flow
6. Process of inserting an element in stack is called _____.
   (a) Create (b) Push
   (c) Evaluation (d) Pop
7. Process of removing an element from stack is called _____.
   (a) Create (b) Push
   (c) Evaluation (d) Pop
8. The following sequence of operations is performed on a stack push(1), push(2), pop, push(1), push(2), pop, pop, pop, push(2), pop. The sequence of popped out values are _____.
   (a) 2, 2, 1, 1, 2 (b) 2, 2, 1, 2, 2
   (c) 2, 1, 2, 2, 1 (d) 2, 1, 2, 2, 2

9. The data structure required to check whether an expression contains balanced parenthesis is _____.

   (a) Stack (b) Tree

   (c) Queue (d) Array

10. In evaluating the arithmetic expression 2*3 – (4+5), using stacks to evaluates its equivalent postfix form, which of the following stake configuration is not possible?



(a) (b) (c) (d)

11. Stack A has the entries a, b, c (with a on top). Stake B is empty. An entry popped out of stake A can be printed immediately or pushed to stake B. An entry popped out of stake B can only be printed. In this agreement, which of the following permutations of a, b, and c are not possible?

   (a) b a c (b) b c a

   (c) c a b (d) a b c

12. Stacks cannot be used to _____.

   (a) Evaluate the arithmetic expression in postfix form.

   (b) Implement recursion.

   (c) Convert a given arithmetic expression in infix form to its equivalent postfix form.

   (d) Allocate resources (like CPU) by the operating system.

13. Which is the postfix expression for the following infix expression?

   A+B*(C+D)/F+D*E

   (a) AB+CD+*F/D+E* (b) ABCD+*F/+DE*+

   (c) A*B +CD / F*DE++ (d) A+*BCD/F*DE++

14. Static implementation of stack uses _____.

   (a) pointers (b) arrays

   (c) strings (d) All of these

15. A stack follows _____ principle.

   (a) FIFO (First In First Out) (b) LIFO (Last In First Out)

   (c) Ordered array (d) Linear tree

16. Which operation is used to insert elements in stack.

   (a) PUSH (b) POP

   (c) Overflow (d) IsEmpty

Answers

1. (a)	2. (b)	3. (c)	4. (a)	5. (a)	6. (b)	7. (d)	8. (a)
9. (a)	10. (d)	11. (c)	12. (d)	13. (b)	14. (b)	15. (b)	16. (a)

Q.II Fill in the Blanks:

1. A _____ is an ordered list with the restriction that elements are added or deleted from only one end of the list termed TOP of stack.
2. _____ implementation of stack uses linked list.
3. _____ operation get the top data element of the stack, without removing it.
4. The process of deleting an element from the top of stack is called _____ operation.
5. isFull**()** function is used to check whether a stack becomes _____ or not.
6. _____ occurs when the stack is full and there is no space for a new element, and an attempt is made to push a new element. If the stack is not full, PUSH operation can be performed successfully.
7. If operator symbols are placed before its operands, then the expression is in _____ notation (+AB).
8. The evaluation of postfix expressions is also implemented through _____.
9. A _____ is a block of statements that can be used to perform a specific task.
10. When a function definition includes a call to itself, it is referred to as a _____ function.
11. To convert infix expression to postfix expression, we will use the _____ data structure.
12. A string is, _____ if string remains same after reversing the sequence of its character (For example, "asdfdsa").

Answers

1. stack	2. dynamic	3. peek()	4. POP	5. full	6. overfull
7. prefix	8. stacks	9. function	10. recursive	11. stack	12. palindrome

Q.III State True or False:

1. A stack is one of the most essential non-primitive linear data structure.
2. The insertion of element onto the stack is called as PUSH and deletion operation is called POP.
3. Static implementation of stack uses linked list to represent stack in memory.
4. The process of putting a new data element onto stack is known as a PUSH Operation.
5. CREATE operation is used to create an empty stack. Initially set the value of TOP is – 1 which is used to show the stack is empty.
6. Underflow occurs when the stack is empty, and an attempt is made to POP an element. If the stack is not empty, POP operation can be performed successfully.
7. If operator symbols are placed after its operands, then the expression is in postfix notation (ab+).
8. A stack can be conveniently used for the evaluation of prefix expressions.

9. The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.
10. The postfix form of A*B+C/D is AB*CD/+.

Answers

1. (T)	2. (T)	3. (F)	4. (T)	5. (T)	6. (T)	7. (T)	8. (T)	9. (T)	10.(T)

Q.IV Answer the following Questions:

(A) Short Answer Questions:

1. What is stack?
2. What is TOP of stack?
3. Give purpose of PUSH operation.
4. Define the term stack overflow.
5. What is means by stack underflow?
6. List any two applications of stack.
7. Give purpose of POP operation.
8. Which Polish notations used in stack?
9. What is the value of the postfix expression 6 3 2 4 + – *.
10. Give postfix form of the expression (A + B) * (C * D – E)*F / G.
11. Define recursion?
12. What is recursive function?

(B) Long Answer Questions:

1. Define stack. How to create it? Explain with example.
2. With the help of diagram describe PUSH and POP operations of stack.
3. Show the stack contents and output while converting the following infix string to postfix.
   A / B $ C + D * E – A * C
4. Describe static implementation of stack in detail.
5. Consider the postfix expression PQ + RS – *. Given the steps for evaluation when P = 9, Q = 8, R = 7 and S = 4.
6. Describe following terms with suitable diagram:
   (i) Stack overflow
   (ii) Stack underflow.
7. Convert following infix expression into a postfix expression. Show all steps.
   (P + (Q * R – (S/T ↑ U)* V)* W)
8. Recursion is one of the application of stack – YES/NO? Explain it for calculating the factorial of a number 5 using recursion.
9. How stack is useful in reversing a string? Write a C program to reverse a string using stack.

10. Translate the following infix expression to its equivalent prefix expression. (x + y) * (p – q). Evaluate the above prefix expression with following values x = 2, y = 3, p = 6, q = 2.
11. Give an algorithm to evaluate postfix expressions.
12. Give an algorithm to convert an expression in infix from into postfix form.
13. Explain the procedure of conversion of infix expression to postfix expression with example.
14. Describe the procedure of conversion of prefix expression to postfix expression with example.
15. Write C program to convert the infix expression to postfix expression.
16. Transform the following infix expressions into their equivalent postfix expressions:
    (i) (A–B)*(D/E)
    (ii) (A+B^D)/(E–F)+G
    (iii) A*(B+D)/E–F*(G+H/K)
    (iv) (A+B)*(C$(D–E)+F)/G)$(H–J).
17. Transform the following infix expressions into their equivalent prefix expressions:
    (i) (A–B)*(D/E)
    (ii) (A+B^D)/(E–F)+G
    (iii) A*(B+D)/E–F*(G+H/K).
18. Write short note on
    (i) Evaluation of Infix expression
    (ii) Evaluation of Postfix expression
19. What are different forms to write an expression. Explain with suitable examples.
20. Write C program to convert infix expression to prefix expression.

UNIVERSITY QUESTIONS AND ANSWERS

April 2015

1. Convert the infix expression to prefix: ((A + B) * C – (D – E)). **[1 M]**

Ans. Refer to Section 4.4.4.3.

2. Name the data structure used for Reversing a string. **[1 M]**

Ans. Refer to Section 4.4, Point (4).

3. Consider the given infix expression u * v + z/w. Write its postfix expression. Also show steps to evaluate the postfix expression using stack. Given: u = 3, v = 1, z = 4, w = 2. **[4 M]**

Ans. Refer to Section 4.4.4.5.

4. Give the output of the following code segment:

```
void fun (int n)
{
   struct stack s; init (&s);
   white (n > 0)
   {
      push (&s, n%2);
      n = n/2;
   }
   while (! isempty (&s))
   printf ("%d", pop (&s));
}
```

The call to the function is fun (10), also explain the steps of the output. **[3 M]**

Ans. Refer to Section 4.4.1.

April 2016

1. Convert the expression: ((A + B) * C – (D – E) ^ (F + G) to equivalent postfix notation. **[1 M]**

Ans. Refer to Section 4.4.4.5.

2. Write a C function to reverse a sentence using stack (do not define any stack function). **[5 M]**

Ans. Refer to Section 4.4.2.

3. Convert the infix expression (A +B) * D + E/F to prefix notation. Show the stack elements. **[5 M]**

Ans. Refer to Section 4.4.4.3.

October 2016

1. Write any two applications of stack. **[1 M]**

Ans. Refer to Section 4.4.

2. Evaluate the postfix expression: xy + z – wu*/ where, x = 7, y = 15, z = 2,
w = 4, u = 3. **[4 M]**

Ans. Refer to Section 4.4.4.5.

3. Trace the output of the following:

```
void fun()
{
   int i=5, x, y, z;
   initstack();
   while (i<8)
   {
      push (i+3);
      i=i+1;
   }
   x = pop();
   y = pop();
   push(i+0);
   z = pop();
   push(x+y+z);
   push(y+z);
   while(!stackempty())
   printf("\n %d", pop());
}
```
[3 M]

Ans. Refer to Section 4.2.

April 2017

1. Name the data structure used in recursion. **[1 M]**

Ans. Refer to Section 4.4.1.

2. Convert the infix expression to postfix expression showing the contents of stack at each step: (A + B * C – (D/E ^ F) * G) * H). **[4 M]**

Ans. Refer to Section 4.4.2.

3. Give the output of the following code:

```
int i=1, x, y, z;
initstack( );
while(i < 3)
{
   push(i * i);
   i=i+1;
}
x=pop();
y=pop();
push(i*i);
2=pop();
push(x+y+z);
push(x*y);
while(!stack empty())
printf("\n%d",pop());
```
[3 M]

Ans. Refer to Section 4.2.

October 2017

1. What are postfix and prefix forms of the expression? A + B * (C – D)/(P – R). **[1 M]**

Ans. Refer to Section 4.4.4.

2. Define stack. List out two operations of stack. **[1 M]**

Ans. Refer to Sections 4.1 and 4.2.

3. What are the applications of stack? **[4 M]**

Ans. Refer to Section 4.4.

4. Evaluate the expression using stack: A + B * C – D.
Given data: A = 4, B = 3, C = 5 and D = 1. **[4 M]**
First convert expression to postfix.

Ans. Refer to Sections 4.4.4.4 and 4.4.4.5.

April 2018

1. List different operations of stack. **[1 M]**

Ans. Refer to Section 4.2.

2. Convert the infix expression:
A | B $ C + D * E – A * C to postfix notation show the stack contents. **[4 M]**

Ans. Refer to Section 4.4.4.2.

3. Give the output of the following sample code: **[3 M]**

```
initstack (s)
push (s, 9);
push (s, 4);
i = pop (s);
while (i > 0)
{
   push (s, i*i);
   i – –;
}
while (! stack empty (s))
printf("%d\n", pop (s));
```
[3 M]

Ans. Refer to Section 4.2.

■■■

CHAPTER 5

Queue

Objectives ...

- ➢ To study Queue Data Structure
- ➢ To learn different Operations and Applications on Queues
- ➢ To study different Types of Queues with Implementations

5.0 INTRODUCTION

- A queue is a non-primitive, linear data structure. A queue is First-In-First-Out (FIFO) structure.
- Queue is an ordered collection of homogeneous data elements like stack. It is a linear data structure in which insertion and deletion operations take place at two ends called rear (R) and front (F) of the queue respectively.
- We know the simple applications of queues in our everyday life such as line of people waiting for a bus, or a railway reservation counter, or a movie theatre, ticket counter, where we have to wait till our turn.
- In Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free. In computer science, for multi-user system, there is queue of process (jobs) waiting to get CPU.

5.1 OVERVIEW AND DEFINITION OF QUEUE

- A queue is a linear list of elements in which data can only be inserted at one end, called the rear and deleted from other end, called the front.
- Queue is an ordered, homogenous collection of elements in which elements are appended at one end called rear end and elements are deleted at other end called front end,
- The first element in a queue will be removed first from the list i.e. data are processed through the queue in the order in which they are received. Therefore, queue is called First-In-First-Out (FIFO).

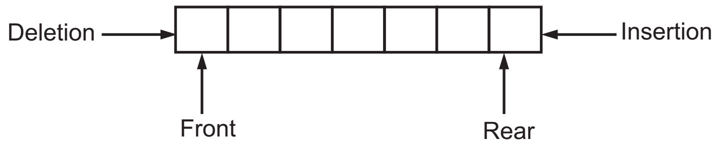
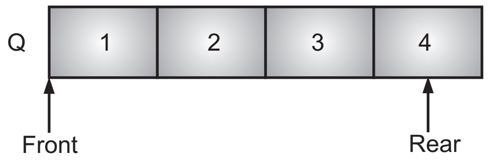


Fig. 5.1: A Queue

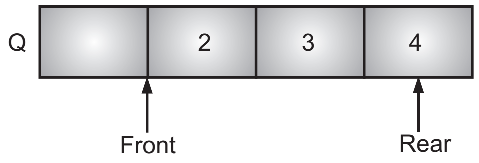
Example of Queue:

- Consider a queue of five elements we can represent the queue as shown in Fig. 5.2 (a).



(a)

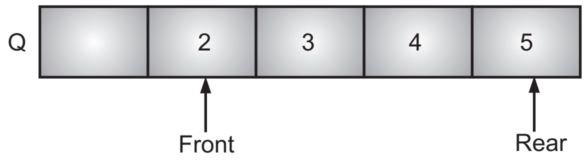
- If we want to delete the element from queue it is deleted only at front end; we get,



(b)

1 is deleted from first and front is incremented by one.

- If we want to insert the element 5, it is inserted from rear shown in Fig. 5.2 (c).



(c)

Fig. 5.2: Queue with Operations

Advantages of Queues:

1. Queues are flexible, requiring no communication programming.
2. Adding or removing elements can be done quickly.
3. Queue is used in many applications such as printing documents.
4. Queue provides first-in, first-out access.

Disadvantages of Queues:

1. A queue is not readily searchable.
2. Adding or removing elements from the middle of the queue is very complex.
3. Slow access to other items.

5.2 OPERATIONS ON QUEUE

- A queue is an Abstract Data Type (ADT). The operations on queue are as follows:
  1. **Create(Q):** Create (Q) creates an empty queue Q.
  2. **Front:** Get the front item from queue.
  3. **Rear:** Get the last item from queue.
  4. **Enqueue or Insert(Q, x):** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition. It adds the element x to the rear end of queue Q and returns the resulting queue.

5. **Dequeue or Delete(Q):** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition. It deletes the element from the front end of queue and returns the resulting queue.
6. **IsFull(Q):** It returns true if queue is full otherwise returns false.
7. **IsEmpty(Q):** It returns true if queue is empty otherwise returns false.
8. **peek(Q):** It is used to return the value of first element without deleting it.
9. **init(Q):** This operation is used to initialize a queue.t

Time Complexity in Queue:

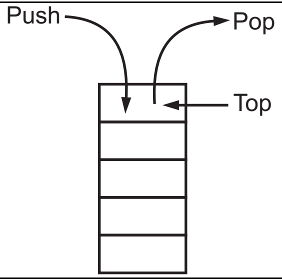
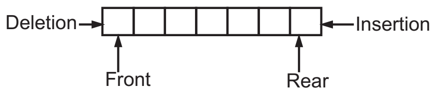
- The time complexity of the enqueue operation is O(1) because no need to traverse the entire list to find the new tail, just need to add a new node.
- For dequeueing, we only need to set the next node of the current head as the new head and return the value of the old head. Therefore the time complexity of the dequeue operation is O(1).
- Searching for a value is done by traversing through all the items, starting from the head. Therefore, the time complexity is O(n).
- Queues offer random access to their contents by shifting the first element off the front of the queue. We have to do this repeatedly to access an arbitary element somewhere in the queue. Therefore, access is O(n).

5.3 DIFFERENCE BETWEEN STACK AND QUEUE

- Following table compares stack and queue data structures:

Terms	Stack	Queue
Definition	A linear data structure with an ordered collection of elements that are inserted and deleted from same end called top.	A linear data structure with an ordered collection of elements that are inserted at one end called rear and deleted from other end called front.
Structure	Same end is used to insert and delete elements.	One end is used for insertion, i.e., rear end and another end is used for deletion of elements, i.e., front end.
Working principle	LIFO (Last in First out). In stacks, the last inserted object is first to come out.	FIFO (First in First out). In queues, the object inserted first is first deleted.
Variant	Stack does not have variants.	Queue has variants like circular queue, priority queue.
Number of pointers used	In stacks only one pointer is used. It points to the top of the stack.	In queues, two different pointers are used for front and rear ends.

contd. ...

Operations performed	Stack operations are called push and pop.	Queue operations are called enqueue and dequeue.
Examination of empty condition	Top == −1	Front == −1
Examination of full condition	Top == Max – 1	Rear == Max – 1
Visualized as	Stacks are visualized as vertical collections.	Queues are visualized as horizontal collections.
Diagrammatic representation		
Example	A stack of plates/books in a cup-board.	Checkout at any book store, cashier line in any store.
Applications	Backtracking, recursive function call, conversion of infix to postfix expression, postfix expression evaluation	Used in serving requests of a single shared resource (printer, disk, CPU)

5.4 REPRESENTATION OF QUEUES

- There are two ways to represent a queue in memory:
  1. Static (using an array), and
  2. Dynamic (using a linked list).

5.4.1 Static Implementation of Queue (Using Array)

- We know that array stores fixed number of elements and often insertion, deletion operations are not suitable on array data structures. On the other hand, queue keeps changing due to addition and removal of items.
- Hence, implementation of array should be such that, queue may expand or shrink dynamically. There is no limitation on the number of items in the queue.
- A one-dimensional array is used to represent a queue. The Fig. 5.3 shows the representation of a queue as an array.

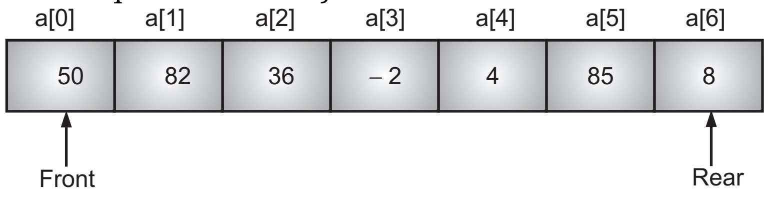


Fig. 5.3: Representation of a Queue as an Array

- For implementation of queue as an array we required a one-dimensional array, say data and two variable front and rear.
- The **Declaration of Queue** is as follows:

```
#define max 20
struct queue
{
   int a[max];
   int front, rear;
};
```

- Let us see implementation of operations on queue:

1. **Create(Q):** This operation creates an empty queue. Front and Rear will indicate empty queue. Hence, as in 'C' language array index ranges between 0 to (max – 1), front and rear are initialize to – 1. We write a function as:

```
void initq (struct queue *q)
{
   q→front = q→rear = – 1;
}
```

2. **IsEmpty(Q):** This operation checks whether, the queue is empty or not. Before we delete element from queue we must check whether queue is empty or not. This is verified by comparing front and rear. **[April 16]**

   If front = rear then IsEmpty(Q) returns true otherwise returns false.

```
int isEmpty (struct queue *q)
   {
   if (q→front == q→rear)
        return 1;
   else
        return 0;
   }
```

3. **IsFull(Q):** For array implementation, due to its fixed size, we need to check queue is full or not. Before insertion, we must check whether queue is full or not. When rear is pointing to last location in the array, then there is no space for new item and queue is full.

```
int IsFull(struct queue *q)
{
   if (q → rear == max – 1)
      return 1;
   else
      return 0;
}
```

4. **Insert(Q, x):** This operation insert the item 'x' in queue if queue is not full. New item is added to the location (rear + 1). **[Oct. 16]**

```
void insert(struct queue *q, int x)
{
    q → data[++(q → rear)] = x;
}
```

5. **Delete(Q):** This operation deletes elements from the front of queue and returns the same and also sets front to point to next item. We should increment front first and then delete the item and then return. **[Oct. 16]**

```
int delete(struct queue *q)
{
    return (q → data[q → front++]);
}
```

- Actual implementation of above functions along with a caller function is given in the following program.

Program 5.1: Static implementation of queue (using an array).

```
#include<stdio.h>
#define MAX 5
struct queue // structure of queue
{
    int data[MAX];
    int front, rear;
};
void initQ(struct queue *q) //initialization
{
    q->front = q->rear = -1;
}
int isEmptyQ(struct queue *q)
{
    if(q->front == q->rear)
    return 1;
    else
    return 0;
}
int isFullQ(struct queue *q)
{
    if (q -> rear == MAX - 1)
    return 1;
    else
    return 0;
}
```

```
void insertQ(struct queue *q, int x)
{
   q ->data[++(q -> rear)] = x;
}
int deleteQ(struct queue *q)
{
   return (q -> data[++(q -> front)]);
}
void display(struct queue *q)
{
   int i;
   printf("\nQueue contents are:\t");
   for(i = q -> front + 1 ;i <= q -> rear; i++)
   printf(" %d\t", q -> data[i]);
}
/* main program */
void main()
{
   struct queue q1;
   int ch,x;
   initQ(&q1);
   do
   {
      printf("\n1-Insert\n2-Delete\n");
      printf("Enter your choice\n");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1: if(isFullQ(&q1))
                 printf("Queue is Full\n");
                 else
                 {
                   printf("Enter element to be insert\n");
                   scanf("%d",&x);
                   insertQ(&q1,x);
                   display(&q1);
                 }
                 break;
```

```
            case 2: if(isEmptyQ(&q1))
                    printf("Queue is Empty\n");
                    else
                    {
                      printf("Deleted element is %d\n",deleteQ(&q1));
                      display(&q1);
                    }
                  break;
          }
    }while(ch>0 && ch <3);
    }
```

Output:

```
1-Insert
2-Delete
Enter your choice
1
Enter element to be insert
4
Queue contents are :   4
1-Insert
2-Delete
Enter your choice
1
Enter element to be insert
6
Queue contents are :   4      6
1-Insert
2-Delete
Enter your choice
1
Enter element to be insert
7
Queue contents are :   4      6      7
1-Insert
2-Delete
Enter your choice
2
Deleted element is 4
Queue contents are :   6      7
1-Insert
2-Delete
```

```
Enter your choice
1
Enter element to be insert
8
Queue contents are :   6      7      8
1-Insert
2-Delete
Enter your choice
2
Deleted element is 6
Queue contents are :   7      8
1-Insert
2-Delete
Enter your choice
2
Deleted element is 7
Queue contents are :   8
1-Insert
2-Delete
Enter your choice
2
Deleted element is 8
Queue contents are :
1-Insert
2-Delete
Enter your choice
2
Queue is Empty
```

Program 5.2: Static implementation of queue (using an array) with operations.

```
/* header file “stqueue.h”*/
#define MAX 10
struct queue
{
   int a[MAX];
   int f,r;
};
void initq(struct queue *);
void addq(struct queue*,int);
int delq(struct queue *);
int peekq(struct queue *);
int isemptyq(struct queue *);
int isfullq(struct queue *);
```

```
/*Program Main File*/
   #include <stdio.h>
   #include "stqueue.h"
   int main()
   {
      struct queue q;
      int opt,x,n,e;
      initq(&q);
      do
      {
         printf("1.add\n2.del\n3.Peek\n4.isEmpty\n5.isfull\n6.Exit\n");
         scanf("%d",&opt);
         switch(opt)
         {
            case 1: printf("enter element to push\n");
                    scanf("%d",&x);
                    addq(&q,x);
                    break;
            case 2: e=delq(&q);
                    if(e!=0)
                    {
                      printf("deleted element is %d\n",e);
                    }
                    else
                    printf("queue is underflow\n");
                    break;
            case 3: e=peekq(&q);
                    if(e==1)
                    printf("list is empty\n");
                    else
                    printf("element in front is %d\n",e);
                    break;
            case 4: e=isemptyq(&q);
                    if(e==1)
                    printf("list is empty\n");
                    else
                    printf("list is not empty\n");
                    break;
            case 5: e=isfullq(&q);
                    if(e==1)
                    printf("list is full\n");
```

```
                  else
                  printf("list is not full\n");
                  break;
    }
}while(opt!=6);
return 0;
}
void initq(struct queue *pq)
{
    pq->f=pq->r=-1;
}
void addq(struct queue *pq,int x)
{
    if(isfullq(pq))
{
    printf("queue overflow\n");
    return;
}
else
pq->a[++pq->r]=x;
}
int delq(struct queue *pq)
{
    if(isemptyq(pq))
    {
        //printf("queue underflow\n");
        return 0;
    }
else
return pq->a[++pq->f];
}
int isemptyq(struct queue *pq)
{
    if(pq->f==pq->r)
      return 1;
    else
      return 0;
}
int isfullq(struct queue *pq)
{
    if(pq->r==MAX-1)
```

```
        return 1;
     else
        return 0;
  }
  int peekq(struct queue *pq)
  {
     if(!isemptyq(pq))
     return pq->a[pq->f+1];
  }
```

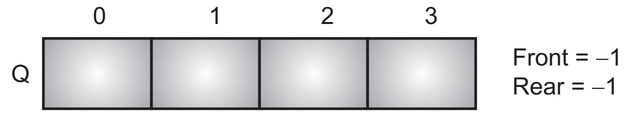
Output:

```
1.add
2.del
3.Peek
4.isEmpty
5.isfull
6.Exit
4
list is empty
1.add
2.del
3.Peek
4.isEmpty
5.isfull
6.Exit
5
list is not full
1.add
2.del
3.Peek
4.isEmpty
5.isfull
6.Exit
1
enter element to push
24
1.add
2.del
3.Peek
4.isEmpty
5.isfull
6.Exit
```

```
4
list is not empty
1.add
2.del
3.Peek
4.isEmpty
5.isfull
6.Exit
1
enter element to push
56
1.add
2.del
3.Peek
4.isEmpty
5.isfull
6.Exit
3
element in front is 24
1.add
2.del
3.Peek
4.isEmpty
5.isfull
6.Exit
2
deleted element is 24
1.add
2.del
3.Peek
4.isEmpty
5.isfull
6.Exit
3
element in front is 56
1.add
2.del
3.Peek
4.isEmpty
5.isfull
6.Exit
```

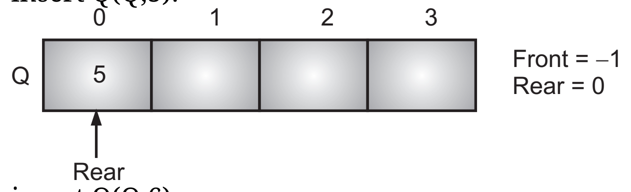
Example:

- Let us consider the example which uses the all above functions.
  Let Q be an empty queue with front = rear = – 1.
  Let max = 4:

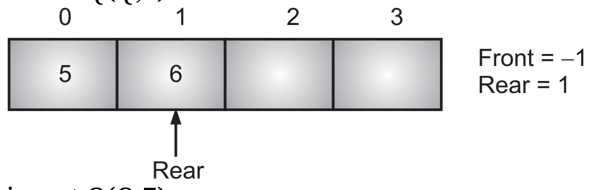


Consider the following statements:

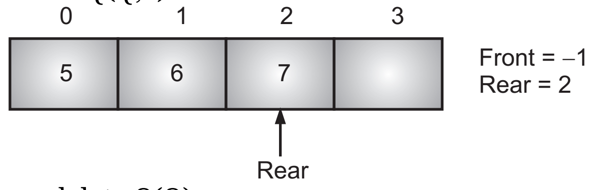
(i) insert Q(Q,5):



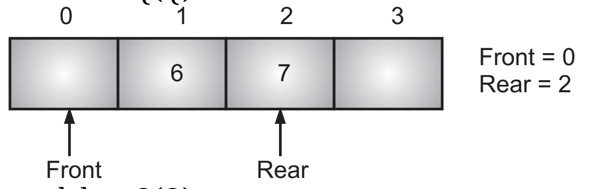
(ii) insert Q(Q,6):



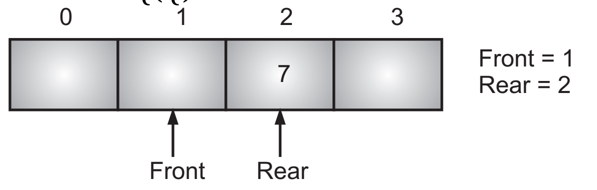
(iii) insert Q(Q,7):



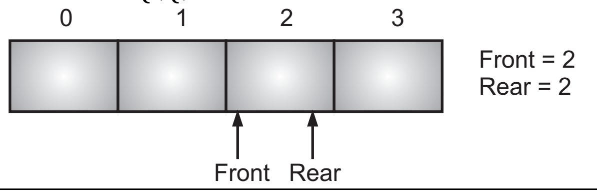
(iv) x = delete Q(Q):



(v) x = delete Q(Q):



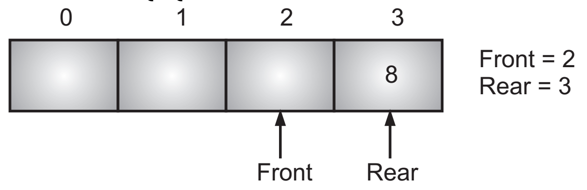
(vi) x = delete Q(Q):



(vii) x = delete Q(Q):

This generates "Queue Empty" error as front = rear = 2.

(viii) insert Q(Q,8):



(ix) insert Q(Q,9).

- This generates "Queue full" message. Is queue really full? Here, we have only one element in the queue, but "Queue full" message is generated. This is drawback in array representation so solution is to have linked list representation of queue or circular queue.

Advantage and Disadvantage of representing Queue using an Array:

1. Implementation of a queue using an array is easy and simple.
2. It is possible to store only a fixed number of elements in the queue.
3. Wastage of memory space can occur. For example, if the size of the queue is 15 but the number of elements stored in the queue may be just 5 then the space allocated for the 10 elements are a mere waste.

5.4.2 Dynamic Implementation of Queue (Using Linked List)

[April 15]

- We have discussed queue are represent using sequential organization i.e. array, but this representation has following drawbacks:
  1. Fixed size queue do not give flexibility to user to dynamically exceed the maximum size.
  2. Poor memory utilization.
  3. Updation of front and rear.
- The elements in the linked list are allocated dynamically, hence it can grow as long as there is sufficient memory available for dynamic allocation.
- The Fig. 5.4 shows the linked queue.

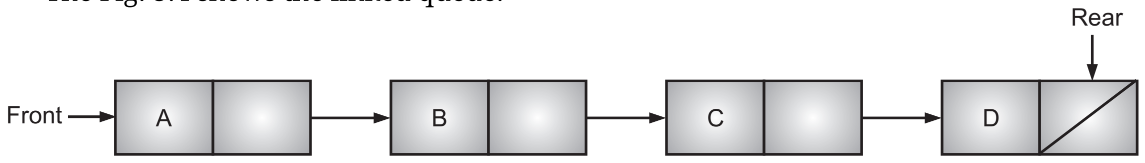
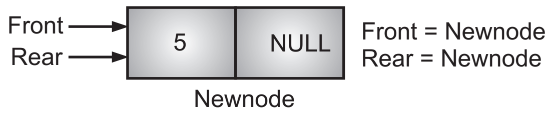


Fig. 5.4: Linked Queue

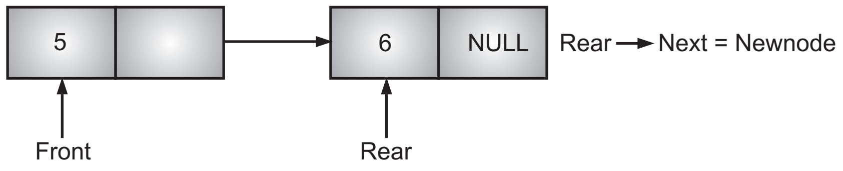
- **Data Structure for Queue using Linked List:**

```
struct queue
{
   int data;
   struct queue *next;
} *front, *rear;
```

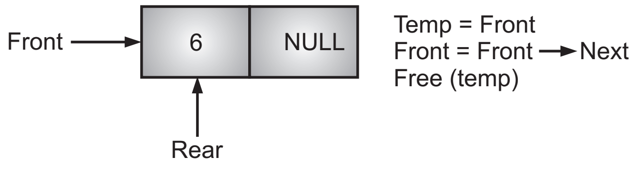
- The various operation on queue are as follows:
  (i) Initialization of queue:
      front = rear = null
      List is null
  (ii) Insert 5:



  (iii) Insert 6:



  (iv) delete:



  (v) delete:
      front = rear = NULL

Program 5.3: Dynamic implementation of queue (using linked list).

```
#include<stdio.h>
#include<stdlib.h>
struct queue          // structure of queue
{
int data;
struct queue *next;
}*front,*rear;
void insertQ(int n)
{
   struct queue *temp;
   temp = (struct queue *)malloc(sizeof(struct queue));
   temp -> data = n;
   temp -> next = NULL;
   if(front == NULL)
      rear = front = temp;
   else
   {
      rear -> next = temp;
```

```
        rear = temp;
    }
}
int deleteQ()
{
    int x;
    struct queue *temp = front;
    x = front -> data;
    if (front==rear)
        front = rear = NULL;
    else
        front= front -> next;
    free(temp);
    return(x);
}
void display()
{
    struct queue *temp = front;
    printf("\nQueue contents are :\t");
    while(temp)
    {
        printf("%d\t", temp -> data);
        temp = temp -> next;
    }
}
void main()
{
int ch,x;
do
{
    printf("\n1-Insert\n2-Delete\n");
    printf("Enter your choice\n");
    scanf("%d",&ch);
switch(ch)
    {
    case 1:printf("Enter element to be insert\n");
         scanf("%d",&x);
         insertQ(x);
         display();
         break;
```

```
      case 2: if(front == NULL)
             printf("Queue is Empty\n");
            else
            {
          printf("Deleted element is %d\n",deleteQ());
          display();
           }
        break;
      }
   }while(ch>0 && ch <3);
   }
```

Output:

```
   1-Insert
   2-Delete
   Enter your choice
   1
   Enter element to be insert
   3
   Queue contents are :   3
   1-Insert
   2-Delete
   Enter your choice
   1
   Enter element to be insert
   5
   Queue contents are :   3      5
   1-Insert
   2-Delete
   Enter your choice
   2
   Deleted element is 3
   Queue contents are :   5
   1-Insert
   2-Delete
   Enter your choice
   2
   Deleted element is 5
   Queue contents are :
   1-Insert
   2-Delete
   Enter your choice
   2
   Queue is Empty
```

Advantage and Disadvantage of Linked List representtaion of Queue:

1. No wastage of memory space.
2. Insertion and deltion are easy to perfrom.
3. Size of queue is not fixed. That means any number of elements can be placed in queue.

5.4.3 Difference between Static and Dynamic Representations/ Implementations of Queue

- In implementing the queues using arrays is called as static implementation. In implementing queues using linked list called ad dynamic implementation.
- A static queue is a collection of one or more elements arranged in memory in a contiguous fashion. A dynamic queue is a collection of one or more elements arranged in memory in a discontinuous fashion.
- In static queue, only one and single type of information is stored while dynamic queue also stored the address for the next node along with it's content.
- A static queue is a queue of fixed size implemented using array. A dynamic queue is a queue of variable size implemented using linked list.
- Static queue is index based while dynamic queue is reference (pointer) based.
- In static implementation of queue, the memory allocation is fixed so we cannot allocate or deallocate memory as per need. In dynamic implementation we can allocate or deallocate memory as per need.
- Implementation of a queue using an array is easy and simple than dynamic queue.

5.5 TYPES OF QUEUE

- There are different types of queues namely, Linear queue, Circular queue, Double ended queue (Dequeue) and Priority Queue as explained below:

1. Linear Queue:

- A queue is a linear data structure in which data can only be inserted at one end, called the rear, and deleted from the other end, called the front.
- These restrictions ensure that the data are processed through the queue in the order in which they are received.
- In simple words, queue is First In First Out (FIFO) structure. In linear queue elements are arranged in sequential mode such that front position is always be less than or equal to the rear position.
- One of the best day-to-day life examples for a queue is line at library counter, where the first student in line is usually the first to be checked out, (See Fig. 5.5). Other examples of queue are bus stand line, bus ticket line, rail-way reservation counter line etc.

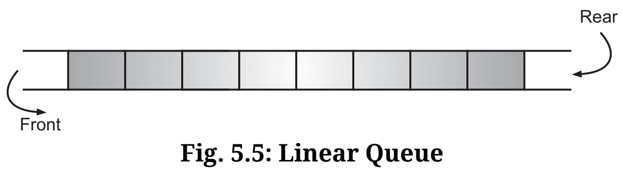


Fig. 5.5: Linear Queue

2. Circular Queue:

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as Ring Buffer.
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.

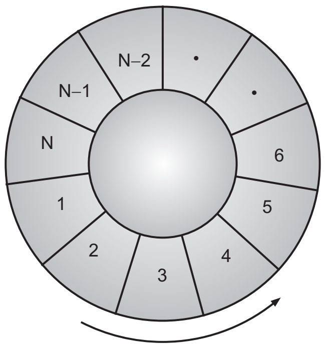


Fig. 5.6: Circular Queue

3. Dequeue (Double Ended Queue): [S-18]

- In Double Ended Queue, insert and delete operation can be occur at both ends that is front and rear of the queue.

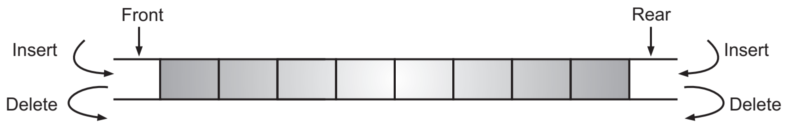


Fig. 5.7: Dequeue

4. Priority Queue:

- Priority queue contains data items which have some preset priority.
- While removing an element from a priority queue, the data item with the highest priority is removed first.

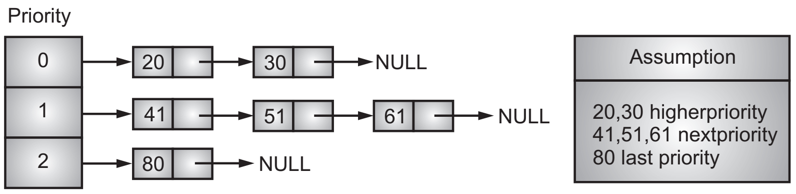


Fig. 5.8: Priority Queue

- In a priority queue, insertion is performed in the order of arrival and deletion is performed based on the priority.

5.5.1 Linear Queue

- A queue is "an ordered list in which all insertions take place at one end called the rear while all deletions take place at the other end, the front".
- Following are basic features of linear queue:
  1. Like stack, queue is also an ordered list of elements of similar data types.
  2. Queue is a FIFO structure.
  3. Two pointers called FRONT and REAR are used for queues. Here, FRONT represents the end at which deletion can be performed and REAR represents the end at which insertion can be performed as shown in Fig. 5.9.

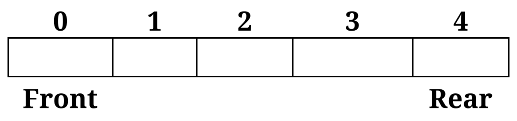


Fig. 5.9

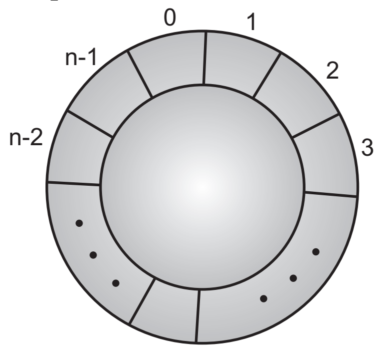
- Consider a queue of size 5. Front and rear are two pointers which are associated with queue. The initial value of front and rear is – 1, For implementation of queue, we are considering here that initial value of front = – 1 and rear = – 1 which represent that queue is initially empty.
- The user can also assign 0 instead of – 1. However, the – 1 is suitable because we are implementing this problem with arrays and an array element counting begins always with zero.
- Linear Queue is explained in previous sections.

5.5.2 Circular Queue [April 15]

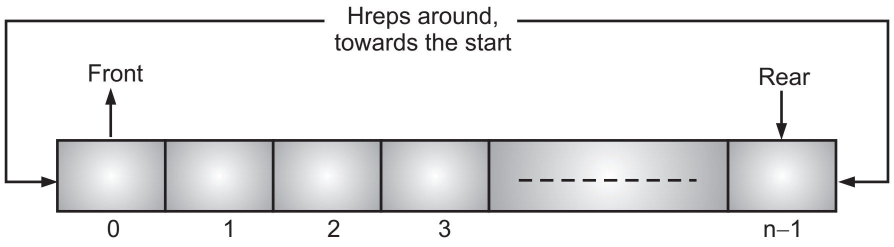
- So far we have discussed linear queues. Linear queue using array has certain drawbacks. There is possibility that the queue is reported full even though slots of the queue are empty i.e. actually queue is not full.
- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

5.5.2.1 Definition

- The technique which essentially allows the queue to wrap around upon reaching the end of array, eliminates these drawbacks.
- Such a technique which allows queue to wrap around from end to start is called circular queue; which is more efficient queue representation using array Q as circular.
- Suppose an array Q is of size n. Now is we go on adding elements in queue, we may reach the location n – 1. If queue is not circular, we cannot add more elements when end is reached, even though there are free locations at the front side of array.
- In such case, these slots would be filled by new elements added to the queue; rather than signaling a error as queue is full.
- Hence, circular queue allows us to continue adding elements, even though rear = n – 1. The queue is said to be full only when there are n elements in the queue.
- Fig. 5.10 shows the pictorial representation of a circular queue.



(a) Logical View



(b) Physical View

Fig. 5.10: Circular Queue

- Consider circular queue Q of size n, we have studied the functions of different operations on linear queue using array. Few changes are required. In circular queue, when rear = n – 1 and new element is to be added, the rear should be set to 0, i.e.

```
if(rear == n – 1)
   rear = 0;  ⇒ rear = (rear + 1) % n
else rear = rear + 1;
```

- Initially, both front and rear are set to 0. Front will always have value one less than the actual front.

Applications of a Circular Queue:

1. **Memory Management:** Circular queue is used in memory management.
2. **Process Scheduling:** A CPU uses a queue to schedule processes.
3. **Traffic Systems:** Queues are also used in traffic systems.

5.5.2.2 Operations of Circular Queue [Oct. 16, April 17]

- Various operations on circular queue are given below:
  1. **Enqueue:** Adding an element in the queue if there is space in the queue.
  2. **Dequeue:** Removing elements from a queue if there are any elements in the queue
  3. **Front:** Get the first item from the queue.
  4. **Rear:** Get the last item from the queue.
  5. **isEmpty/isFull:** checks if the queue is empty or full.

Advantages of Circular Queue:

1. Circular Queue consumes less space or memory compared to linear queue, as if we delete any element that position is used later, because it is circular.
2. It avoids shifting of queue elements.
3. Circular queues are used in many applications such as memory management, CPU scheduling, traffic system.

5.5.2.3 Circular Queue Representations/Implementations

- A circular queue is a queue in which all nodes are treated as circular such that the first node follows the last node. There are two ways for implementation of circular queue, i.e. using arrays and using linked list.

1. Using Array: **[April 17]**

- In arrays the range of a subscript is 0 to n–1 where n is the maximum size.
- To make the array as a circular array by making the subscript 0 as the next address of the subscript n–1 by using the formula,

  subscript = (subscript +1) % maximum size.
- In circular queue the front and rear pointer are updated by using the above formula.
- Fig. 5.11 shows circular queue using arrays.

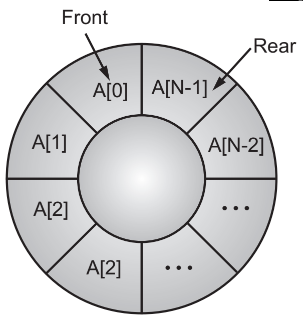


Fig. 5.11

Program 5.4: Program to implements circular queue using array.

```
#include <stdio.h>
#define MAX 5
int main()
{
   char ch;
   int i,queue[MAX],front=-1,rear=-1;
   setbuf(stdout, NULL);
   do
   {
      if((front == rear + 1) || (front == 0 && rear == MAX-1))
      {
         printf("\nQueue is Full!!");
         return 0;
      }
      else
      {
         if(front == -1)
            front = 0;
         rear=(rear+1)%MAX;
         printf("\nEnter element to Insert:");
         scanf("%d",&queue[rear]);
      }
      printf("\nAfter Insertion position of Front=%d and
                                                    Rear=%d",front,rear);
      printf("\nDo you want to insert more elements?");
      scanf(" %c",&ch);  //use a space before %c to clear stdin
   }while(ch=='y'||ch=='Y');
   printf("\n**** Elements in Queue*****\n");
   for(i=front; i!=rear; i=(i+1)%MAX)
```

```
      printf("%d  ",queue[i]);
      printf("%d  ",queue[i]);
      return 0;
   }
```

Output:

```
Enter element to Insert:3
After Insertion position of Front=0 and Rear=0
Do you want to insert more elements?y
Enter element to Insert:5
After Insertion position of Front=0 and Rear=1
Do you want to insert more elements?y
Enter element to Insert:4
After Insertion position of Front=0 and Rear=2
Do you want to insert more elements?n
**** Elements in Queue*****
3  5  4
```

2. Using Linked List:

- In a linked list circular queue, the rear node always has the reference of the front node. Even if the front node is removed the rear node has the reference of the new front node.
- Fig. 5.12 shows circular queue with linked list.

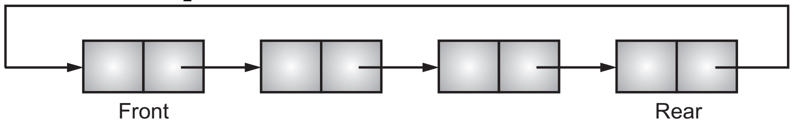


Fig. 5.12

Program 5.5: Program to implements circular queue using linked list.

```
#include <stdio.h>
#include <process.h>
#include <malloc.h>
struct node
{
   int info;
   struct node *ptr;
}*front,*rear,*temp,*front1;
int main()
{
   int data;
   char ch;
   front = rear = NULL;
   setbuf(stdout, NULL);
```

```
    do
    {
       printf("\nEnter element to Insert:\n");
       scanf("%d",&data);
       temp = (struct node *)malloc(sizeof(struct node));
       temp->ptr = NULL;
       temp->info = data;
       if (rear == NULL)
       {
          rear=temp;
          front = rear;
       }
       else
       {
          rear->ptr = temp;
          rear = temp;
       }
       printf("\nDo you want to insert more elements?");
       scanf(" %c",&ch);  //use a space before %c to clear stdin
    }while(ch=='y'||ch=='Y');
    printf("\n**** Elements in Circular Queue*****\n");
    if ((front == NULL) && (rear == NULL))
    {
       printf("Queue is empty");
       return 0;
    }
    while (front != rear)
    {
       printf("%d ", front->info);
       front = front->ptr;
    }
    printf("%d", front->info);
    return 0;
 }
```

Output:

```
Enter element to Insert:
10
Do you want to insert more elements?y
Enter element to Insert:
20
Do you want to insert more elements?n
**** Elements in Circular Queue*****
10 20
```

Example: To find the insertion and deletion operations on circular queue where max = 4. **[Oct. 17]**

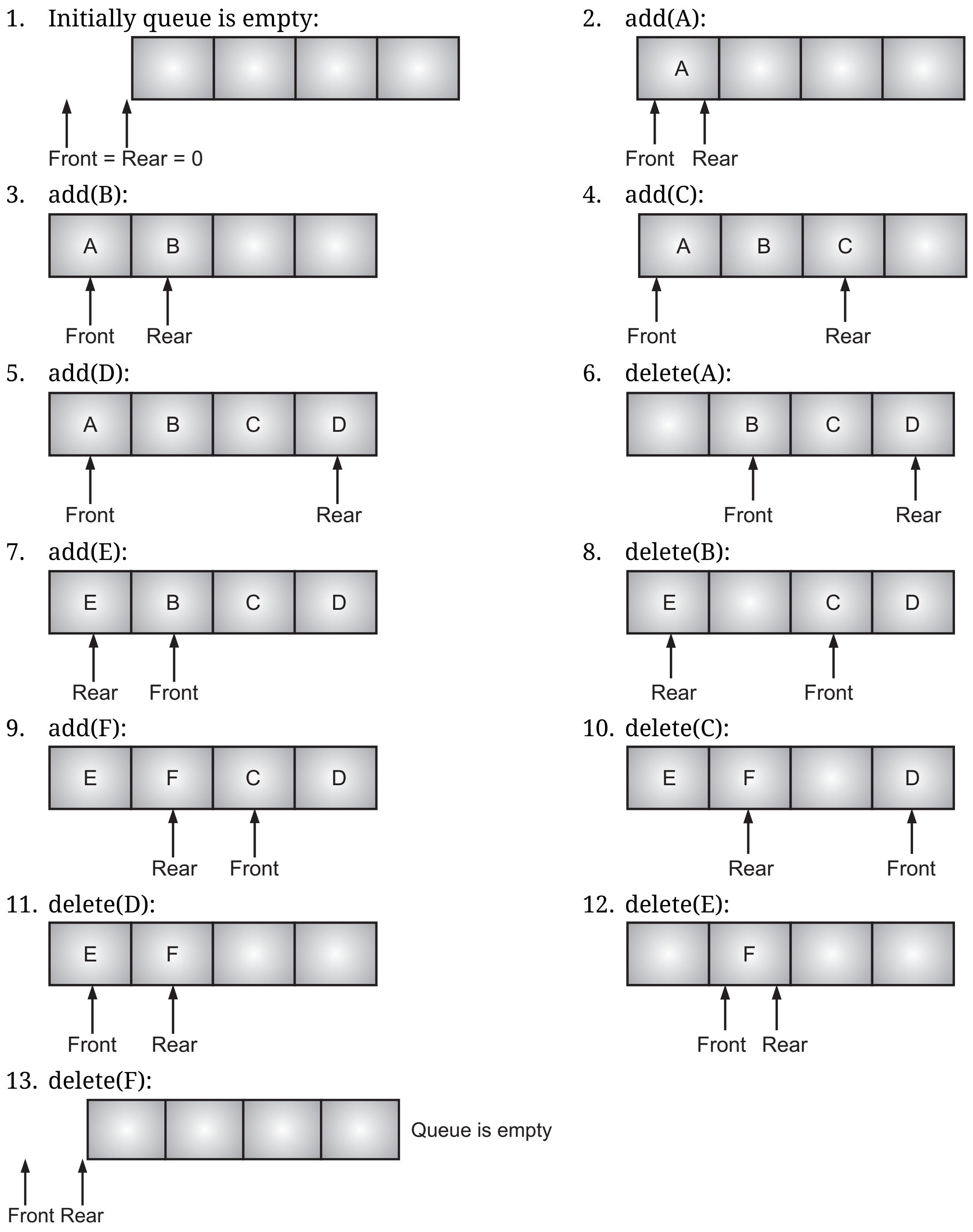


Fig. 5.13: Operations on Circular Queue

Program 5.6: Program for Enqueue operation in circular queue.

```
#include <stdio.h>
#define MAX 5
int main()
{
   char ch;
   int i,queue[MAX],front=-1,rear=-1;
   setbuf(stdout, NULL);
   do
   {
      if((front == rear + 1) || (front == 0 && rear == MAX-1))
      {
         printf("\nQueue is Full!!");
         return 0;
      }
      else
      {
         if(front == -1)
            front = 0;
         rear=(rear+1)%MAX;
         printf("\nEnter element to Insert:");
         scanf("%d",&queue[rear]);
      }
      printf("\nAfter Insertion position of Front=%d and
                                                    Rear=%d",front,rear);
      printf("\nDo you want to insert more elements?");
      scanf(" %c",&ch);  //use a space before %c to clear stdin
   }while(ch=='y'||ch=='Y');
   printf("\n**** Elements in Queue*****\n");
   for(i=front; i!=rear; i=(i+1)%MAX)
      printf("%d  ",queue[i]);
   printf("%d  ",queue[i]);
   return 0;
}
```

Output:

```
Enter element to Insert:10
After Insertion position of Front=0 and Rear=0
Do you want to insert more elements?y
Enter element to Insert:20
After Insertion position of Front=0 and Rear=1
Do you want to insert more elements?n
**** Elements in Queue*****
10  20
```

Program 5.7: Program for dequeue operation in circular queue.

```
#include <stdio.h>
#define MAX 5
int main()
{
   char ch;
   int queue[MAX]={10,20,30,40,50}; //queue with some elements
   int i,front=0,rear=MAX-1;
   setbuf(stdout, NULL);
   printf("\n**** Elements in Queue*****.\n");
   for(i=front; i!=rear; i=(i+1)%MAX)
            printf("%d  ",queue[i]);
   printf("%d  ",queue[i]);
   printf("\nDo you want to delete element?");
   scanf(" %c",&ch);  //use a space before %c to clear stdin
   while(ch=='y'||ch=='Y')
   {
      if(front==-1)
      {
         printf("\nQueue is Empty!!");
         return 0;
      }
      else
      {
         printf("Deleted Element is %d",queue[front]);
         if (front == rear)
         {
            front = -1;
            rear = -1;
         }
         else
            front=(front +1)%MAX;
      }
      printf("\nAfter Deletion position of Front=%d and
                                                   Rear=%d ",front,rear);
      printf("\n\nDo you want to delete element?");
      scanf(" %c",&ch);  //use a space before %c to clear stdin
   }
   if(front==-1)
      printf("\nQueue is Empty!!");
   else
   {
      printf("\n**** Elements in Queue*****.\n");
      for(i=front; i!=rear; i=(i+1)%MAX)
         printf("%d  ",queue[i]);
      printf("%d  ",queue[i]);
   }
}
```

Output:

```
   **** Elements in Queue*****.
   10  20  30  40  50
   Do you want to delete element?y
   Deleted Element is 10
   After Deletion position of Front=1 and Rear=4
   Do you want to delete element?y
   Deleted Element is 20
   After Deletion position of Front=2 and Rear=4
   Do you want to delete element?n
   **** Elements in Queue*****.
   30  40  50
```

5.5.3 Priority Queue **[April 15, 16, 17, 18, Oct. 17]**

- The priority queue is a collection of elements in which each element is arranged in the queue based on its priority.
- A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed comes from the following rules:
  1. An element of higher priority is processed before any element of lower priority.
  2. Two elements with the same priority are processed according to the order in which they were added to the queue.
- There are two types of priority queue namely ascending priority queue and descending priority queue.
- An ascending priority queue is a collection of items in which items can be inserted arbitrarily and from which only the smallest item can be removed.
- A descending priority queue is a collection of items in which items can be inserted arbitrarily and from which only the largest item can be removed.

5.5.3.1 Definition **[April 16, 17, 18, Oct. 16]**

- A priority queue is a collection in which items can be added at any time, but the only item that can be removed is the one with the highest priority.
- So far we discuss queue with FIFO operations. In some cases, the elements are inserted and deleted from any position based on intrinsic ordering rather than strict ordering and determine which element get removed first.

- Priority queues are the queues in which we can use intrinsic ordering of the elements. Items can be inserted in any order in a priority queue but when an item is removed from the priority queue, it is always the one with higher priority.

Definition:

- A priority queue is a data structure in which each element has been assigned a value called the priority of the element and an element can be inserted (or) deleted not only at the ends but at any position on the queue.
- Fig. 5.14 shows the following rules are applied to maintain a priority queue:
    1. The elements with a higher priority is processed before any element of lower priority.
    2. If there were elements with the same priority then the elements are processed according to the order in which they are added to the queue.

Representation of Priority Queue using Linked List in Memory:

- A Node of three fields i.e. it contains data, priority and pointer to next. Fig. 5.14 shows the priority queue of four elements where node B and node C has same priority, A has higher priority.

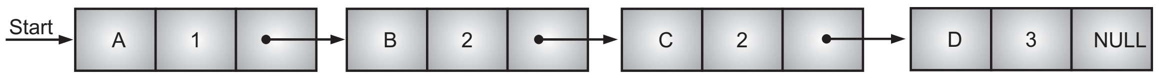


Fig. 5.14: Priority Queue

Examples of Priority Queue:

- Priority queues are used for implementing job scheduling by the operating system where jobs with higher priority are to be processed first.
- A list of patients in an emergency room, each patient might be given a ranking that depends on the severity of the patient's illness.
- A list of background jobs on a multi-user computer, each background job is given a priority level.

5.5.3.2 Operations on a Priority Queue

- Various operations on priority queue are:
    1. **Initialize:** Initializes a priority queue as empty queue.
    2. **IsEmpty:** To check priority queue is empty or not.
    3. **IsFull:** To check priority queue is full or not.
    4. **Insert:** If not Full.
    5. **Delete:** if not empty; delete the item with higher priority.

5.5.3.3 Array Implementation of a Priority Queue

- If array is used to stored elements of a priority queue, then insertion is easy but deletion is difficult. Since, for deletion, to find highest priority element, we have to search entire array.

- Each element in an array can have following structure.

```
struct data
{
 int item;
 int priority;
 int order;
}
```

where, priority is priority number of the element and order is the order in which the element has been added to the queue.

Program 5.8: To implement priority queue using array.

```
#include<stdio.h>
#include<string.h>
#define MAX 10
struct data
{
    char job[20] ;
    int prno ;
    int ord ;
} ;
struct pque
{
    struct data d[MAX] ;
    int front ;
    int rear ;
} ;
void initPQ ( struct pque *pq )
{
     pq -> front = pq -> rear = -1 ;
}
int isFullPQ(struct pque *pq)
{
  if ( pq -> rear == MAX - 1 )
         return 1;
  else
   return 0;;
}
int isEmptyPQ(struct pque *pq)
{
  if ( pq -> front == -1 )
         return 1;
  else
   return 0;
}
```

```
void insertPQ( struct pque *pq, struct data dt )
{
    struct data temp ;
    int i, j ;
    pq -> rear++ ;
    pq -> d[pq -> rear] = dt ;
    if ( pq -> front == -1 )
        pq -> front = 0 ;
    for ( i = pq -> front ; i <= pq -> rear ; i++ )
    {
        for ( j = i + 1 ; j <= pq -> rear ; j++ )
        {
            if ( pq -> d[i].prno > pq -> d[j].prno )
            {
                temp = pq -> d[i] ;
                pq -> d[i] = pq -> d[j] ;
                pq -> d[j] = temp ;
            }
            else
            {
                if ( pq -> d[i].prno == pq -> d[j].prno )
                {
                    if ( pq -> d[i].ord > pq -> d[j].ord )
                    {
                        temp = pq -> d[i] ;
                        pq -> d[i] = pq -> d[j] ;
                        pq -> d[j] = temp ;
                    }
                }
            }
        }
    }
}
struct data  deletePQ ( struct pque *pq )
{
    struct data  t = pq -> d[pq -> front] ;
    if ( pq -> front == pq -> rear )
         pq -> front = pq -> rear = -1 ;
    else
         pq -> front++ ;
    return  t ;
}
```

```
void main( )
{
    struct pque q ;
    struct data temp;
    int i=0,ch;
    initPQ ( &q ) ;
    do
    {
      printf ( "\n1-Insert\n2-Delete\nEnter your choice\n");
      scanf("%d",&ch);
      switch(ch)
      {
         case 1: if(isFullPQ(&q))
                 printf ("\nQueue is full." ) ;
                 else
                 {
                    printf("Lower priority number
                                           indicates higher priority\n" ) ;
                    printf ( "Job     Priority\n" ) ;
                    scanf ( "%s%d", temp.job, &temp.prno ) ;
                          temp.ord = i++ ;
                    insertPQ( &q, temp ) ;
                 }
                 break;
         case 2: if(isEmptyPQ(&q))
                 printf ( "\nQueue is Empty." ) ;
                 else
                 {
                    temp  = deletePQ( &q ) ;
                    printf("%s\t%d\n",temp.job, temp.prno);
                 }
                 break;
      }
    }while(ch>0 && ch<3);
}
```

Output:

```
1-Insert
2-Delete
Enter your choice
1
Lower priority number indicates higher priority
```

```
Job     Priority
2          4
1-Insert
2-Delete
Enter your choice
1
Lower priority number indicates higher priority
Job     Priority
4         5
1-Insert
2-Delete
Enter your choice
1
Lower priority number indicates higher priority
Job     Priority
3     1
1-Insert
2-Delete
Enter your choice
2
3  1
1-Insert
2-Delete
Enter your choice
2
2  4
1-Insert
2-Delete
Enter your choice
2
4  5
1-Insert
2-Delete
Enter your choice
2
Queue is Empty.
```

- The operations performed in a queue are similar to the queue except that the insertion and deletion elements made in it.
  1. **Insert Operation:** Elements can be inserted in any order, but are arranged in order of their priority value in the queue.

     For example: Fig. 5.15 shows a priority queue.

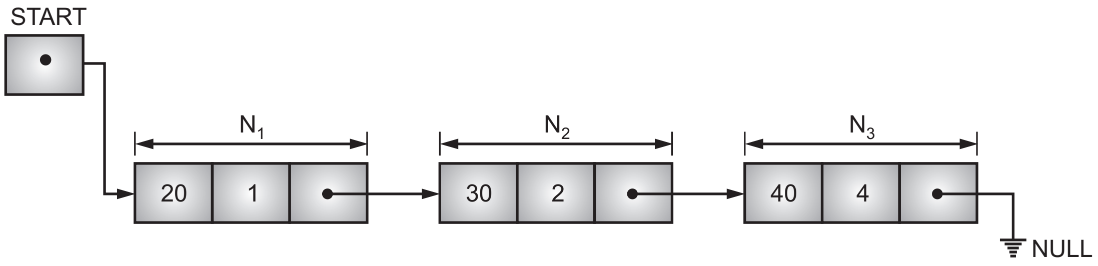


Fig. 5.15

- Suppose one wants to add new node N_4 having priority value 5 which is shown in Fig. 5.16.

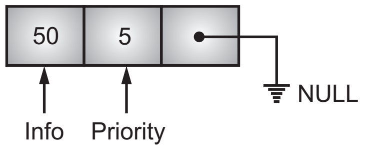


Fig. 5.16

- Since, new node has priority 5 then we insert this node at the end of existing queue as its priority is less compare to existing elements (See Fig. 5.17).

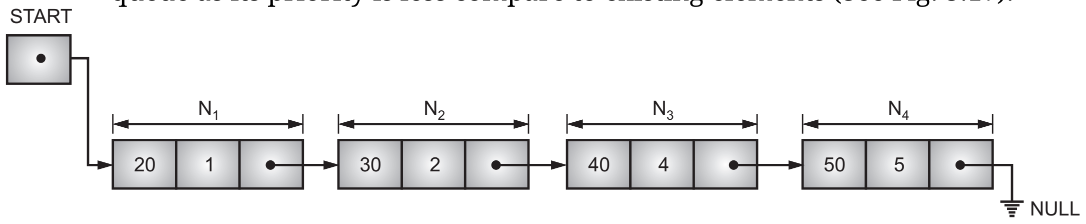


Fig. 5.17

2. **Delete operation:** The elements are deleted from the queue in the order of their priority. A highest priority element removed first. The elements with the same priority are given equal importance and processed accordingly.

   For example: Suppose we have a list of priority queue as shown in Fig. 5.18.

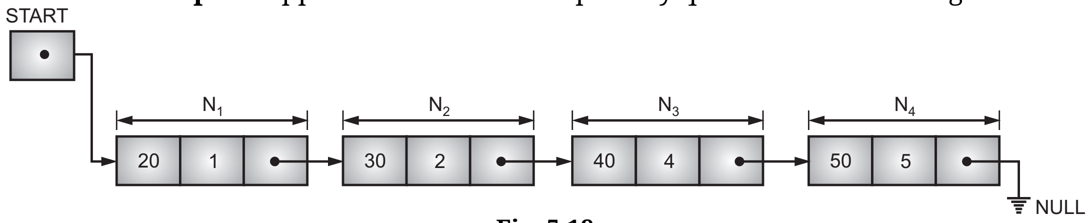


Fig. 5.18

- Fig. 5.19 shows priority queue after deletion node N_1.

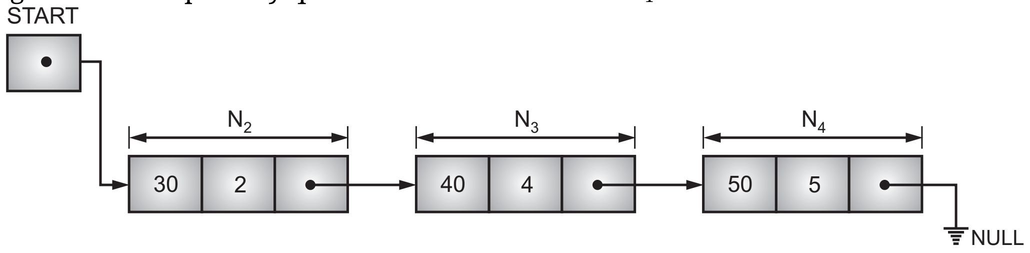


Fig. 5.19

Program 5.9: Program for implementation of priority queue.

```
/* Insertion in Priority Queue */
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data, priority;
   struct node *next;
}*start;  //start is declared
int main()
{
   int data, pr;
   setbuf(stdout, NULL);
   char ch;
   do
   {
      printf("\nEnter element and its priority: ");
      scanf("%d%d",&data,&pr); //input from user
      struct node *temp, *t;
      temp = (struct node *)malloc(sizeof(struct node));
      temp->data=data;
      temp->priority=pr;
      temp->next=NULL;
      if(start==NULL)
         start = temp;
      else if(start->priority>pr)
      {
         temp->next=start;
         start=temp;
      }
      else
      {
         t=start;
         while(t->next!=NULL && (t->next)->priority<=pr )
            t=t->next;
         temp->next=t->next;
         t->next=temp;
      }
      printf("\nDo you want to insert more elements?");
      scanf(" %c",&ch);  //use a space before %c to clear stdin
   }while(ch=='y'||ch=='Y');
```

```
    printf("\n**** Elements in Circular Queue*****\n");
    struct node *temp1 = start;
    while(temp1!=NULL)
    {
       printf("\nData=%d Priority=%d ",temp1->data, temp1->priority);
       temp1=temp1->next;
    }
    return 0;
 } //end of Main
```

Output:

```
Enter element and its priority: 10 4
Do you want to insert more elements?y
Enter element and its priority: 20 3
Do you want to insert more elements?y
Enter element and its priority: 30 1
Do you want to insert more elements?n
**** Elements in Circular Queue*****
Data=30 Priority=1
Data=20 Priority=3
Data=10 Priority=4
```

Program 5.10: Program for deletion in priority queue.

```
#include<stdio.h>
#include<stdlib.h>
void insert();
void delete();
void display();
struct node
{
   int data, priority;
   struct node *next;
}*start;  //start is declared
int main()
{
   setbuf(stdout, NULL);
   int ch;
   printf("\n*** Priority Queue Menu ***");
   printf("\n1.Insertion \n2.Deletion \n3.Display \n4.Exit");
```

```
    while(1)
    {
    printf("\nEnter the Choice(1.Insertion 2.Deletion 3.Display 4.Exit):");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1:
            insert();
            break;
        case 2:
            delete();
            break;
        case 3:
            display();
            break;
        case 4:
            exit(0);
        default:
            printf("\nWrong Choice!!");
        }
    }
    return 0;
}
void insert()
{
    int data, pr;
    printf("\nEnter element and its priority: ");
    scanf("%d%d",&data,&pr); //input from user
    struct node *temp, *t;
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data=data;
    temp->priority=pr;
    temp->next=NULL;
    if(start==NULL)
        start = temp;
    else if(start->priority>pr)
    {
        temp->next=start;
        start=temp;
    }
```

```
      else
      {
         t=start;
         while(t->next!=NULL && (t->next)->priority<=pr )
            t=t->next;
         temp->next=t->next;
         t->next=temp;
      }
   }
   void delete()
   {
      if(start!=NULL)
      {
         printf("\nRemoved Element: %d",start->data);
         start = start->next;
      }
      else
         printf("\nQueue is Empty");
   }
   void display()
   {
      printf("\n**** Elements in Circular Queue*****\n");
      struct node *temp1 = start;
      while(temp1!=NULL)
      {
         printf("\nData=%d Priority=%d ",temp1->data, temp1->priority);
         temp1=temp1->next;
      }
   }
```

Output:

```
   *** Priority Queue Menu ***
   1.Insertion
   2.Deletion
   3.Display
   4.Exit
   Enter the Choice(1.Insertion 2.Deletion 3.Display 4.Exit):1
   Enter element and its priority: 10 3
   Enter the Choice(1.Insertion 2.Deletion 3.Display 4.Exit):1
   Enter element and its priority: 20 1
   Enter the Choice(1.Insertion 2.Deletion 3.Display 4.Exit):1
   Enter element and its priority: 30 2
```

```
Enter the Choice(1.Insertion 2.Deletion 3.Display 4.Exit):3
**** Elements in Circular Queue*****
Data=20 Priority=1
Data=30 Priority=2
Data=10 Priority=3
Enter the Choice(1.Insertion 2.Deletion 3.Display 4.Exit):2
Removed Element: 20
Enter the Choice(1.Insertion 2.Deletion 3.Display 4.Exit):3
**** Elements in Circular Queue*****
Data=30 Priority=2
Data=10 Priority=3
Enter the Choice(1.Insertion 2.Deletion 3.Display 4.Exit):
```

5.6 DOUBLE ENDED QUEUE [April 15, 18]

- A deque, also known as a double ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection.
- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear).
- That means, we can insert at both front and rear positions and can delete from both front and rear positions.

5.6.1 Definition

- A DeQue is a linear list in which elements can be added or removed at either end but not in the middle, i.e. elements can be added or deleted at front or rear end.

Array Representation of DeQue:

- Like a simple queue, DeQue is also represented by array. We maintain two pointers LEFT and RIGHT which indicates the left and right position of the DeQue.
- Let us consider a DeQue which is shown in Fig. 5.20.

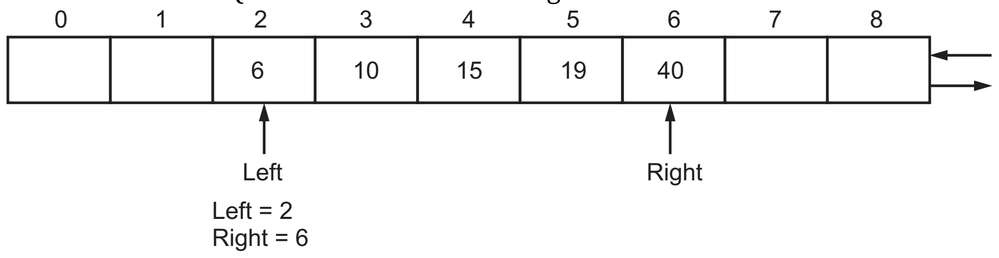


Fig. 5.20: Array Representation of DeQue

Applications of DeQue:

- DeQue is useful where the data to be stored has to be ordered compact storage is needed and the retrieval to data elements has to be faster.

5.6.2 Types of DeQue

- The two types of DeQue are explained below:
  1. **Input Restricted DeQue:** A DeQue which allows insertion at only at one end of the list but allows deletion at both the ends of the list is called Input Restricted DeQue. The input restricted DeQue allows insertion at one end (it can be either front or rear) only.
  2. **Output Restricted DeQue:** A DeQue which allows deletion at only at one end of the list but allows insertion at both the ends of the list is called Output restricted Deque. The output restricted dequeue allows deletion at one end (it can be either front or rear) only.

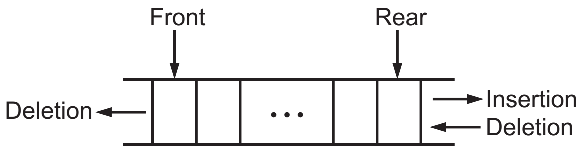


Fig. 5.21: Input Restricted DeQue

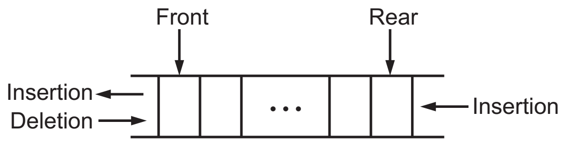


Fig. 5.22: Output Restricted DeQue

5.6.3 Operations performed on Deques

- The common operations on dequeue are:
  1. **insertFirst():** To add element at the end of a dequeue.
  2. **insertLast():** To delete element at the beginning of a dequeue.
  3. **deleteFirst():** To remove first element of the dequeue.
  4. **deleteLast():** To remove last element of the dequeue.
  5. **isFull():** It finds out whether the dequeue is full.
  6. **isEmpty():** It finds out whether the circular dequeue is empty.
  7. **create():** To create an empty dequeue.

Program 5.11: Program to implement DeQue.

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
int deque[MAX];
int left=-1, right=-1;
void insert_right(void);
void insert_left(void);
void delete_right(void);
void delete_left(void);
void display(void);
```

```
int main()
{
    int choice;
    do
    {
        printf("\n1.Insert at right ");
        printf("\n2.Insert at left ");
        printf("\n3.Delete from right ");
        printf("\n4.Delete from left ");
        printf("\n5.Display ");
        printf("\n6.Exit");
        printf("\n\nEnter your choice ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                    insert_right();
                    break;
            case 2:
                    insert_left();
                    break;
            case 3:
                    delete_right();
                    break;
            case 4:
                    delete_left();
                    break;
            case 5:
                    display();
                    break;
        }
    }while(choice!=6);
    getch();
    return 0;
}
//-------INSERT AT RIGHT-------
void insert_right()
{
    int val;
    printf("\nEnter the value to be added ");
    scanf("%d",&val);
    if( (left==0 && right==MAX-1) || (left==right+1) )
    {
        printf("\nOVERFLOW");
    }
    if(left==-1)        //if queue is initially empty
    {
        left=0;
        right=0;
    }
    else
    {
        if(right==MAX-1)
            right=0;
        else
            right=right+1;
    }
    deque[right]=val;
}
//-------INSERT AT LEFT-------
void insert_left()
{
    int val;
    printf("\nEnter the value to be added ");
    scanf("%d",&val);
    if( (left==0 && right==MAX-1) || (left==right+1) )
    {
        printf("\nOVERFLOW");
    }
    if(left==-1)        //if queue is initially empty
    {
        left=0;
        right=0;
    }
    else
    {
        if(left==0)
            left=MAX-1;
        else
            left=left-1;
    }
    deque[left]=val;
}
//-------DELETE FROM RIGHT-------
```

```
void delete_right()
{
    if(left==-1)
    {
        printf("\nUNDERFLOW");
        return;
    }
    printf("\nThe deleted element is %d\n", deque[right]);
    if(left==right)          //Queue has only one element
    {
        left=-1;
        right=-1;
    }
    else
    {
        if(right==0)
            right=MAX-1;
        else
           right=right-1;
    }
}
//-------DELETE FROM LEFT-------
void delete_left()
{
    if(left==-1)
    {
        printf("\nUNDERFLOW");
        return;
    }
    printf("\nThe deleted element is %d\n", deque[left]);
    if(left==right)          //Queue has only one element
    {
        left=-1;
        right=-1;
    }
    else
    {
        if(left==MAX-1)
```

```
            left=0;
        else
            left=left+1;
    }
}
//-------DISPLAY-------
void display()
{
    int front=left, rear=right;
    if(front==-1)
    {
        printf("\nQueue is Empty\n");
        return;
    }
    printf("\nThe elements in the queue are: ");
    if(front<=rear)
    {
        while(front<=rear)
        {
            printf("%d\t",deque[front]);
            front++;
        }
    }
    else
    {
        while(front<=MAX-1)
        {
            printf("%d\t",deque[front]);
            front++;
        }
        front=0;
        while(front<=rear)
        {
            printf("%d\t",deque[front]);
            front++;
        }
    }
    printf("\n");
}
```

5.7 APPLICATIONS OF QUEUES [April 18]

- Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios:
  1. Queues are used in job scheduling by the operating system. For examples: Printer queue, Process queue etc.
  2. Queues are used in simulation. For example, in Airport simulation, to keep the track of problems like number of plane processed, the average time spent waiting, etc. queues are used.
  3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, first come first served.
  4. To process the jobs in multi-user systems queues are used.
  5. In batch programming to process the job sequentially.

5.7.1 CPU Scheduling in Multiprogramming Environment

- As we know that in multiprogramming environment, multiple processes run concurrently or simultaneously to increase CPU utilization.
- All the processes that are residing in memory and are ready to execute are kept in a list referred as ready queue. It is the job of scheduling algorithm to select a process from the processes and allocate the CPU to it.
- In a multiprogramming environment, a single CPU has to serve more than one program simultaneously. The jobs in the multiprogramming environment are classified into following three groups:
  1. **Interrupts to be Serviced:** Number of devices and terminals are connected with the CPU and they may interrupt at any moment to get a service from it.
  2. **Interactive Users to be Serviced.**
  3. **Batch Jobs to be Serviced:** Now the jobs in the multiprogramming environment are put in the queue and assigned to the CPU, based on the priority given to the queues. Here, multilevel queue scheduling is used. Fig. 5.23 shows its arrangement.

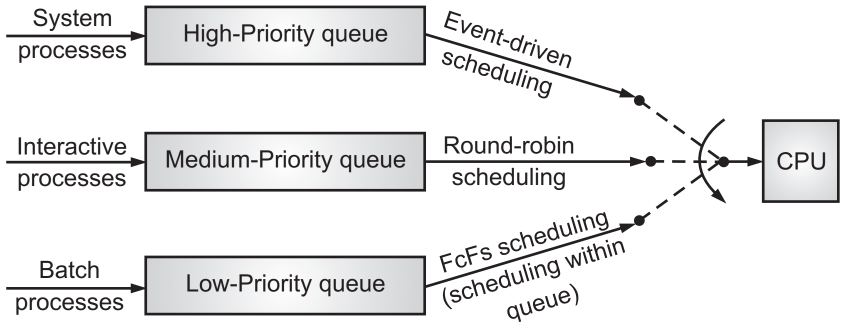


Fig. 5.23: Process scheduling with Multi-level queues

- Processor scheduling is a one of the basic service of any of the operating system. It is fact that any of the computer resource are scheduled before use by any of the process.
- The CPU is one of the primary computer resources. In the design of the operating system, CPU scheduling is central to it. When two or more processes are ready to run, the OS is responsible for deciding which one will proceed.
- The module of the OS related with this decision is called scheduler and the algorithm it uses is called scheduling algorithm.
- CPU switches among a number of processes in multi-programmed environment. CPU scheduling has its great role in multiprogramming environment.
- The one of the basic objective of the operating system should be to ensure that processor will remain always busy. In order to maximize the CPU utilization some processes should be running all the times.
- To implement multi-programming environment, multi-level queue scheduling algorithm is used.
- In multi-level queue algorithm, the ready queue is partitioned into multiple queues (See Fig. 5.24). The processes are assigned to the respective queues.
- The higher priority processes are executed before the lower priority processes. For example, no batch process can run unless all the system.
- The processes from the highest priority queue are serviced by the CPU until the queue becomes empty. Then the CPU switches to the queue of interactive processes which has a medium priority.
- Then the CPU switches to the queue of batch processes which has the lowest priority. Any low-level priority processes may be pre-empted by a higher priority process.
  1. The main drawback of a multi-level queue strategy is that the processes in the lower priority queue may starve for a long time.
  2. Another drawback is that processes occupy a fixed queue, i.e., processes don't move among queues.
- To overcome above problems, the following methods may be used:
  1. Time slice is allotted for the different queues. Hence, each queue gets a certain portion of CPU time, so that starvation problem of lower-priority queues can be avoided.
  2. Multi-level feedback queues may be employed. Here, processes with less burst time are placed in the highest pri ority queue. Processes with more burst time occupy lowest priority queue. Pro cesses are also allowed to move among the queues. Fig. 5.24 shows this concept.

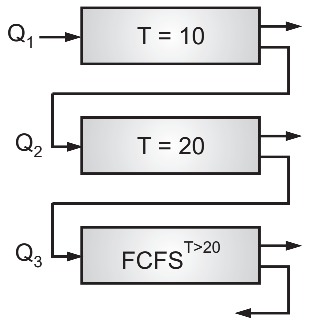


Fig. 5.24: Multilevel feedback queue

- Here, three queues Q_1, Q_2 and Q_1 are maintained. Time slice assigned for Q_1 is 10 ms. Time slice assigned for Q_2 is 20 ms, and time slice assigned for Q_3 is more than 20 ms.
- Processes in Q_1 should be processed within 10 ms. Suppose if a particular process in Q_1 is not processed within 10 ms then it is pushed into the tail end of Q_2.
- Processes in Q_2 should be processed within 20 ms. The processes which are not finished by the CPU within 20 ms are pushed into the tail end of Q_3.
- For Q_3 also, a certain time is assigned. Processes from Q_3 are then are pushed into the tail end of queue Q_1.

5.7.2 Round Robin Algorithm

- The Round Robin (RR) algorithm is one of the CPU scheduling algorithms designed for the time sharing systems.
- In RR algorithm, the CPU is allocated to a process for a small time inter val called time quantum (generally from 10 to 100 milliseconds).
- Whenever, a new process enters, it is inserted at the end of the ready queue. The CPU scheduler picks the first process from the ready queue and processes it until the time quantum elapsed.
- Then CPU switches to the next process in the queue and first process is inserted at the end of the queue if it has not been finished. If the process is finished before the time quantum, the process itself will release the CPU voluntarily and the process will be deleted from the ready queue.
- This process continues until all the processes are finished. When a process is finished, it is deleted from the queue. To implement the RR algorithm, a circular queue can be used.
- Here, circular queue is used to implement RR algorithm. There are n processes P_1, P_2, … P_n required to be served by the CPU (only one CPU).
- Different processes require different execution time. The sequence of processes arrival are P_1, P_2, … P_n. This algorithm decides a smallest unit of time called a time quantum or time slice T.
- This time slice is usually from 10 to 100 ms. Here, CPU starts servicing process $P_1 \cdot P_1$ gets CPU time for T instant of time. Then CPU switchs to P_2 and so on.

- When CPU reaches the end of time quantum of P_n it returns to P_1 and the same process will be repeated.
- Suppose if some process finishes its execution before its time quantum is finished the process simply releases the CPU. So the next process gets the CPU and will be serviced.

Example: Consider three processes given below, Here, time slice is 2:

Process	Burst-time
P_1	5
P_2	3
P_3	10

- The RR scheduling for these processes are as follows:

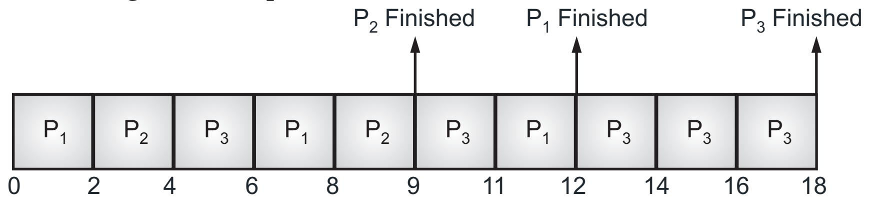


Fig. 5.25

- The advantage of RR scheduling is that the average turnaround time is reduced. Note that the turn around time means that the time gap between the time of completion and the time of arrival of the process.
- In time-sharing systems, any process may arrive at any instant of time. They are placed in the queue (at the end). When a Process finished its execution, that process is deleted from the queue.

PRACTICE QUESTIONS

Q.I Multiple Choice Questions:

1. A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as ______.
   (a) Stack (b) Queue
   (c) Tree (d) Linked list
2. A queue follows ______.
   (a) FIFO (First In First Out) principle
   (b) LIFO (Last In First Out) principle
   (c) Ordered array
   (d) Linear tree
3. The initial configuration of a queue is A, B, C, D (A is in the front end). To get the configuration D, C, B, A, one needs a minimum of ______.
   (a) 2 deletions and 3 additions (b) 3 deletions and 2 additions
   (c) 3 deletions and 3 additions (d) 3 deletions and 4 additions

4. A simple queue, if implemented using an array of size MAX_SIZE, gets full when
   (a) Rear = MAX_SIZE – 1 (b) Front = (rear + 1)mod MAX_SIZE
   (c) Front = rear + 1 (d) Rear = front
5. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into EMPTY queue?
   (a) Only front pointer (b) Only rear pointer
   (c) Both front and rear pointer (d) None
6. In linked list implementation of a queue, where does a new element be inserted?
   (a) At the head of link list
   (b) At the tail of the link list
   (c) At the center position in the link list
   (d) None
7. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue?
   (a) Only front pointer (b) Only rear pointer
   (c) Both front and rear pointer (d) None of the front and rear pointer
8. If the MAX_SIZE is the size of the array used in the implementation of circular queue. How is rear manipulated while inserting an element in the queue?
   (a) rear=(rear%1)+MAX_SIZE (b) rear=rear%(MAX_SIZE+1)
   (c) rear=(rear+1)%MAX_SIZE (d) rear=rear+(1%MAX_SIZE)
9. A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index.
   (a) 0 (b) 7
   (c) 9 (d) 10
10. If the MAX_SIZE is the size of the array used in the implementation of circular queue, array index starts with 0, front point to the first element in the queue, and rear point to the last element in the queue. Which of the following condition specify that circular queue is EMPTY?
    (a) Front=rear=0 (b) Front= rear=-1
    (c) Front=rear+1 (d) Front=(rear+1)%MAX_SIZE
11. An array of size MAX_SIZE is used to implement a circular queue. Front, Rear, and count are tracked. Suppose front is 0 and rear is MAX_SIZE -1. How many elements are present in the queue?
    (a) Zero (b) One
    (c) MAX_SIZE-1 (d) MAX_SIZE

Answers

1. (b)	2. (a)	3. (c)	4. (a)	5. (c)	6. (b)	7. (b)	8. (c)	9. (a)	10. (b)	11. (d)

Q.II Fill in the Blanks:

1. A ______ is a group of linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as queue.
2. ______ queue is finite collection of elements where each element is assigned a priority.
3. Circular queue is also known as ______.
4. Queue is a linear data structure in which items are inserted at one end called ______ and deleted from the other end called as Front.
5. A ______ is a special type of queue that allows insertion and deletion of elements at both ends, i.e., front and rear.
6. Queue is a non-primitive ______ data structure.
7. In a priority queue, elements are inserted as per their ______ levels.
8. A data structure in which elements can be inserted or deleted at/from both the ends but not in the middle is known as ______.
9. In ______ restricted deque, deletions can be done only at one of the ends, while insertions can be done on both ends.
10. The ______ operation is used for creating a deque.
11. In a multiprogramming environment, a ______ CPU has to serve more than one program simultaneously.

Answers

1. queue	2. Priority	3. Ring buffer	4. Rear	5. deque	6. linear
7. priority	8. deque	9. output	10. create	11. single	

Q.III State True or False:

1. A queue is an ordered sequential group of elements in which permits insertion of new element at one end of deletion of an element at other end.
2. Queue is First-In-First-Out (FIFO) list.
3. Queue is not useful in simulation application.
4. In a priority queue, elements are deleted as per their priority levels.
5. A double ended queue supports insertion and removal of elements at both ends.
6. The two variants of a double-ended queue are Input restricted deque and Output restricted deque.
7. In input restricted deque, insertions can be done only at one of the ends, while deletions can be done from both ends.
8. A normal queue, if implemented using an array of size MAX_SIZE, gets full when Rear = MAX_SIZE – 1.
9. Circular queue is a linear data structure in which the operations are performed based on FIFO principle and the last position is connected back to the first position to make a circle.

Answers

1. (T)	2. (T)	3. (F)	4. (T)	5. (T)	6. (T)	7. (T)	8. (T)	9. (T)

Q.IV Answer the following Questions:

(A) Short Answer Questions:

1. What is queue?
2. List any two applications of queue.
3. What are the types of queues?
4. Define priority queue.
5. What is deque?
6. Define the term circular queue.
7. List any two applications of priority queue.
8. What are the operations performed on linear queue?
9. Give two differences between linear, priority and circular queue.

(B) Long Answer Questions:

1. Define the term queue? How to create it? Explain with example.
2. With the help of program explain how to implement queue using array?
3. What is priority queue? How to create it?
4. How to insert and delete elements in priority queue?
5. What is circular queue? Explain with C program.
6. What is linear queue? How to implement it? Explain in detail.
7. Describe circular queue implementation using array.
8. Define deque. How to implement it? Explain with example.
9. List applications of linear queue.
10. Compare priority and circular queues.
11. What are the types of deque? Explain diagrammatically. Also compare them.
12. Describe the term CPU Scheduling in multiprogramming environment in detail.

UNIVERSITY QUESTIONS AND ANSWERS

April 2015

1. List the types of Priority Queue. **[1 M]**

Ans. Refer to Section 5.5.3.

2. Write a 'C' function to add and delete an element from a Linear Queue (Dynamic Implementation). **[5 M]**

Ans. Refer to Section 5.4.2.

3. Define the term dequeue. **[1 M]**

Ans. Refer to Section 5.6.1.

April 2016

1. 'Queue full condition in a linear queue necessary implies that there is no free space in array'. State True/False. **[1 M]**

Ans. Refer to Section 5.4.1, Point (3).

2. Define priority queue. Explain its types. **[3 M]**

Ans. Refer to Section 5.5.3.

3. Consider a circular queue of characters of size 5. Initially queue contents are as follows:

Front	Rear
1	2

Circular queue

	P	Q		
0	1	2	3	4

Show the value of front, rear and contents of queue for the following operations:

(i) addqueue (R)
(ii) remove queue ( )
(iii) remove queue ( )
(iv) add queue (S)
(v) add queue (T)
(vi) add queue (U)
(vii) remove queue ( )
(viii) addqueue (V) **[4 M]**

Ans. Refer to Section 5.5.2.2.

October 2016

1. Write a 'C' function to add and delete element from Linear Queue. **[5 M]**

Ans. Refer to Section 5.4.2.

2. Define the term Priority Queue. **[1 M]**

Ans. Refer to Section 5.5.3.1.

April 2017

1. Write the statement to increment rear in a circular queue implemented using array. **[1 M]**

Ans. Refer to Section 5.5.2.3, Point (3).

2. Write a 'C' function for adding and deleting elements from a circular queue. **[5 M]**

Ans. Refer to Page .

3. Define the term Priority Queue. **[1 M]**

Ans. Refer to Section 5.5.3.1.

October 2017

1. Define Priority Queue. **[1 M]**

Ans. Refer to Section 5.5.3.1.

2. Write a 'C' functions to ADD and REMOVE from circular queue implemented using array. **[5 M]**

Ans. Refer to Section 5.4.2.

April 2018

1. Write any two applications to Queue. **[1 M]**

Ans. Refer to Section 5.7.

2. Define Priority Queue. Explain its types. **[5 M]**

Ans. Refer to Section 5.5.3.1.

3. Define the term Double Ended Queue. **[1 M]**

Ans. Refer to Section 5.6.1.

■■■

Notes